RINALDI MUNIR

ALGORITMA & PEMROGRAMAN

DALAM BAHASA PASCAL dan C

Edisi Revisi

Prakata

Pemrograman komputer sudah menjadi kebutuhan penting di era teknologi informasi ini. Meskipun paket program komersil sudah banyak tersedia untuk membantu menyelesaikan masalah, namun membuat program sendiri untuk kebutuhan spesifik tetap diperlukan. Program komersil tidak dapat menyelesaikan masalah yang beraneka ragam dan kompleks.

Saat ini aktivitas menulis program tidak lagi didominasi orang-orang dari lingkungan pendidikan komputer dan informatika. Siswa SD, SMP, SMA, dan mahasiswa yang bukan berlatar belakang pendidikan komputer banyak yang mampu menulis program. Pemrograman bukan pekerjaan yang sulit, siapa pun dapat melakukannya asalkan dia mempunyai minat dan motivasi untuk belajar. Semakin hari kegiatan memprogram sudah menjadi kebutuhan penting banyak orang, mulai dari sekadar hobi sampai kebutuhan untuk mendukung studi. Sebagian besar dari mereka memperoleh pengetahuan pemrograman dari *learning by doing*. Memang banyak literatur, baik cetak maupun *online*, yang dapat digunakan sebagai referensi belajar pemrograman, tetapi sebagian besar literatur tersebut berisi materi bahasa pemrograman atau pemrograman dengan bahasa/kakas tertentu.

Memprogram tidak hanya sekadar menulis program yang "asal jadi" dan memberikan hasil sesuai yang diharapkan. Memprogram itu perlu metodologi agar program yang ditulis adalah program yang efektif. Memang benar banyak orang yang bisa membuat program, tetapi hanya sedikit yang bisa memprogram dengan baik.

Buku ini menyajikan materi yang lengkap mengenai algoritma dan pemrograman. Pembaca diajak untuk tidak memfokuskan pada bahasa pemrograman, tetapi pada langkah-langkah pemecahan masalah. Langkah-langkah pemecahan masalah, yang kemudian dinamakan algoritma, adalah hal yang paling inti di dalam pemrograman. Algoritma harus independen dari bahasa pemrogaraman dan komputer yang mengimplementasikannya. Jika algoritma pemecahan masalah sudah dirancang dengan baik, maka langkah berikutnya adalah menuliskan algoritma tersebut ke dalam notasi bahasa pemrograman. Belajar memecahkan masalah dan menuliskan langkah-langkahnya, itulah yang ditekankan di dalam buku ini. Di dalam buku ini Penulis menggunakan notasi *pseudo-code* untuk menuliskan algoritma. Notasi ini dibuat sedemikian rupa untuk memudahkan pemrogram menerjemahkannya ke dalam bahasa pemrograman.

Buku ini disusun dari pengalaman penulis mengajar kuliah algoritma pemrograman lebih dari 10 tahun, baik di ITB maupun di luar ITB. Berbekal pengalaman mengajar dan referensi yang cukup banyak (lihat judul-judul buku yang dijadikan referensi di dalam Daftar Pustaka), maka buku ini dihadirkan kepada pembaca yang ingin mempelajari pemrograman. Buku ini dibagi menjadi 19 Bab. Penjelasan singkat masing-masing bab dipaparkan di bawah ini.

Bab 1, **Apakah Algoritma dan Pemrograman Itu?**, mengantarkan pembaca untuk memahami apa yang disebut algoritma dan apa yang disebut pemrograman. Diberikan juga contoh-contoh masalah dalam kehidupan sehari-hari dan langkah-langkah pemecahannya (dalam notasi algoritmik deksriprif).

Bab 2, **Struktur Dasar Algoritma**, mengenalkan tiga konstruksi (struktur) dasar di dalam algoritma, yaitu runtunan, pemilihan, dan pengulangan. Contoh-contoh ketiga konstruksi ini di dalam kehidupan sehari-hari diberikan supaya pembaca dapat melihat bahwa algoritma bukan sesuatu yang sulit dipahami.

Bab 3, **Notasi Algoritmik**, menjelaskan notasi *pseudo-code* yang digunakan di dalam buku ini. Notasi *pseudo-code* mirip dengan notasi di

dalam bahasa pemrograman, tujuannya untuk memudahkan penerjemahan algoritma ke dalam bahasa pemrograman. Pada setiap bab nanti diberikan tabel translasi dari notasi *pseudo code* ke notasi bahasa pemrograman (dalam buku ini hanya bahasa *Pascal* dan bahasa *C*).

Bab 4, **Tipe, Operator dan Ekspresi**, menjelaskan tipe data yang dipakai di dalam pemrograman, operator yang memanipulasinya, dan cara menuliskan ekspresi.

Bab 5, **Runtunan**. Di sini pembaca diajak secara mendalam untuk memahami struktur program yang paling dasar, yaitu runtunan. Runtunan adalah deretan instruksi yang dikerjakan sesuai urutan penulisannya.

Bab 6, **Pemilihan**, berisi struktur program yang memiliki satu atau lebih pilihan aksi yang dilakukan. Notasi *if-then*, *if-then-else*, dan *case* kita gunakan untuk menyatakan konstruksi pemilihan.

Bab 7, **Pengulangan**, berisi struktur pengulangan, yaitu sebuah pernyataan yang di dalamnya berisi satu atau lebih aksi yang dikerjakan secara berulang-ulang. Notasi *for*, *while* dan *repeat-until* digunakan untuk menyatakan konstruksi pengulangan.

Bab 8, **Contoh-contoh Pemecahan Masalah I**, berisi pembahasan beberapa masalah yang terkait dengan materi pada bab-bab sebelumnya, beserta algoritmanya.

Bab 9, **Pengantar Pemrograman Modular**, mengenalkan konsep program yang dipecah menjadi sejumlah modul (atau sub-program). Sub-program dapat dinyatakan sebagai prosedur atau fungsi.

Bab 10, **Prosedur**, menyajikan sub-program yang dinyatakan sebagai prosedur. Sebuah prosedur pada hakikatnya adalah sebuah program tetapi melakukan tugas yang spesifik. Prosedur dapat dipanggil dari program lain dengan atau tanpa melewatkan parameter pada saat pemanggilan.

Bab 11, **Fungsi**, menyajikan sub-program yang dinyatakan sebagai fungsi. Sebuah fungsi adalah program yang menghasilkan keluaran yang digunakan di dalam program pemanggil. Fungsi dapat dipanggil dari program lain dengan atau tanpa melewatkan parameter pada saat pemanggilan. Di dalam bab ini juga dijelaskan kapan menggunakan fungsi dan kapan menggunakan prosedur.

Bab 12, **Larik**, memaparkan tipe data terstruktur yang penting. Larik menyimpan data secara temporer di dalam memori. Larik adalah tipa data terstruktur yang sederhana dan banyak dipakai karena data yang disimpan di dalamnya dapat diakses hanya melalui indeksnya.

Bab 13, **Contoh-contoh Pemecahan Masalah II**, berisi pembahasan beberapa masalah yang terkait dengan materi prosedur, fungsi, dan larik.

Bab 14, **Matriks**, menyajikan larik dwi-matra, yaitu larik yang yang disusun dalam baris dan kolom. Elemen matriks diacu melalui indeks baris dan kolomnya.

Bab 15, **Algoritma Pencarian**, menjelaskan algoritma-algoritma dasar yang digunakan untuk melakukan pencarian data di dalam larik. Algoritma pencarian yang mendasar adalah algoritma pencarian beruntun (*sequential search*) dengan berbagai variasinya dan algoritma pencarian biner (*binary search*).

Bab 16, **Algoritma Pengurutan**, menjelaskan algoritma-algoritma pengurutan data yang fundamental, yaitu *bubble sort, insertion sort, dan selection sort.* Algoritma *shell sort* diberikan sebagai pengembangan algoritma *insertion sort.* Saat ini ada puluhan algoritma pengurutan, tetapi kita tidak perlu mengetahui semuanya, cukup hanya beberapa algoritma pengurutan yang mendasar saja.

Bab 17, **Arsip Beruntun**, menyajikan algoritma pemrosesan data di dalam arsip beruntun (*sequential file*). Pada dasarnya tidak ada konsep yang baru di sana, kita tetap menggunakan tiga konstruksi dasar serta konsep pemrograman modular. Perbedaannya dengan larik hanya satu, data yang diproses tidak berada di dalam memori utama komputer, tetapi di dalam memori sekunder.

Bab 18, **Algoritma rekursif**, menjelaskan konsep rekursifitas yang membuat program menjadi *powerfull* dan alami. Ini materi yang agak sulit dipelajari, tetapi aplikasinya di dalam pemrograman sangat penting.

Bab 19, **Contoh-contoh Pemecahan Masalah III**, berisi pembahasan beberapa masalah yang terkait dengan materi algoritma pencarian, algoritma pengurutan, matriks, arsip beruntun, dan algoritma rekursif.

Demikianlah paparan ringkas isi buku ini. Mudah-mudahan isi buku ini dapat memberikan pencerahan dan pengetahuan kepada pembacanya.

Penulis mengucapkan terima kasih kepada pihak yang baik langsung atau tidak langsung banyak membantu penulis mengerti dan memahami dunia pemrograman. Kepada dosen senior yang mengajarkan banyak hal, kepada teman-teman yang memberikan saran dan pencerahan, dan kepada Penerbit Informatika yang menerbitkan buku ini.

Kata Pengantar

Buku ini merupakan penggabungan dari dua buah buku terdahulu: *Algoritma dan Pemrograman dalam Bahasa Pascal dan C (Buku 1)* dan *Algoritma dan Pemrograman dalam Bahasa Pascal dan C (Buku2)*. Penggabungan ini dimaksudkan agar materi algoritma dan pemrograman dapat dipahami secara menyeluruh dan terintegrasi. Penggabungan ini juga sekaligus untuk memperbaiki beberapa kesalahan yang terdapat pada buku sebelumnya.

Cara penyajian materi disusun dalam bahasa yang lebih mudah dipahami; setiap pokok bahasan dipertajam pembahasannya. Contoh-contoh pemecahan masalah diperbanyak, termasuk soal-soal latihan. Cara terbaik untuk mempelajari buku ini adalah dengan mempraktekkannya (*learning by doing*).

Buku ini memuat materi yang lengkap, dimulai dengan konsep dasar algoritma, struktur dasar pembangun algoritma (runtunan, pemilihan, pengulangan), fungsi dan prosedur, larik (*array*), matriks, algoritma pencarian, algoritma pengurutan, arsip beruntun, dan algoritma rekursif.

Akhirnya, sebagai karya manusia biasa, Penyusun menyadari pasti banyak kekurangan dan kelemahan yang terdapat di dalam buku ini. Untuk itu, Penyusun terbuka terhadap kritik dan saran dari Pembaca. Jika Anda ingin mengirimkan surat, silakan layangkan *e-mail* Anda ke rinaldi@informatika.org.

Wassalam

Bandung, Agustus 2007

Penyusun

Daftar Isi

1

Apakah Algoritma dan Pemrograman Itu?

Pemrograman sudah menjadi kegiatan yang sangat penting di era eknologi informasi. Program yang berjalan di komputer *desktop*, *laptop*, telepon genggam, *PDA*, dan sebagainya, tidak tercipta begitu saja, tetapi ditulis melalui proses analisis dan perancangan yang cermat. Sebuah program komputer pada dasarnya mengimplementasikan suatu algoritma. Dengan kata lain, algoritma adalah ide dibalik program komputer apa pun. Tetapi, tahukah Anda apakah algoritma itu? Apakah program? Bab awal ini akan mengantarkan pembaca kepada pemahaman yang baik mengenai algoritma dan pemrograman. Beberapa ilustrasi dalam kehidupan sehari-hari diberikan untuk lebih memperjelas pemahaman kedua istilah ini ini. Pada bab-bab selanjutnya pembaca akan mempelajari lebih mendalam dasar-dasar menulis algoritma dan pemrograman.

1.1 Masalah

Manusia hidup dengan segala masalah yang melingkupinya. Hidup pada dasarnya adalah serangkaian aktivitas menyelesaikan masalah. Dalam Bahasa Indonesia, istilah lain yang sepadan pengertiannya dengan kata "masalah" adalah "persoalan" (*problem*). Di dalam buku ini, kita tidak membedakan makna kedua istilah ini, oleh karena itu kita akan menggunakan kedua istilah ini secara bergantian, kadang-kadang kita sebut masalah kadang-kadang kita namakan persoalan. Menurut [NEA96],

***masalah** adalah pertanyaan atau tugas yang kita cari jawabannya.*

Contoh beberapa masalah dalam kehidupan sehari-hari misalnya:

1. Diberikan setumpuk kartu pasien di sebuah tempat praktek dokter yang tersusun acak. Setiap kartu mempunyai nomor registrasi pasien. Bagaimana mengurutkan kartu-kartu tadi berdasarkan nomor urut pasien sehingga tersusun dengan nomor kecil di atas dan nomor besar di bawah (Gambar 1.1)? Jawaban dari masalah ini adalah barisan kartu pasien yang sudah terurut dari kecil ke besar.
2. Diberikan sebuah daftar yang berisi nama-nama mahasiswa baru yang diterima di sebuah perguruan tinggi. Daftar tersebut hanya berisi nomor peserta ujian yang diterima. Carilah apakah seorang calon mahasiswa baru yang nomor peserta ujiannya diketahui terdapat di dalam daftar tersebut? Jawaban dari masalah ini adalah "ya" jika nomor tersebut ditemukan di dalam daftar, atau "tidak" jika tidak terdapat di dalamnya.
3. Diberikan daftar nama pejabat baru beserta jumlah kekayaannya. Tentukan pejabat mana yang mempunyai kekayaan paling besar? Jawaban dari masalah ini adalah nama pejabat yang mempunyai kekayaan paling besar.

Tentu masih banyak lagi masalah yang muncul di bidang pekerjaan seseorang. Seorang pustakawan mempunyai masalah dalam mengelola daftar buku yang dipinjam, seorang statistikawan mempunyai masalah mengolah hasil jajak pendapat, seorang mahasiswa tingkat akhir mempunyai masalah dalam menganalisis hasil eksperimen, dan sebagainya.

Gambar 1.1 Mengurutkan kartu, masalah sehari-hari yang banyak kita temui (Sumber: www.infodesign.com.au).

Beberapa masalah yang muncul di dalam dunia nyata banyak memiliki kemiripan substansi. Misalnya masalah mengurutkan tumpukan kartu yang tersusun acak secara substansi sama dengan masalah mengurutkan sekumpulan nilai ujian agar terurut menaik atau menurun. Begitu pula masalah mencari nama pejabat yang mempunyai kekayaan terbesar secara substansi sama dengan masalah mencari nama siswa yang meraih nilai ujian nasional tertinggi. Oleh karena itu, secara generik kita sering

mendeksripsikan masalah-masalah yang muncul di dunia nyata dengan menggunakan beberapa ukuran (parameter), misalnya sebagai berikut:

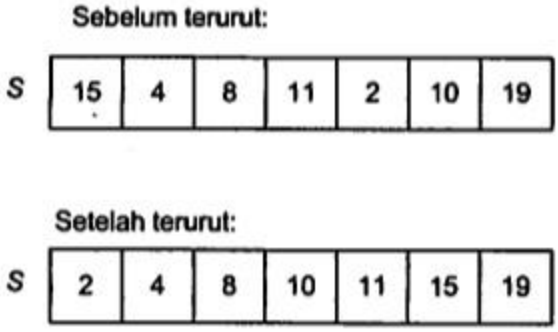
1. [Masalah pengurutan] Diberikan sebuah senarai (*list*) S yang terdiri dari n buah nilai bilangan bulat. Bagaimana mengurutkan n buah nilai tersebut sehingga terurut secara menaik?
2. [Masalah pencarian] Tentukan apakah suatu nilai x terdapat di dalam sebuah senarai S yang beriri n buah bilangan bulat!
3. [Masalah mencari elemen terbesar] Diberikan senarai (*list*) S yang terdiri dari n buah nilai bilangan bulat. Carilah elemen terbesar di dalam senarai tersebut.

Setiap masalah umumnya mengandung satu atau lebih **parameter** yang dinyatakan di dalam masalah tersebut. Misalnya pada contoh masalah pengurutan di atas, S dan n adalah parameter masalah. Parameter ini di dalam pernyataan masalah belum diberi nilai spesifik, dan semua nilai parameter merupakan masukan (*input*) untuk masalah tersebut. Setiap pemberian nilai untuk semua parameter masalah dinamakan **instansiasi masalah** (*instance of a problem*). Jawaban terhadap instansiasi masalah disebut **solusi** [NEA96].

Sebagai contoh, instansiasi masalah untuk masalah pengurutan adalah

$$S = [15, 4, 8, 11, 2, 10, 19], \qquad n = 7$$

dan solusinya adalah barisan nilai terurut $S = [2, 4, 8, 10, 11, 15, 19]$. Gambar 1.2 memperlihatkan keadaan awal S dan keadaan akhir S sesudah diurutkan.



Gambar 1.2 Senarai S yang belum terurut (atas) dan yang sudah terurut menaik (bawah).

1.2 Algoritma

Untuk masalah dengan instansiasi yang kecil, kita dapat menemukan solusinya dengan mudah dan cepat. Bagaimana kalau instansiasi masalah berukuran besar? Misalnya pada masalah pengurutan, jika n = 1000 jelas tidak mudah mengurutkan data sebanyak itu. Oleh karena itu, kita perlu

menuliskan prosedur yang berisi langkah-langkah pengurutan sehingga prosedur tersebut dapat 'dijalankan' oleh sebuah pemroses (komputer, manusia, robot, dan sebagainya) untuk menghasilkan solusi setiap instansiasi masalah pengurutan. Kita katakan langkah-langkah pengurutan itu untuk memecahkan masalah. Prosedur yang berisi langkah-langkah penyelesaian masalah disebut **algoritma**.

> **Algoritma** adalah urutan langkah-langkah untuk memecahkan suatu masalah.

Terdapat beberapa definisi lain dari algoritma – tetapi pada prinsipnya senada dengan definisi yang diungkapkan di atas – yang kita kutip dari berbagai literatur, antara lain:

> **Algoritma** adalah deretan langkah-langkah komputasi yang mentransformasikan data masukan menjadi keluaran [COR92].

> **Algoritma** adalah deretan instruksi yang jelas untuk memecahkan masalah, yaitu untuk memperoleh keluaran yang diinginkan dari suatu masukan dalam jumlah waktu yang terbatas. [LEV03].

> **Algoritma** adalah prosedur komputasi yang terdefinisi dengan baik yang menggunakan beberapa nilai sebagai masukan dan menghasilkan beberapa nilai yang disebut keluaran. Jadi, algoritma adalah deretan langkah komputasi yang mentransformasikan masukan menjadi keluran [COR89].

Mari kita tinjau kembali masalah mengurutkan kartu. Jika kita mempunyai setumpuk kartu yang berjumlah n = 50 buah, maka secara tradisional langkah-langkah yang biasa dilakukan orang kalau mengurutkan kartu adalah dengan mencari kartu bernomor terkecil lebih dahulu, lalu letakkan pada posisi pertama (atas). Selanjutnya, cari kartu dengan nomor terkecil berikutnya, lalu letakkan di bawah kartu pertama. Begitulah, kita mencari kartu dengan nomor terkecil ketiga, keempat, dan seterusnya, sampai hanya tersisa satu kartu (kalau sudah tersisa satu kartu maka tidak perlu diurutkan lagi), dan kita memperoleh tumpukan kartu yang seluruh kartunya sudah terurut dari nomor kecil ke besar. Langkah-langkah pengurutan 50 buah kartu dapat kita tuliskan sebagai berikut (setiap langkah diberi nomor urut 1, 2, 3, dan seterusnya):

1. Cari kartu dengan nomor terkecil
2. Tempatkan kartu tersebut pada posisi paling atas
3. Cari kartu dengan nomor terkecil berikutnya
4. Tempatkan karrtu tersebut di bawah kartu pertama
5. Cari kartu dengan nomor terkecil berikutnya
6. Tempatkan kartu tersebut di bawah kartu kedua

... .

xx. Cari kartu dengan nomor terkecil berikutnya
xxx. Tempatkan kartu tersebut di bawah kartu ke-48
(tersisa satu kartu lagi, tetapi tidak perlu diurutkan lagi)

Perhatikanlah bahwa sebenarnya di dalam rangkaian langkah-langkah di atas terjadi pengulangan dua langkah penting, yaitu (i) cari kartu dengan nomor terkecil, dan (ii) tempatkan kartu tersebut pada posisi yang tepat. Kedua langkah utama ini diulang berkali-kali sampai hanya tersisa sebanyak 1 kartu saja (berarti jumlah pengulangan adalah sebanyak $n = 50 - 1 = 49$ kali). Maka, secara garis besar kita dapat menuliskan langkah-langkah mengurutkan n buah kartu sebagai berikut:

1. Cari kartu dengan nomor terkecil di antara kartu yang tersisa.
2. Tempatkan kartu tersebut pada posisi yang tepat.
3. Ulangi kembali dari langkah 1 sebanyak $n - 1$ kali.

Langkah-langkah yang kita tuliskan di atas itulah yang disebut algoritma. Dikatakan bahwa kita telah menspesifikasikan algoritma pengurutan. Perhatikan bahwa langkah 1 dan 2 masih perlu lebih dirinci lagi. Bagaimana langkah-langkah mencari nomor terkecil? Bagaimana menempatkan nilai terkecil pada posisi yang seharusnya, apakah memerlukan proses pertukaran tempat dengan kartu yang lain? Hal ini akan kita bahasa di dalam Bab 2 nanti.

Dalam kehidupan sehari-hari kita banyak menemukan langkah-langkah pengerjaan sesuatu meskipun kita tidak menyebutnya sebagai algoritma: "Itu bukan algoritma, tapi cara melakukan sesuatu". Contohnya resep membuat masakan (terdapat di dalam tabloid, majalah, atau buku resep masakan), panduan praktikum (terdapat di dalam buku modul praktikum), cara menggunting pola pakaian (terdapat di dalam majalah wanita), langkah-langkah mengaktifkan *voucher* pulsa ponsel (terdapat di belakang kartu prabayar), cara mencari saluran (ditulis di buku panduan TV), cara merakit barang elektronik, dan sebagainya.

Contoh langkah-langkah pengerjaan di dalam resep masakan:

1. Tuangkan satu gelas santan ke dalam wajan.
2. Masukkan bumbu-bumbu yang sudah dihaluskan, aduk hingga merata.
3. Tambahkan garam, merica, dan kecap asin.
4. Masak dengan api sedang sambil diaduk.

Contoh langkah-langkah pengerjaan di dalam praktikum kimia:

1. Siapkan tabung reaksi ukuran 100 cc
2. Teteskan 50 cc larutan H_2SO_4 ke dalam tabung.
3. Campurkan 20 cc larutan NaOH ke dalamnya, aduk hingga tercampur merata.

Contoh langkah-langkah pengisian pulsa kartu prabayar:

1. Tekan #888 lalu tombol t

2. Tekan "2" untuk mulai memasukkan pulsa.
3. Masukkan 14 digit kartu prabayar
4. Akhiri dengan menekan tombol #

Meskipun kita tidak menyebutkan langkah-langkah pengerjaan itu sebagai algoritma, tetapi dalam konteks ini semua di atas adalah algoritma.

Sekarang Anda sudah mulai mengerti bahwa sebuah algoritma mengerjakan sebuah proses. Secara umum, benda yang mengerjakan proses disebut *pemroses* (*processor*). Pemroses tersebut dapat berupa manusia, komputer, robot, atau alat-alat mekanik/elektronik lainnya. Pemroses melakukan suatu proses dengan melaksanakan atau mengeksekusi algoritma yang menjabarkan proses tersebut. Melaksanakan algoritma berarti mengerjakan langkah- langkah di dalam algoritma tersebut. Ibu-ibu yang mencoba resep suatu masakan akan membaca satu per satu langkah pembuatannya, lalu ia mengerjakan proses sesuai yang ia baca.

Setiap masalah mempunyai algoritma pemecahannya. Tugas kita sebagai pemecah masalah (*problem solver*) untuk mendeskripsikan langkah-langkah penyelesaiannya. Di bawah ini diberikan contoh masalah sederhana dalam kehidupan sehari-hari dan algoritma pemecahannya.

1) Misalkan ada dua buah ember atau bejana yang berisi air (larutan) yang berwarna, sebut ember *A* dan *B*. Ember *A* berisi air yang berwarna merah, sedangkan ember *B* berisi air berwarna biru (Gambar 1.3). Volume air di dalam kedua ember sama. Bagaimana mempertukarkan isi kedua ember itu sedemikian sehingga nantinya ember *A* akan berisi air berwarna biru dan ember *B* berisi air berwarna merah.

Ember *A* Ember *B*

Gambar 1.3 Dua buah ember berisi air yang berbeda warna. Ember A berisi air berwarna merah, ember B berisi air berwarna biru.

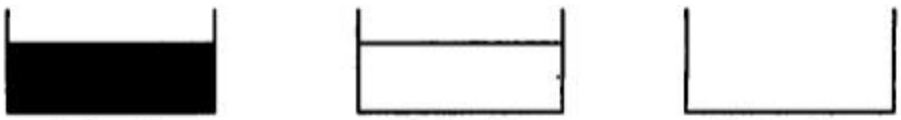
Penyelesaian:

Kita tidak bisa langsung mempertukarkan air di dalam kedua ember tadi begitu saja sebab cara seperti itu menyebabkan terjadinya percampuran. Agar bisa dipertukarkan, kita memerlukan sebuah ember tambahan sebagai tempat penampungan sementara. Misalkan tambahan tersebut adalah ember *C*. Dengan menggunakan ember *C* ini, algoritma mempertukarkan isi kedua buah ember adalah seperti algoritma berikut ini:

```
ALGORITMA memperturkarkan isi dua buah ember, A dan B:
1. Tuangkan air dari ember A ke dalam ember C.
2. Tuangkan air dari ember B ke dalam ember A.
3. Tuangkan air dari ember C ke dalam ember B.
```

Proses pertukaran tersebut diilustrasikan dalam urutan gambar di bawah ini:

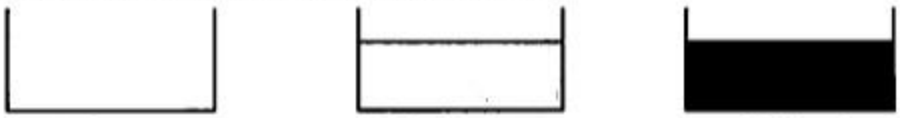
Keadaan awal sebelum pertukaran:



Ember *A* Ember *B* Ember *C*

Langkah pertukaran:

1. Tuangkan air dari ember *A* ke dalam ember *C*



Ember *A* Ember *B* Ember *C*

2. Tuangkan air dari ember *B* ke dalam ember *A*

Ember *A* Ember *B* Ember *C*

3. Tuangkan air dari ember *C* ke dalam ember *B*

Ember *A* Ember *B* Ember *C*

Keadaan Akhir Setelah Pertukaran:

Ember *A* Ember *B* Ember *C*

2) Masih soal ember ☺. Di dalam literatur klasik terdapat persoalan yang dinamakan *water jug problem*. Misalkan Anda mempunyai dua buah

ember, masing-masing bervolume 5-liter dan 3-liter (gambar 1.4). Anda diminta mendapatkan air (dari sebuah danau) sebanyak 4 liter dengan menggunakan bantuan hanya kedua ember tersebut (tidak ada peralatan lain yang tersedia, hanya kedua ember itu saja yang ada!), terserah bagaimana caranya. Anda boleh memindahkan air dari satu ember ke ember lain, membuang seluruh air dari ember, dan sebagainya. Bagaimana algoritmanya?

Gambar 1.4 Dua buah ember, masing-masing bervolume 5 liter dan 3 liter.

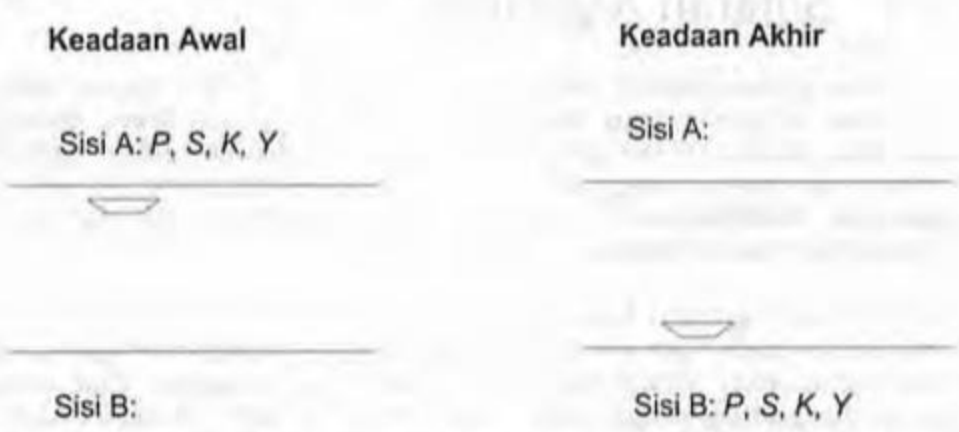
Penyelesaian:

Masalah ini mempunyai banyak kemungkinan penyelesaian. Salah satunya dituliskan sebagai berikut:

```
ALGORITMA mendapatkan air dengan volume 4 liter.
1. Isi penuh ember 3-liter dengan air. {ember 3-liter berisi 3 liter
   air}
2. Tuangkan air dari ember 3-liter ke dalam ember 5-liter. {ember 5-
   liter sekarang berisi 3 liter air}.
3. Isi penuh ember 3-liter dengan air. {ember 3-liter berisi 3 liter
   air}
4. Tuangkan air dari ember 3-liter ke dalam ember 5-liter hingga
   penuh. { di dalam ember 3-liter sekarang tersisa 1 liter air}
5. Buang seluruh air dari ember 5-liter. {ember 5-liter kosong}
6. Tuangkan air dari ember 3-liter ke dalam ember 5-liter. {ember 5-
   liter sekarang berisi 1 liter air }
7. Isi penuh ember 3-liter dengan air. {ember 3-liter berisi 3 liter
   air}
8. Tuangkan air dari ember 3-liter ke dalam ember 5-liter. {ember 5-
   liter sekarang berisi 1 + 3 = 4 liter air}.
```

Hmmm... ternyata cukup repot juga mendapatkan air 4 liter. Mungkin Anda punya langkah penyelesaian yang berbeda, atau mungkin jumlah langkahnya lebih sedikit atau lebih panjang dari algoritma. Perhatikanlah di dalam algoritma di atas kita tambahkan catatan/komentar di dalam tanda kurung kurawal. Komentar ini kelak berguna untuk memahami algoritma. Kita akan membicarakan lebih jauh pentingnya komentar pada bab-bab selanjutnya.

3) Misalkan seorang pemuda tiba di tepi sebuah sungai. Pemuda tersebut membawa seekor kambing, seekor srigala, dan sekeranjang sayur. Mereka bermaksud hendak menyeberangi sungai. Pemuda itu menemukan sebuah perahu kecil di pinggir sungai tetapi sayang hanya dapat memuat satu bawaan saja setiap kali menyeberang. Situasinya dipersulit dengan kenyataan bahwa srigala tidak dapat ditinggal berdua dengan kambing (karena srigala akan memangsa kambing) atau kambing tidak dapat ditinggal berdua dengan sekeranjang sayur (karena kambing akan memakan sayur). Bagaimana algoritma si pemuda menyeberangkan seluruh bawaannya itu sehingga mereka sampai ke seberang sungai dengan selamat. Tentu saja hanya si pemuda yang bisa mendayung dan perahu.



Gambar 1.5 Keadaan awal dan keadaan akhir proses penyeberangan pemuda (P) dan bawaannya yang terdiri dari srigala (S), kambing (K), dan sekeranjang sayur (Y). Perahu hanya dapat memuat satu bawaan saja pada setiap kali menyeberang. Srigala tidak dapat ditinggalkan bersama kambing, begitu pula kambing tidak dapat ditinggalkan bersama sayur.

Penyelesaian:

Misalkan sisi sungai kita namakan *A* dan sisi sungai seberangnya kita namakan *B*. Keadaan awalnya, di sisi *A* ada pemuda (*P*), srigala (*S*), kambing (*K*), dan sayur (*Y*). Keadaan akhir yang kita inginkan adalah di sisi *B* ada pemuda (*P*), srigala (*S*), kambing (*K*), dan sayur (*Y*). Gambar 1.5 memperlihatkan keadaan yang dimaksud.

Algoritma menyeberangkan seluruh bawaan tersebut kita tuliskan seperti di bawah ini:

```
ALGORITMA menyeberangkan pemuda dan bawaannya:
   { sisi A: (P,S,K,Y)   sisi B: (-,-,-,-) }
1. Pemuda menyeberangkan kambing dari sisi A ke sisi B.
   { sisi A: (-,S,K,Y)   sisi B: (P,-,K,-) }
2. Pemuda menyeberang sendiri dari B ke A.
```

```
   { sisi A: (P,S,-,Y)    sisi B: (-,-,K,-) }
3. Pemuda menyeberangkan srigala dari sisi A ke sisi B.
   { sisi A: (-,-,-,Y)    sisi B: (P,S,K,-) }
4. Pemuda menyeberangkan kambing dari sisi B ke sisi A.
   { sisi A: (P,-,K,Y)    sisi B: (-,S,-,-) }
5. Pemuda menyeberangkan sayur dari sisi A ke sisi B.
   { sisi A: (-,-,K,-)    sisi B: (P,S,-,Y) }
6. Pemuda menyeberang sendiri dari B ke A.
   { sisi A: (P,-,K,-)    sisi B: (-,S,-,Y) }
3. Pemuda menyeberangkan kambing dari sisi A ke sisi B.
   { sisi A: (-,-,-,-)    sisi B: (P,S,K,Y) }
```

1.3 Sejarah Algoritma

Algoritma adalah *jantung* ilmu komputer atau informatika. Banyak cabang dari ilmu komputer yang diacu dalam terminologi algoritma, misalnya algoritma perutean (*routing*) pesan di dalam jaringan komputer, algoritma *brensenham* untuk menggambar garis lurus (bidang grafika komputer), algoritma Knuth-Morris-Pratt untuk mencari suatu pola di dalam teks (bidang *information retrievel*), dan sebagainya.

Ditinjau dari asal usul kata, kata "algoritma" sendiri mempunyai sejarah yang cukup aneh. Kata ini tidak muncul di dalam kamus Webster sampai akhir tahun 1957. Orang hanya menemukan kata *algorism* yang berarti proses menghitung dengan angka Arab [KNU73]. Anda dikatakan *algorist* jika Anda menggunakan angka Arab. Para ahli bahasa berusaha menemukan asal kata *algorism* ini namun hasilnya kurang memuaskan. Akhirnya para ahli sejarah matematika menemukan asal mula kata tersebut. Kata *algorism* berasal dari nama penulis buku Arab yang terkenal, yaitu Abu Ja'far Muhammad ibnu Musa al-Khuwarizmi (al-Khuwarizmi dibaca orang Barat menjadi *algorism*). Al-Khuwarizmi (lihat perkiraan wajahnya pada Gambar 1.6, diambil dari www.iescarrus.com) menulis buku yang berjudul *Kitab al jabar wal-muqabala*, yang artinya "Buku pemugaran dan pengurangan" (*The book of restoration and reduction*). Dari judul buku itu kita juga memperoleh akar kata "aljabar" (*algebra*). Perubahan dari kata *algorism* menjadi *algorithm* muncul karena kata *algorism* sering dikelirukan dengan *arithmetic*, sehingga akhiran *-sm* berubah menjadi *-thm*. Karena perhitungan dengan angka Arab sudah menjadi hal yang sudah biasa/ lumrah, maka lambat laun kata *algorithm* berangsur-angsur dipakai sebagai metode perhitungan (komputasi) secara umum, sehingga kehilangan makna aslinya [PAR95]. Dalam bahasa Indonesia, kata *algorithm* diserap menjadi "algoritma".

Gambar 1.6 (a) Sketsa wajah al-Khuwarizmi, (b) al-Khuwarizmi dalam sebuah perangko.

Pada tahun 1950, kata algoritma pertama kali digunakan pada "algoritma Euclidean" (*Euclid's algorithm*). Euclid, seorang matematikawan Yunani (lahir pada tahun 350 M), dalam bukunya yang berjudul *Element* menuliskan langkah-langkah untuk menemukan pembagi bersama terbesar (*common greatest divisor* atau *gcd*), dari dua buah bilangan bulat, m dan n [KNU73] (tentu saja Euclid tidak menyebut metodenya itu sebagai algoritma, baru di abad modernlah orang-orang menyebut metodenya itu sebagai "algoritma Euclidean"). Pembagi bersama terbesar dari dua buah bilangan bulat tak-negatif adalah bilangan bulat positif terbesar yang habis membagi kedua bilangan tersebut.

Misalnya, $m = 80$ dan $n = 12$. Semua faktor pembagi 80 adalah

1, 2, **4**, 5, 8, 10, 16, 20, 40, 80,

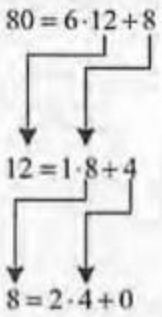
dan semua faktor pembagi 12 adalah

1, 2, 3, **4**, 6, 12,

maka $gcd(80,12) = 4$. Langkah-langkah mencari $gcd(80, 12)$ dengan algoritma Euclidean sebagai berikut:

80 dibagi 12 hasilnya = 6, sisa = 8	(atau: $80 = 6 \cdot 12 + 8$)
12 dibagi 8 hasilnya = 1, sisa = 4	(atau: $12 = 1 \cdot 8 + 4$)
8 dibagi 4 hasilnya = 2, sisa = 0	(atau: $8 = 4 \cdot 2 + 0$)

Karena pembagian yang terakhir menghasilkan 0, maka sisa pembagian terakhir sebelum 0, yaitu 4, menjadi $gcd(80,12)$. Jadi, $gcd(80,12) = gcd(12,8) = gcd(8,4) = gcd(4,0) = 4$. Proses mencari *gcd* dari 80 dan 12 juga dapat diilustrasikan dalam diagram berikut:



Terdapat beberapa versi algoritma Euclidean, salah satu versinya dituliskan di bawah ini.

```
ALGORITMA Eucliedan:
{Diberikan dua buah bilangan bulat tak-negatif m dan n (m ≥ n).
Algoritma Euclidean mencari pembagi bersama terbesar, gcd, dari kedua
bilangan tersebut, yaitu bilangan bulat positif terbesar yang habis
membagi m dan n. }
1. Jika n = 0 maka
     m adalah jawabannya;
     stop.
   tetapi jika n ≠ 0,
     lanjutkan ke langkah 2.
2. Bagilah m dengan n dan misalkan r adalah sisanya.
3. Ganti nilai m dengan nilai n dan nilai n dengan nilai r, lalu ulang
   kembali ke langkah 1.
```

Dengan menggunakan algoritma Eucliedan ini, kita dapat menghitung *gcd* dari dua buah bilangan bulat sembarang secara sistematis.

Contoh-contoh algoritma yang sudah dijelaskan di atas memberi dua pesan penting. Pertama, sebuah algoritma harus benar. Kedua, algoritma harus berhenti, dan setelah berhenti, algoritma memberi hasil yang benar. Menurut Donald E. Knuth dalam bukunya yang berjudul *The Art of Computer Programming* [KNU73], sebuah algoritma harus mempunyai lima ciri penting:

1. Algoritma harus berhenti setelah mengerjakan sejumlah langkah terbatas. Sebagai contoh, tinjau kembali algoritma Euclidean. Pada langkah 1, jika $n = 0$, algoritma berhenti. Jika $n \neq 0$, maka nilai n selalu berkurang sebagai akibat langkah 2 dan 3, dan pada akhirnya nilai $n = 0$. Program yang tidak pernah berhenti mengindikasikan bahwa program tersebut berisi algoritma yang salah.
2. Setiap langkah harus didefinisikan dengan tepat dan tidak berarti-dua (*ambiguous*). Pembaca harus mengerti apa yang dimaksud dengan "m dan n adalah bilangan bulat tak-negatif". Contoh lainnya, pernyataan "bagilah p dengan sejumlah beberapa buah bilangan bulat positif" dapat bermakna ganda. Berapakah yang dimaksud dengan "berapa"? Algoritma menjadi jelas jika langkah tersebut ditulis "bagilah p dengan 10 buah bilangan bulat positif".

3. Algoritma memiliki nol atau lebih masukan (*input*). Masukan ialah besaran yang diberikan kepada algoritma untuk diproses. Algoritma Euclidean mempunyai dua buah masukan, m dan n.
4. Algoritma mempunyai nol atau lebih keluaran (*output*). Keluaran dapat berupa pesan atau besaran yang memiliki hubungan dengan masukan. Algoritma Euclidean mempunyai satu keluaran, yaitu m pada langkah 1, yang merupakan pembagi bersama terbesar dari kedua masukannya.
5. Algoritma harus sangkil (*effective*). Setiap langkah harus sederhana sehingga dapat dikerjakan dalam sejumlah waktu yang masuk akal.

1.4 Program dan Pemrograman

Algoritma baru efektif jika dijalankan oleh sebuah pemroses (*processor*). Pemroses itu bisa manusia, komputer, robot, mesin, dan sebagainya. Pemroses membaca setiap instruksi di dalam algoritma lalu mengerjakannya. Menurut [GOL88], suatu pemroses harus:
1. Mengerti setiap langkah dalam algoritma,
2. Mengerjakan operasi yang bersesuaian dengan langkah tersebut.

Kita memfokuskan pemroses algoritma adalah komputer. Komputer adalah alat bantu untuk menjalankan perintah-perintah di dalam algoritma yang telah "dimasukkan" ke dalamnya. Agar komputer mengerti perintah yang dimaksudkan, maka perintah tersebut harus ditulis dalam bahasa yang dipahami olehnya. Oleh karena itu, algoritma harus ditulis dalam bahasa khusus, yaitu bahasa komputer. Algoritma yang ditulis dalam bahasa komputer dinamakan **program**. Bahasa komputer yang digunakan dalam menulis program dinamakan **bahasa pemrograman**. Orang yang membuat program komputer disebut **pemrogram**, dan kegiatan merancang dan menulis program disebut **pemrograman**. Di dalam pemrograman ada aktivitas menulis kode program, kegiatan ini dinamakan *coding*.

Salah satu dari sekian banyak bahasa pemrograman, dan akan kita pakai di dalam buku ini, adalah bahasa Pascal. Program 1.1 di bawah ini adalah contoh sebuah kode program dalam bahasa Pascal. Program tersebut melakukan pengurutan sekumpulan nilai ujian mahasiswa. Data nilai ujian mahasiswa dibaca dari papan ketik (*keyboard*), lalu diurutkan dari kecil ke besar, dan akhirnya hasil pengurutan ditampilkan di layar peraga (monitor). Untuk sementara, Anda tidak perlu memikirkan bagaimana membuat program ini, karena materi seperti inilah yang akan Anda pelajari di dalam buku algoritma dan pemroraman. Jadi, sabarlah dulu.

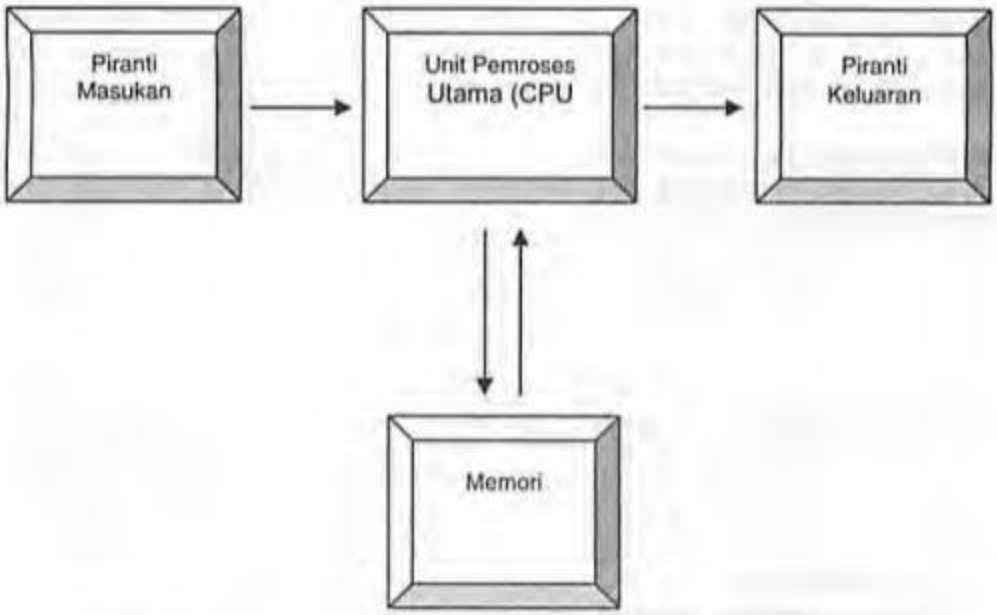
```
program PENGURUTAN;
{ Program untuk mengurutkan nilai ujian sejumlah mahasiswa }
const
   Nmaks = 1000;      { jumlah maksimum data }
```

```
var
   Nilai : array[1..Nmaks] of integer; { tempat menyimpan data }
   j,k,temp, N, Imaks : integer;

begin
   {baca data nilai ujian N orang mahasiswa}
   read(N);
   for j:=1 to N do
     readln(Nilai[j]);
   {endfor}

   {urutkan data dengan langkah-langkah berikut:}
   for j:=1 to N-1 do  { ulangi sebanyak N - 1 kali }
    begin
      {cari nilai terbesar }
      Imaks:=j;
      for k:=j+1 to N do
        if Nilai[k] > Nilai[j] then
          Imaks:=k;
        {endif}
      {endfor}

      {tempatkan nilai terbesar pada posisi yang tepat }
      temp:=Nilai[j];
      Nilai[j]:=Nilai[Imaks];
      Nilai[Imaks]:=temp;
    end; {for}

    { tuliskan nilai yang sudah terurut!}
    for j:=1 to N do
      writeln(Nilai[j]);
    {endfor}
end.
```

Program 1.1 Program pengurutan nilai ujian N orang mahasiswa

Secara garis besar komputer tersusun atas empat komponen utama: piranti masukan, piranti keluaran, unit pemroses utama, dan memori (Gambar 1.7). Unit pemroses utama (*Central Processing Unit - CPU*) adalah "otak" komputer, yang berfungsi mengerjakan operasi-operasi dasar seperi operasi perbandingan, operasi perhitungan, operasi membaca, dan operasi menulis. Memori adalah komponen yang berfungsi menyimpan atau mengingat-ingat. Yang disimpan di dalam memori adalah program (berisi operasi-operasi yang akan dikerjakan oleh *CPU*) dan data atau informasi (sesuatu yang diolah oleh operasi-operasi). Piranti masukan dan keluaran (*I/O devices*) adalah alat yang memasukkan data atau program ke dalam memori, dan alat yang digunakan komputer untuk mengomunikasikan hasil-hasil aktivitasnya. Contoh piranti masukan antara lain papan ketik (*keyboard*), pemindai (*scanner*), tetikus (*mouse*), *joystick*, dan cakram (*disk*). Contoh piranti keluaran adalah layar peraga (*monitor*), pencetak (*printer*), perajah (*plotter*), dan cakram.

Mekanisme kerja keempat komponen di atas dapat dijelaskan sebagai berikut. Mula-mula program dimasukkan ke dalam memori komputer. Ketika program dieksekusi (*execute*), setiap perintah di dalam program yang telah tersimpan di dalam memori dikirim ke *CPU*. *CPU* mengerjakan operasi-operasi yang bersesuaian dengan perintah tersebut. Bila suatu perintah di dalam program meminta data masukan, maka data dibaca dari piranti masukan, lalu dikirim ke *CPU* untuk operasi yang memerlukannya. Bila program menghasilkan keluaran, maka keluaran tersebut ditulis ke piranti keluaran (misalkan dengan mencetaknya ke layar peraga).



Gambar 1.7 Komponen-komponen utama komputer. Komputer terdiri atas Unit Pemroses Utama, Memori, Piranti Masukan, dan Piranti Keluaran.

1.5 Belajar Memprogram dan Belajar Bahasa Pemrograman

Belajar memprogram tidak sama dengan belajar bahasa pemrograman. Belajar memprogram berarti mempelajari metodologi pemecahan masalah, kemudian menuliskan algoritma pemecahan masalah dalam notasi tertentu [LIE96]. Sedangkan belajar bahasa pemrograman berarti belajar memakai suatu bahasa komputer, aturan tata bahasanya, instruksi-instruksinya, tata cara pengoperasian *compiler*-nya, dan memanfaatkan instruksi-instruksi

tersebut untuk membuat program yang ditulis hanya dalam bahasa itu saja [LIE96].

Di dalam pemrograman, kita lebih menekankan pada pemecahan masalah, sementara menulis kode program adalah aktivitas terakhir. Mula-mula kita pikirkan rancangan pemecahan masalah tanpa bergantung pada bahasa pemrograman yang digunakan atau komputer yang menjalankan program itu nanti. Rancangan tersebut berisi urutan langkah-langkah pencapaian solusi yang biasanya ditulis dalam notasi-notasi deskriptif (notasi ini nanti kita namakan **notasi algoritmik**). Karena belajar memprogram yang baik bukanlah belajar membuat program "yang penting hasilnya benar", tetapi perlu dipikirkan membuat program dengan menggunakan skema yang benar. Hal ini, akan membuat program yang kita buat dapat bersih dari kesalahan yang timbul pada waktu eksekusi.

Bila rancangan pemecahan masalah sudah dibuat dengan skema yang benar, maka rancangan tersebut siap dikodekan ke dalam bahasa pemrograman agar program bisa dieksekusi oleh komputer. Di sinilah perlunya kita belajar bahasa pemrograman. Ada banyak bahasa pemrograman yang tersedia saat ini, tetapi desain pemecahan masalah harus dapat diterjemahkan ke dalam bahasa apa pun.

Hingga saat ini terdapat puluhan bahasa pemrograman. Kita dapat menyebutkan antara lain bahasa rakitan (*assembly*), *Fortran*, *Cobol*, *Ada*, *PL/I*, *Algol*, *Pascal*, *C*, *C++*, *Basic*, *Prolog*, *LISP*, *PRG*, bahasa-bahasa simulasi seperti *CSMP*, *Simscript*, *GPSS*, *Dinamo*, dan masih banyak lagi. Belakangan juga muncul bahasa pemrograman baru seperti *Java* dan *C#*. Berdasarkan tujuan aplikasinya, bahasa pemrograman dapat digolongkan menjadi dua kelompok:

1. **Bahasa pemrograman bertujuan khusus (*specific purpose programming language*).** Yang termasuk kelompok ini adalah *Cobol* (untuk terapan bisnis dan administrasi), *Fortran* (aplikasi komputasi ilmiah), bahasa *assembly* (aplikasi pemrograman mesin), *Prolog* (aplikasi kecerdasan buatan), bahasa-bahasa simulasi (*Simscript*), dan sebagainya.
2. **Bahasa pemrograman bertujuan umum (*general purpose programming language*)** yang dapat digunakan untuk berbagai aplikasi. Yang termasuk kelompok ini adalah bahasa *Pascal*, *Basic*, dan *C*, *C++*.

Tentu saja pembagian ini tidak benar-benar kaku. Bahasa-bahasa bertujuan khusus tidak berarti tidak bisa digunakan untuk aplikasi lain. *Cobol* misalnya, dapat juga digunakan untuk terapan ilmiah, tetapi tentu kemampuannya sangat terbatas. Yang jelas, bahasa-bahasa pemrograman yang berbeda dikembangkan untuk bermacam-macam kegunaan yang berbeda pula.

Berdasarkan "kedekatan" bahasa pemrograman apakah lebih condong ke bahasa mesin atau ke bahasa manusia, maka bahasa pemrograman juga dapat dikelompokkan atas dua macam:

1. **Bahasa tingkat rendah.** Bahasa jenis ini dirancang agar setiap instruksinya langsung dikerjakan oleh komputer, tanpa harus melalui *penerjemah (translator).* Contohnya adalah bahasa mesin (*machine language*). Bahasa mesin adalah sekumpulan kode biner (0 dan 1). Setiap perintah dalam bahasa mesin langsung "dimengerti" oleh mesin dan langsung dikerjakan. Bahasa tingkat rendah bersifat primitif, sangat sederhana, dan relatif sulit dipahami manusia. Bahasa *assembly* dimasukkan ke dalam kelompok ini karena notasi yang dipakai dalam bahasa ini merupakan bentuk "manusiawi" dari bahasa mesin, dan untuk melaksanakan instruksinya masih diperlukan penerjemahan (oleh *assembler*) ke dalam bahasa mesin. Bahasa tingkat rendah merupakan bahasa pemrograman generasi pertama yang pernah ditulis orang.

2. **Bahasa tingkat tinggi.** Bahasa jenis ini membuat program menjadi lebih mudah dipahami, lebih "manusiawi", dan lebih dekat ke bahasa manusia (bahasa Inggris terutama). Kelemahannya, program dalam bahasa tingkat tinggi tidak dapat langsung dilaksanakan oleh komputer. Ia perlu diterjemahkan terlebih dahulu oleh sebuah *translator bahasa* (yang disebut kompilator atau *compiler*) ke dalam bahasa mesin sebelum akhirnya dieksekusi oleh *CPU*. Tahapan pemrograman dan pelaksanaan program oleh komputer digambarkan pada Gambar 1.8. Contoh bahasa tingkat tinggi adalah *Pascal, PL/I, Ada, Cobol, Basic, Fortran, C, C++*, dan sebagainya.

Gambar 1.8 Tahapan pelaksanaan program oleh komputer.

Sebenarnya batasan penggolongan bahasa pemrograman itu tidak selalu jelas. Pengertian tentang apa yang dimaksud dengan bahasa tingkat tinggi seringkali berbeda pada beberapa penulis. Ada penulis yang mendefinisikan bahasa tingkat tinggi dari sudut pandang kemudahan pemakaiannya serta orientasinya yang lebih dekat ke bahasa manusia. Les Goldschlager [GOL88] menuliskan spektrum bahasa mulai dari bahasa tingkat tinggi (*Pascal, Ada, PL/I, Cobol*), bahasa tingkat menengah (Bahasa *Assembly, Basic, Fortran*), sampai bahasa tingkat rendah (bahasa mesin). Kita tidak mendebatkan perbedaan cara pengelompokan bahasa pemrograman itu di sini, karena buku ini tidak membahas spesifikasi bahasa pemrograman tetapi, sekali lagi, belajar memprogram!

1.6 Notasi Algoritmik

Di dalam upabab 1.5 sudah kita nyatakan bahwa notasi algoritmik dibuat independen dari spesifikasi bahasa pemrograman dan perangkat keras komputer yang mengeksekusinya. Notasi algoritmik ini dapat diterjemahkan ke dalam berbagai bahasa pemrograman. Analoginya seperti resep membuat kue. Sebuah resep dapat ditulis dalam bahasa apapun, bahasa Inggris, Perancis, Indonesia, Jepang, dan sebagainya. Apa pun bahasa resep, kue yang dihasilkan tetap sama, sebab algoritmanya sama (dengan catatan semua aturan pada resep diikuti). Mengapa bisa demikian? Karena setiap juru masak (yang merupakan pemroses) mampu melakukan operasi dasar yang sama, seperti mengocok telur, menimbang berat gula, dan sebagainya. Jadi, resep membuat kue tidak terikat pada bahasa dan juru masak yang mengerjakannya.

Demikian pula halnya komputer. Meskipun setiap komputer berbeda teknologinya, tetapi secara umum semua komputer dapat melakukan operasi-operasi dasar dalam pemrograman seperti operasi pembacaan data, operasi perbandingan, operasi aritmetika, dan sebagainya. Perkembangan teknologi komputer tidak mengubah operasi-operasi dasar itu, yang berubah hanyalah kecepatan, biaya, atau tingkat ketelitian. Pada sisi lain, setiap program dalam bahasa tingkat tinggi selalu diterjemahkan kedalam bahasa mesin sebelum akhirnya dikerjakan oleh *CPU*. Setiap instruksi dalam bahasa mesin menyajikan operasi dasar yang sesuai, dan menghasilkan efek netto yang sama pada setiap komputer.

Yang perlu dicatat adalah bahwa notasi algoritmik bukan bahasa pemrograman, sehingga siapa pun dapat membuat notasi algoritmik yang berbeda. Hal yang penting mengenai notasi tersebut adalah ia mudah dibaca dan dimengerti. Selain itu, meskipun notasi algoritmik bukan notasi baku sebagaimana pada notasi bahasa pemrograman, namun ketaatasasan terhadap notasi perlu diperhatikan untuk menghindari kekeliruan.

Di bawah ini kita kemukakan beberapa notasi yang digunakan untuk menulis algoritma. Masalah yang dijadikan contoh ilustrasi adalah menghitung pembagi bersama terbesar dengan algoritma Euclidean (lihat kembali Bab 1.2).

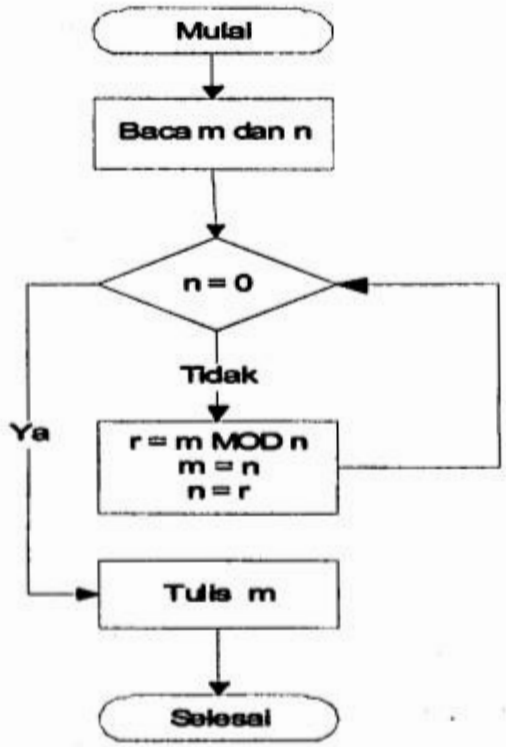
1. Notasi I: menyatakan langkah-langkah algoritma dengan untaian kalimat deskriptif.

```
PROGRAM Euclidean
Diberikan dua buah bilangan bulat tak-negatif m dan n (m ≥ n).
Algoritma Euclidean mencari pembagi bersama terbesar, gcd, dari kedua
bilangan tersebut, yaitu bilangan bulat positif terbesar yang habis
membagi m dan n.

ALGORITMA:
1. Jika n = 0 maka
     m adalah jawabannya;
     stop.
   tetapi jika n ≠ 0,
     lanjutkan ke langkah 2.
2. Bagilah m dengan n dan misalkan r adalah sisanya.
3. Ganti nilai m dengan nilai n dan nilai n dengan nilai r, lalu ulang
   kembali ke langkah 1.
```

Dengan notasi bergaya kalimat ini, deskripsi setiap langkah dijelaskan dengan bahasa yang gamblang. Proses diawali dengan kata kerja seperti 'baca', 'hitung', 'bagi', 'ganti', dan sebagainya, sedangkan pernyataan kondisional dinyatakan dengan 'jika ... maka ...'. Notasi ini baik buat orang awam, tetapi terdapat kesukaran menerjemahkannya langsung ke dalam notasi bahasa pemrograman.

2. Notasi II: menggunakan bagan-alir (*flowchart*)

Pada masa awal perkembangan komputer, ilmuwan menspesifikasikan algoritma sebagai bagan-alir (*flowchart*), yang mengekspresikan algoritma sebagai sekumpulkan bentuk-bentuk geometri (seperti persegi panjang, lingkaran, jajaran genjang, bentuk-intan, dan sebagainya) yang berisi langkah-langkah komputasi. Kotak empat persegi panjang menyatakan proses, sedangkan pernyataan kondisional dinyatakan dengan bentuk intan (*diamond*). Bagan-alir populer pada awal era pemrograman dengan komputer (terutama dengan bahasa *Basic*, *Fortran*, dan *Cobol*). Bagan-alir tidak banyak digunakan lagi saat ini karena ia cenderung tidak praktis dikonversi ke bahasa pemrograman, namun beberapa algoritma sederhana masih sering digambarkan sebagai bagan-alir. Bagan-alir mengambarkan aliran instruksi di dalam program secara visual. Notasi algoritmik dengan diagram alir cocok untuk masalah yang kecil, namun tidak cocok untuk masalah yang besar karena membutuhkan berlembar halaman kertas. Selain itu, pengonversian notasi algoritma ke notasi bahasa pemrograman juga cenderung relatif lebih sukar.



Keterangan:

a. MOD adalah *operator* pembagian bilangan bulat yang menghasilkan sisa hasil pembagian. Contohnya, 9 MOD 2 = 1 karena 9 dibagi 2 = 4 dan memberikan sisa = 1.

b. Di dalam diagram alir di atas ditambahkan instruksi pembacaan nilai *m* dan *n*.

3. Notasi III: menggunakan *pseudo-code*

Para ilmuwan komputer lebih menyukai menuliskan algoritma dalam notasi yang lebih praktis, yaitu notasi *pseudo-code*. *Pseudo-code* (*pseudo* artinya semu atau tidak sebenarnya) adalah notasi yang mirip dengan notasi bahasa pemrograman tingkat tinggi, khususnya (paling sering) bahasa *Pascal* dan *C*. Lebih tepatnya *pseudo-code* adalah campuran antara bahasa alami dengan bahasa pemrograman. Namun, tidak seperti bahasa pemrograman yang direpotkan dengan tanda titik koma (*semicolon*), indeks, format keluaran, aturan khusus, dan sebagainya, maka sembarang versi *pseudo-code* dapat diterima asalkan perintahnya tidak membingungkan pembaca [LIU85]. Jadi, tidaklah mengherankan jika banyak ilmuwan komputer tidak menyepakati suatu *pseudo-code* tertentu, bahkan mereka cenderung membuat "dialek" *pseudo-code* dengan *style* mereka sendiri. Keuntungan menggunakan notasi *pseudo-code* adalah kemudahan mengonversinya–lebih tepat disebut

mentranslasi–ke notasi bahasa pemrograman, karena terdapat korespondensi antara setiap *pseudo-code* dengan notasi bahasa pemrograman. Korespondensi ini dapat diwujudkan dengan tabel translasi dari notasi algoritmik ke notasi bahasa pemrograman apa pun.

Pseudo-code yang digunakan di dalam buku ini banyak diadopsi dari bahasa Pascal, namun tidak benar-benar mematuhi semua sintaks Pascal. Algoritma Euclidean jika kita tulis dengan menggunakan notasi *pseudo-code* adalah seperti di bawah ini:

```
PROGRAM Euclidean
Program untuk mencari gcd dari dua buah bilangan bulat tak-negatif m
dan n (m ≥ n). gcd dari m dan n adalah bilangan bulat positif terbesar
yang habis membagi m dan n.

DEKLARASI:
  m, n : integer   { bilangan bulat yang akan dicari pbt-nya}
  r    : integer   { sisa hasil bagi }

ALGORITMA:
  read(m,n)        { m ≥ n}
  while n ≠ 0 do
    r ← m MOD n    { hitung sisa hasil pembagian }
    m ← n
    n ← r
  endwhile
  { kondisi selesai pengulangan: n = 0, maka gcd(m,n) = m }

  write(m)
```

Kata-kata yang digarisbawahi menyatakan kata-kata penting (*keywords*) yang nantinya berpadanan dengan kata penting pada bahasa komputer yang dipilih untuk mentranslasikan algoritma tersebut. Kalimat yang diapit dengan pasangan kurung kurawal ({ dan }) menyatakan komentar. Komentar berguna untuk lebih memperjelas instruksi yang dituliskan. Peubah yang digunakan di dalam algoritma dituliskan pada bagian **Deklarasi**, sedangkan langkah-langkah penyelesian masalah dinyatakan pada bagian **Algoritma**. Data masukan yang diperlukan algoritma dibaca dengan perintah input, sedangkan keluaran algoritma ditulis dengan perintah output. Karakter "←" menyatakan bahwa nilai di sebelah kanannya diiisikan ke dalam peubah di sebelah kirinya. Kata-kata lain seperti while, integer, dan lain-lain akan dijelaskan di dalam Bab 3.

1.7 Pemrograman Prosedural

Algoritma berisi urutan langkah-langkah penyelesaian masalah. Ini berarti langkah-langkah di dalam algoritma menyatakan proses yang prosedural.

Definisi prosedur menurut Kamus Besar Bahasa Indonesia (KBBI):

Prosedur : 1. tahap-tahap kegiatan untuk menyelesaikan suatu aktivitas;
: 2. metode langkah demi langkah secara eksak dalam memecahkan suatu masalah (KBBI 1988).

Pada pemrograman prosedural, program dibedakan antara bagian data dengan bagian instruksi. Bagian instruksi terdiri atas runtunan (*sequence*) instruksi yang dilaksanakan satu per satu secara berurutan oleh sebuah pemroses. Alur pelaksanaan instruksi dapat berubah karena adanya pencabangan/kondisional. Data yang disimpan di dalam memori dimanipulasi oleh instruksi secara beruntun. Kita katakan bahwa tahapan pelaksanaan program mengikuti pola beruntun atau prosedural. Paradigma pemrograman seperti ini dianamakan **pemrograman prosedural**. Bahasa-bahasa tingkat tinggi seperti *Cobol*, *Basic*, *Pascal*, *Fortran*, dan *C* mendukung kegiatan pemrograman prosedural, karena itu mereka dinamakan juga bahasa prosedural.

Selain paradigma pemrograman prosedural, paradigma pemrograman yang populer saat ini adalah **pemrograman berorientasi objek** (*Object Oriented Programming* atau OOP). Pada paradigma ini, data dan instruksi dibungkus (*encapsulation*) menjadi satu. Kesatuan ini disebut **kelas** (*class*) dan instansiasi kelas pada saat *run-time* disebut **objek** (*object*). Data di dalam objek hanya dapat diakses oleh instruksi yang ada di dalam objek itu saja.

Paradigma pemrograman yang lain adalah **pemrograman fungsional**, **pemrograman deklaratif**, dan **pemrograman konkuren**. Buku ini hanya menyajikan paradigma pemrograman prosedural saja. Paradigma pemrograman yang lain di luar cakupan buku.

Soal Latihan Bab 1

1. Tuliskan beberapa contoh algoritma yang lain dalam kehidupan sehari-hari. Tuliskan juga beberapa contoh langkah di dalam algoritmanya.

2. Tiga pasang suami istri yang sedang menempuh perjalanan sampai ke sebuah sungai. Di situ mereka menemukan sebuah perahu kecil yang hanya bisa membawa tidak lebih dari dua orang setiap kali menyeberang. Penyeberangan sungai dirumitkan oleh kenyataan bahwa para suami sangat pencemburu dan tidak mau meninggalkan istri-istri mereka jika ada lelaki lain. Tulislah algoritma untuk menunjukkan bagaimana penyeberangan itu bisa dilakukan.

3. Misalkan terdapat dua buah ember, masing-masing mempunyai volume 5 liter dan 3 liter. Tuliskan algoritma untuk memperoleh air sebanyak 1 liter dengan hanya menggunakan kedua ember tersebut.

4. Tiga buah cakram yang masing-masing berdiameter berbeda mempunyai lubang di titik pusatnya. Ketiga cakram tersebut dimasukkan pada sebuah batang besi *A* sedemikan sehingga cakram yang berdiameter lebih besar selalu terletak di bawah cakram yang berdiameter lebih kecil (Gambar 1.5). Tulislah algoritma untuk memindahkan seluruh cakram tersebut batang besi B; setiap kali hanya satu cakram yang boleh dipindahkan, tetapi pada setiap perpindahan tidak boleh ada cakram yang lebih besar berada di atas cakram kecil. Batang besi *C* dapat dipakai sebagai tempat peralihan dengan tetap memegang aturan yang telah disebutkan.

5. Pada peristiwa pemilihan kepala desa (kades), setiap warga yang mempunyai hak pilih memilih satu di antara 4 calon kades. Kartu suara memuat foto dan nomor urut kades. Warga mencoblos calon kades yang dipilihnya, lalu memasukkan kartu suara ke dalam sebuah kotak. Setelah pemungutan suara usai, kegiatan selanjutnya adalah menghitung jumlah suara untuk masing-masing calon. Untuk menghitungnya, panitia tidak menggunakan tabel *cayley* seperti yang biasa dilakukan orang, tetapi menyediakan empat buah kotak kosong (yang merepresentasikan 4 calon kades). Satu per satu kartu suara diambil dan dibaca. Setiap kali kartu suara berisi coblosan nomor satu, maka sebutir batu kecil dimasukkan ke dalam kotak 1. Begitu pula setiap kali kartu suara berisi coblosan nomor dua, maka sebutir batu kecil dimasukkan ke dalam kotak 2. Hal yang sama juga dilakukan untuk kartu yang berisi coblosan nomor 3 dan empat. Demikian seterusnya sampai semua kartu suara habis dibaca. Akhirnya, jumlah batu di dalam setiap kotak menyatakan jumlah suara yang diraih oleh setiap calon kades. Tulislah algoritma untuk menghitung jumlah suara untuk masing-masing calon kades dengan metode perhitungan yang unik ini. Asumsikan bahwa semua suara adalah sah (tidak ada golput).

6. Di manakah letak kesalahan algoritma menjalankan sepeda motor berikut ini:

```
ALGORITMA menjalankan sepeda motor:
1. Hidupkan starter
2. Masukkan kunci kontak.
3. Tekan gigi 1.
4. Perbesar gas
5. Jalan.
```

2
Struktur Dasar Algoritma

Bab 2 ini akan menjelaskan konsep-konsep dasar algoritma, meliputi beberapa istilah yang sering dipakai dalam pemrograman dan struktur dasar pembangun algoritma. Untuk menjelaskan konsep dasar itu, kita mengambil contoh-contoh sederhana yang terdapat dalam kehidupan sehari yang mudah dimengerti.

2.1 Pernyataan

Sebuah algoritma merupakan deskripsi langkah-langkah pelaksanaan suatu proses. Setiap langkah di dalam algoritma dinyatakan dalam sebuah **pernyataan** (*statement*) atau istilah lainnya **instruksi**. Sebuah pernyataan berisi **aksi** (*action*) yang dilakukan. Bila sebuah pernyataan dieksekusi oleh pemroses, maka aksi yang bersesuaian dengan pernyataan itu dikerjakan.

Sebagai contoh, misalkan di dalam algoritma ada pernyataan berikut:

```
Tulis "Hello, world"
```

maka pernyataan tersebut menggambarkan aksi menulis pesan "`Hello, world`".

Pernyataan

```
Kalikan a dengan 2
```

menggambarkan aksi mengalikan `a` dengan `2` dan hasil perkalian disimpan di dalam peubah `a` lagi.

Pernyataan

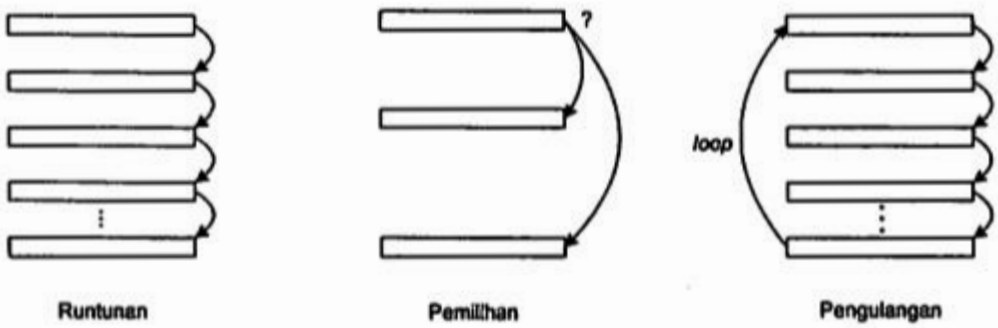
```
Jika bulan = 'januari' maka tulis "jumlah hari = 31"
```

Terdiri dari dua aksi, yaitu membandingkan nilai variabel `bulan` dengan `'januari'`, dan aksi tulis pesan `'jumlah hari = 31'` jika perbandingan itu benar.

Di dalam algoritma terdapat beberapa jenis pernyataan, seperti pernyataan ekspresi, pernyataan pemilihan, pernyataan pengulangan, pernyataan prosedur, pernyataan gabungan, dan sebagainya. Semua jenis pernyataan tersebut akan kita bahas nanti.

2.2 Konstruksi Dasar

Algoritma berisi langkah-langkah penyelesaian suatu masalah. Langkah-langkah tersebut dapat berupa runtunan aksi, pemilihan aksi, dan pengulangan aksi. Ketiga jenis langkah tersebut membentuk konstruksi suatu algoritma. Jadi, sebuah algoritma dapat dibangun dari tiga buah konstruksi atau struktur dasar, yaitu **runtunan** (*sequence*), **pemilihan** (*selection*), dan **pengulangan** (*repetition*) (Gambar 2.1).



Gambar 2.1 Tiga konstruksi dasar algoritma. Setiap persegi panjang melambangkan pernyataan. Anak panah menunjukkan aliran instruksi.

Gambaran ringkas masing-masing konstruksi dasar tersebut akan dijelaskan pada upa-bab di bawah ini. Pembahasan lengkap mengenai masing-masing konstruksi diberikan di dalam Bab 4, 5, dan 6.

2.2.1 Runtunan

Sebuah runtunan terdiri dari satu atau lebih pernyataan, setiap pernyataan ditulis dalam satu baris atau dipisahkan dengan tanda titik koma. Tiap pernyataan dikerjakan secara berurutan (sekuensial) sesuai dengan

urutannya di dalam teks algoritma, yaitu sebuah instruksi dilaksanakan setelah instruksi sebelumnya selesai dilaksanakan. Urutan instruksi menentukan keadaan akhir algoritma. Bila urutannya diubah, maka hasil akhirnya mungkin juga berubah. Runtunan satu atau lebih pernyataan dinamakan **pernyataan-gabungan** (*compound statements*).

Perhatikan runtunan lima buah pernyataan yang dilambangkan dengan `S1`, `S2`, `S3`, `S4`, dan `S5` berikut:

```
S1
S2
S3
S4
S5
```

Mula-mula pemroses melaksanakan pernyataan `S1`. Pernyataan `S2` dilaksanakan setelah pernyataan `S1` selesai. Selanjutnya, pernyataan `S3` dilaksanakan setelah pernyataan `S2` selesai. Demikian seterusnya sampai pernyataan terakhir `S5` dilaksanakan. Setelah pernyataan `S5` selesai dilaksanakan, algoritma berhenti.

Sebagai contoh pertama, tinjau kembali algoritma mempertukarkan isi dua buah ember *A* dan *B* di dalam Bab 1:

```
ALGORITMA memperturkarkan isi dua buah ember, A dan B:
1. Tuangkan air dari ember A ke dalam ember C.
2. Tuangkan air dari ember B ke dalam ember A.
3. Tuangkan air dari ember C ke dalam ember B.
```

Algoritma di atas adalah sebuah runtunan yang terdiri atas tiga buah pernyataan. Tiap pernyataan akan dieksekusi dalam urutan yang sama sebagaimana tertulis di dalam teks algoritma tersebut. Hasil akhir algoritma ini adalah: ember *A* berisi air dari ember *B* semula, dan ember *B* berisi air dari ember *A* semula.

Algoritma pertukaran isi dua buah ember memberikan inspirasi bagaimana mempertukarkan nilai dari dua buah peubah (variabel). Misalkan peubah *A* berisi nilai 8 dan peubah *B* berisi nilai 5. Kita ingin mempertukarkan nilai *A* dan *B* sedemikian sehingga *A* bernilai 5 dan *B* bernilai 8. Kalau Anda menulis algoritmanya seperti ini:

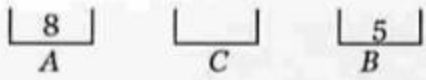
```
Masukkan nilai A ke dalam B
Masukkan nilai B ke dalam A
```

maka hasilnya $A = 8$ dan $B = 8$, tidak terjadi pertukaran karena algoritmanya salah. Agar pertukaran keduanya benar, maka kita memerlukan peubah bantu, misalnya *C*, sehingga algoritmanya menjadi:

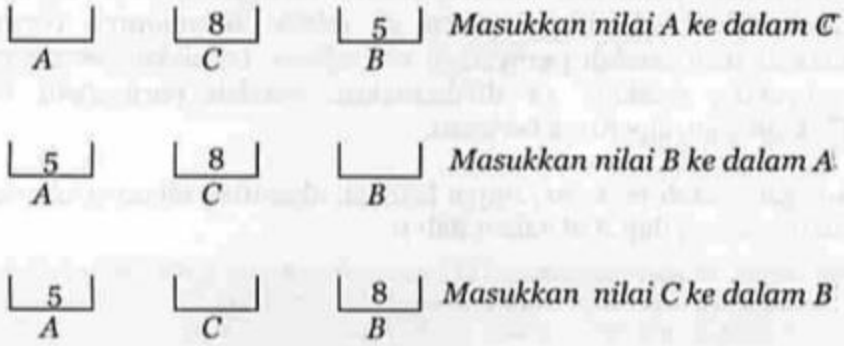
```
Masukkan nilai A ke dalam C
Masukkan nilai B ke dalam A
Masukkan nilai C ke dalam B
```

Ilustrasi ketiga pernyataan ini ditunjukkan pada Gambar 2.2.

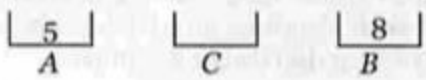
Sebelum pertukaran:



Proses pertukaran:



Setelah pertukaran:



Gambar 2.2 Proses mempertukarkan nilai A dan B dengan menggunakan peubah bantu C.

2.2.2 Pemilihan

Adakalanya sebuah instruksi dikerjakan jika kondisi tertentu dipenuhi. Misalkan kendaraan Anda tiba di perempatan yang ada *traffic light*. Jika lampu *traffic light* sekarang berwarna merah, maka kendaraan Anda harus berhenti. Langkah ini kita tulis dalam pernyataan berikut:

```
jika lampu traffic light berwana merah, maka berhenti
```

Pernyataan di atas dapat ditulis dalam pernyataan-pemilihan (*selection-statement*), atau disebut juga pernyataan-kondisional, sebagai berikut:

```
if kondisi then
   aksi
```

Dalam bahasa Indonesia, `if` berarti "jika" dan `then` artinya "maka"; *kondisi* adalah persyaratan yang dapat bernilai benar atau salah; *aksi* sesudah kata `then` hanya dilaksanakan apabila kondisi bernilai benar. Sebaliknya, apabila kondisi bernilai salah, maka aksi tidak dilaksanakan. Perhatikan bahwa kata yang digarisbawahi, `if` dan `then`, merupakan kata kunci (*keywords*) untuk struktur pemilihan ini.

Dalam kehidupan sehari-hari, kita sering menuliskan pelaksanaan aksi bila suatu persyaratan dipenuhi. Misalnya:

```
if air di dalam ketel mendidih then
   matikan api kompor

if suhu ruangan di atas 50 °C then
   bunyikan alarm tanda bahaya

if mobil rusak then
   naik angkot

if x habis dibagi 2 then
   tulis bahwa x bilangan genap
```

dan lain-lain sebagainya.

Struktur pemilihan *if-then* hanya memberikan satu pilihan aksi bila kondisi (persyaratan) dipenuhi (bernilai benar), dan tidak memberi pilihan aksi lain bila kondisi bernilai salah. Bentuk pemilihan yang lebih umum ialah memilih satu dari dua buah aksi bergantung pada nilai kondisinya:

```
if kondisi then
   aksi 1
else
   aksi 2
```

`else` artinya "kalau tidak". Bila kondisi terpenuhi, *aksi 1* akan dikerjakan, tetapi kalau tidak (yaitu kondisi salah), *aksi 2* yang akan dikerjakan. Misalnya pada pernyataan berikut:

```
if lampu A nyala then
   tekan tombol merah
else
   tekan tombol biru
```

Jika lampu *A* menyala, maka aksi tekan tombol merah dilakukan, sebaliknya, aksi tekan tombol biru dilakukan bila lampu *A* tidak menyala.

Contoh lainnya adalah menentukan nilai terbesar dari dua buah bilangan bulat, x dan y (andaikan $x \neq y$):

```
if x > y then
   tulis nilai x
else
   tulis nilai y
```

Menentukan apakah bilangan bulat *x* merupakan bilangan genap atau ganjil:

```
if x habis dibagi 2 then
   tulis "genap"
else
   tulis "ganjil"
```

Apabila pilihan aksi yang akan dilakukan lebih dari dua buah, maka struktur pemilihannya menjadi lebih rumit, seperti pada contoh berikut (pemilihan bersarang atau *nested-if*):

```
if lampu traffic light berwana merah then
   berhenti
else
   if lampu traffic light berwana kuning then
      jalan hati-hati
   else
      jalan terus
```

Perhatikanlah bahwa penggunaan indentasi (rongak kosong) membuat algoritma menjadi lebih mudah dibaca. Tanpa indentasi, algoritma mungkin menjadi sulit dibaca, misalnya jika algorima di atas ditulis seperti ini:

```
if lampu traffic light berwana merah then
hentikan kendaraan anda
else if lampu traffic light berwana kuning then
jalan dengan hati-hati else
jalan terus
```

Contoh lain tentang pentingnya penggunaan indentasi adalah pada pemilihan bersarang untuk menentukan bilangan terbesar dari tiga buah bilangan: *x*, *y*, dan *z*:

```
if x > y then
      if x > z then
         tulis x sebagai bilangan terbesar
      else
         tulis z sebagai bilangan terbesar
else
      if y > z then
         tulis y sebagai bilangan terbesar
      else
         tulis z sebagai bilangan terbesar
```

Bayangkan betapa sulitnya memahami algoritma di atas jika ia ditulis seperti di bawah ini:

```
if x > y then
if x > z then
tulis x sebagai bilangan terbesar
else tulis z sebagai bilangan terbesar
else if y > z then
tulis y sebagai bilangan terbesar
else tulis z sebagai bilangan terbesar
```

Tentu saja masalah menentukan bilangan terbesar untuk empat bilangan atau lebih mempunyai struktur pemilihan bersarang yang makin rumit. Pada bab tentang larik (*array*), kita akan belajar cara menentukan nilai terbesar dari sekumpulan nilai.

Kelebihan struktur pemilihan terletak pada kemampuannya yang memungkinkan pemroses mengikuti jalur aksi yang berbeda berdasarkan kondisi yang ada. Tanpa struktur pemilihan, kita tidak mungkin menulis algoritma untuk permasalahan praktis yang demikian kompleks.

2.2.3 Pengulangan

Salah satu kelebihan komputer adalah kemampuannya untuk mengerjakan pekerjaan yang sama berulang kali tanpa kenal lelah. Ini berbeda dengan manusia yang cepat lelah bila mengerjakan pekerjaan yang sama berulang-ulang. Tidak hanya lelah, tetapi juga cepat bosan.

Bagaimana menuliskan sebuah kalimat yang sama sebanyak 100 kali? Mungkin pada waktu duduk di sekolah dasar, Anda pernah dihukum oleh Bapak/Ibu guru dengan menuliskan sebuah kalimat yang sama sebanyak 100 kali, misalnya karena Anda nakal atau tidak membuat PR. Misalkan kalimat yang harus ditulis sebanyak 500 kali adalah seperti ini:

```
Saya berjanji tidak akan nakal dan malas lagi
```

Bagaimana jika kita menuliskan algoritmanya seperti ini:

```
ALGORITMA tulis kalimat 500 kali:
1. Tulis "Saya berjanji tidak akan nakal dan malas lagi"
2. Tulis "Saya berjanji tidak akan nakal dan malas lagi"
3. Tulis "Saya berjanji tidak akan nakal dan malas lagi"
.
.
99. Tulis "Saya berjanji tidak akan nakal dan malas lagi"
100. Tulis "Saya berjanji tidak akan nakal dan malas lagi"
```

Hmm, tentu saja algoritma di atas tidak elegan, karena instruksi

```
Tulis "Saya berjanji tidak akan nakal dan malas lagi"
```

harus ditulis dalam algoritma sebanyak 100 kali! Supaya kita tidak perlu menuliskan aksi yang sama berkali-kali, kita dapat menggunakan notasi pengulangan. Di dalam algoritma terdapat banyak notasi pengulangan yang bisa digunakan, antara lain *repeat N times*, *for*, *repeat-until*, dan *while*.

Dengan notasi pertama, *repeat N times* (yang artinya: ulangi sebanyak *N* kali), maka algoritmanya dapat kita tulis menjadi:

```
ALGORITMA tulis kalimat 100 kali:
   repeat 100 times
      Tulis "Saya berjanji tidak akan nakal dan malas lagi"
```

Struktur pengulangan ini dapat ditulis secara umum dengan pernyataan-pengulangan:

```
repeat N times
  aksi
```

yang artinya: *aksi* diulang dikerjakan sebanyak *N* kali.

Struktur pengulangan yang mirip dengan *repeat N times* adalah *for*:

```
for pencacah pengulangan dari 1 sampai N do
    aksi
```

yang artinya: *aksi* dilakukan sebanyak hitungan cacah pengulangan, yaitu dari 1 sampai *N* (yaitu sebanyak *N* kali). Pencacah pengulangan dapat di-*set* tidak hanya mulai dari 1, tetapi juga dari sembarang nilai yang lain. Contoh masalah penulisan 100 kalimat dengan notasi *for* menjadi:

```
ALGORITMA:
   for i dari 1 sampai 100 do
      Tulis "Saya berjanji tidak akan nakal dan malas lagi"
```

i adalah pencacah pengulangan yang mencacah pengulangan dari 1 sampai 100. Lebih lanjut mengenai notasi ini dapat Anda temukan di dalam Bab 7.

Struktur pengulangan yang ketiga adalah *repeat-until* (*repeat* artinya "ulangi" dan *until* artinya "sampai" atau "hingga") yang mempunyai bentuk umum sebagai berikut:

```
repeat
  aksi
until kondisi
```

yang artinya adalah pengulangan *aksi* dilakukan hingga kondisi (persyaratan) berhenti terpenuhi. Contoh masalah penulisan kalimat 100 kali dengan notasi *repeat-until* menjadi:

```
ALGORITMA tulis kalimat 100 kali:
  repeat
     Tulis "Saya berjanji tidak akan nakal dan malas lagi"
  until sudah 500 kali
```

Contoh yang bagus untuk menjelaskan penggunaan struktur *repeat-until* adalah pada masalah mencari data alamat dan nomor telepon mahasiswa tertentu, di mana data yang diketahui adalah *NIM* (Nomor Induk Mahasiswa) mahasiswa tersebut. Misalkan data pribadi mahasiswa disimpan di dalam sebuah tabel. Tabel terdiri atas kolom (*field*) *NIM*, *Nama*, *Alamat*, dan *Nomor Telepon* (lihat Tabel 2.1). Setiap baris di tabel kita sebut satu *entry*. Kita asumsikan tabel sudah berisi sejumlah data mahasiswa (tabel berisi minimal satu *entry* data).

Tabel 2.1
Tabel data mahasiswa

NIM	Nama	Alamat	Telepon
13599001	Ahmad Sadikin	Jl. Mawar 31 A	2504321
13599009	Rusli Nasution	Jl. Dago 231	2503456
...	...	...	...
13599087	Hamidah Tanjung	Jl. Pelesiran 24	2519038

Secara tradisional, kalau kita mencari informasi dari sebuah tabel, maka cara yang lazim kita lakukan adalah membaca setiap *entry* tabel satu per satu, mulai dari *entry* baris pertama, terus ke bawah, sampai data yang dicari ditemukan atau seluruh *entry* tabel sudah habis ditelusuri. Algoritma pencarian seperti itu sebagai berikut:

```
tinjau entry pertama di dalam tabel
if NIM pada entry tabel sama dengan NIM yang dicari then
   ambil Alamat dan Telepon dari NIM tersebut
else
   tinjau entry berikutnya di dalam tabel
   if NIM pada entry tabel sama dengan NIM yang dicari then
      ambil Alamat dan Telepon dari NIM tersebut
   else
      tinjau entry berikutnya di dalam tabel
      if NIM pada entry tabel sama dengan NIM yang dicari then
         ambil Alamat dan Telepon dari NIM tersebut
      else
         tinjau entry berikutnya di dalam tabel
         ...
```

Algoritma di atas mengandung kelemahan karena pemrogram *tidak tahu* kapan harus berhenti menuliskan pernyataan kondisional. Dengan kata lain, pemrogram tidak tahu berapa kali pernyataan

```
if NIM pada entry tabel sama dengan NIM yang dicari then
   ambil Alamat dan Telepon dari NIM tersebut
else
   tinjau entry berikutnya di dalam tabel
```

harus ditulis sampai data yang dicari ditemukan. Tidak seperti pada struktur *repeat N times* atau *for* yang jumlah pengulangan aksi sudah diketahui sebelum pengulangan dilaksanakan, maka kita dapat menggunakan struktur *repeat-until* jika jumlah pengulangan tidak diketahui di awal. Pengulangan aksi dilakukan sampai ditemukan *entry* dari *NIM* yang dicari atau akhir tabel sudah terlampaui. Algoritma pencarian data mahasiswa di dalam tabel dengan struktur *repeat-until* menjadi seperti di bawah ini:

```
ALGORITMA pencarian data di dalam tabel:
    baca NIM yang dicari {misalkan NIM = X }
    tinjau entry pertama tabel
    repeat
        if NIM pada entry tabel sama dengan X then
          tulis Alamat dan Telepon dari NIM X tersebut
        else
          tinjau entry berikutnya di dalam tabel
    until NIM yang dicari sudah ditemukan atau akhir tabel sudah terlampaui
```

Struktur pengulangan yang terakhir adalah *while* (*while* artinya "selagi" atau "selama"):

```
while kondisi do
  aksi
```

yang artinya adalah selama *kondisi* (persyaratan) pengulangan masih benar, maka *aksi* dikerjakan. Perbedaannya dengan *repeat-until*, jika pada *repeat-until* kondisi pengulangan dievaluasi di akhir, maka pada *while-do* kondisi pengulangan dievalusi di awal pengulangan.

Masalah pencarian data di dalam tabel dapat juga ditulis menggunakan struktur *while-do* ditunjukkan pada Algoritma 2.3 berikut:

```
ALGORITMA pencarian data di dalam tabel:
    baca NIM yang dicari {misalkan NIM = X }
    tinjau entry pertama tabel
    while NIM yang dicari belum ditemukan dan akhir tabel belum
          terlampaui do
        if NIM pada entry tabel sama dengan X then
```

```
    ambil Alamat dan Telepon dari NIM tersebut
  else
    tinjau entry berikutnya di dalam tabel
```

Kapan menggunakan *repeat N times*, *for*, *repeat-until*, dan *while*? Lebih lanjut mengenai struktur pengulangan dapat Anda pelajari di dalam Bab 7.

Soal Latihan Bab 2

1. Tulislah algoritma (dalam notasi kalimat deksriptif) untuk memperoleh informasi nomor telepon berdasarkan data alamat (nama jalan dan nomornya) kepada nomor penerangan lokal (108) PT Telkom. Algoritma harus menjelaskan proses bila (a) nomor 108 sibuk, (b) alamat yang diberikan penelepon belum mempunyai sambungan telepon.

2. Dua buah algoritma [GOL88] di bawah ini menjelaskan sikap yang harus dilakukan bila menemui lampu pengatur lalu lintas (*traffic light*) di jalan raya:

```
(1) if traffic light menyala then
      if lampu merah then
        berhenti
      else
        jalan

(2) if traffic light menyala then
      if lampu merah then
        berhenti
    else
      jalan
```

   Pada keadaan apa kedua algoritma di atas menggambarkan perilaku yang berbeda? Algoritma mana yang Anda anggap lebih memuaskan?

3. Sebuah tabel terdiri atas kolom *NIM*, *Nama*, *Alamat*, dan *Nomor Telepon*. Tabel tersebut sudah berisi data utama sekumpulan mahasiswa di sebuah Perguruan Tinggi. Misalkan seorang mahasiswa pindah kos sehingga alamat dan nomor teleponnya harus diubah. Tuliskan algoritma (dalam notasi kalimat deskriptif) untuk mengubah kolom *Alamat* dan *Nomor Telepon* dari mahasiswa dengan *NIM* tertentu.

4. Algoritma di bawah ini membagi sekantung permen secara adil kepada 3 orang anak dengan cara memberikan satu permen kepada tiap anak secara berulang-ulang:

```
repeat
   berikan satu permen kepada anak pertama
   berikan satu permen kepada anak kedua
   berikan satu permen kepada anak ketiga
until kantung permen kosong
```

Pada keadaan bagaimana algoritma tersebut gagal?

3
Notasi Algoritmik

Algoritma berisi deskripsi langkah-langkah penyelesaian masalah. Langkah-langkah penyelesaian tersebut dapat kita tuliskan dalam notasi algoritmik sembarang, asalkan ia mudah dibaca dan dipahami. Tidak ada notasi yang standar untuk menuliskan algoritma sebagaimana pada notasi bahasa pemrograman. Setiap orang boleh saja mendefinisikan notasi algoritmiknya sendiri. Hal ini bukan persoalan karena notasi algoritma tidak sama dengan kode program komputer. Program komputer adalah implementasi algoritma dalam notasi bahasa pemrograman tertentu. Bab 3 ini akan membahas mengenai notasi algoritmik yang digunakan di dalam buku ini.

3.1 *Pseudo-code*

Notasi algoritmik yang baik adalah notasi yang mudah dibaca dan mudah pula ditranslasikan ke dalam notasi bahasa pemrograman. Notasi algoritmik berupa *pseudo-code* mempunyai korespondensi dengan notasi bahasa pemrograman sehingga proses penerjemahan dari *pseudo-code* ke kode program menjadi lebih mudah.

Tidak ada aturan baku membuat *pseudo-code*. Tidak seperti bahasa pemrograman yang direpotkan dengan tanda titik koma (*semicolon*), indeks, format keluaran, kata-kata khusus, dan sebagainya, sembarang versi *pseudo-code* dapat diterima asalkan notasinya bisa dipahami.

(1) Sebuah pernyataan dalam notasi deksriptif:

```
tulis nilai X dan Y
```

maka *pseudo-code*-nya dalam notasi algoritmik mungkin ditulis sebagai:

```
write(X,Y)
```

Notasi `write` berarti nilai `X` dicetak ke piranti keluaran. Dengan notasi algoritmik, kita tidak terlalu mempersoalkan format tampilan keluaran, misalnya apakah hasil penulisan antara `X` dan `Y` dipisah dengan spasi atau dengan koma, apakah `X` dan `Y` dicetak di dalam baris yang sama atau tidak, apakah `X` dan `Y` dicetak pada koordinat tertentu di layar, dan sebagainya. Bila `X` dan `Y` berisi nilai bertipe bilangan riil (memakai titik desimal) kita juga tidak mempersoalkan berapa angka di belakang koma, dan sebagainya. Hal-hal teknis semacam ini barulah kita pikirkan pada saat translasi notasi algoritmik tersebut ke notasi bahasa pemrograman. Dengan cara seperti ini, kita telah membuat notasi algoritmik terlepas dari hal-hal teknis seperti yang terdapat pada bahasa pemrograman.

Notasi algoritmik dalam bentuk *pseudo-code* sebaiknya mudah ditranslasikan ke dalam notasi bahasa pemrograman pada saat *coding*. Notasi `write` di dalam algoritma berkoresponden dengan `write` atau `writeln` dalam bahasa *Pascal*, `printf` dalam bahasa *C*, `WRITE` dalam bahasa *Basic*, atau `write` dalam bahasa *Fortran*. Selain itu, pada beberapa bahasa pemrograman seperti *Pascal* dan *C*, antara setiap instruksi dipisahkan dengan tanda ";" (*semicolon*). Jadi, translasi `write(X)` ke dalam masing-masing bahasa tersebut adalah (dengan asumsi bahwa piranti keluarannya adalah layar) sebagai berikut:

Algoritmik:

```
write(X,Y)
```

Bahasa Pascal:

```
write(X,Y);
```

Bahasa C:

```
printf("%d %d", X,Y);
```

Bahasa Basic:

```
WRITE X,Y
```

Bahasa Fortran:

```
WRITE(*,*) X,Y
```

Perhatikan bahwa setiap bahasa mempunyai cara yang berbeda dalam menulis perintah penulisan. Bahasa *C* misalnya, mengharuskan pemakaian penentu-format (seperti "%d" pada contoh di atas) untuk nilai yang dicetak. Penentu format "%d" berarti pada *field* tersebut nilai yang akan dicetak bertipe bilangan bulat).

(2) Sebuah pernyataan dalam notasi dekriptif:

```
isikan nilai X ke dalam min
```

maka *pseudo-code*-nya dalam notasi algoritmik menjadi:

```
min ← X
```

Notasi "←" berarti mengisi (*assign*) peubah (*variable*) `min` dengan nilai `X`. Translasi notasi "←" dalam bahasa *Pascal* adalah "`:=`", dalam bahasa *C* adalah "`=`", dalam bahasa *Basic* adalah "`=`" dan dalam bahasa *Fortran* juga "`=`". Jadi, translasi `min←X` ke dalam masing-masing bahasa tersebut adalah:

Algoritmik:

```
min ← X
```

Bahasa Pascal:

```
min := X;
```

Bahasa C:

```
min = X;
```

Bahasa Basic:

```
min = X
```

Bahasa Fortran:

```
min = X
```

3.2 Struktur Teks Algoritma

Dua buah algoritma sederhana diberikan untuk mengilustrasikan teks algoritma seperti apa yang dipakai di dalam buku ini.

(1) Algoritma pertama adalah untuk mencetak tulisan:

```
Hello, world
```

Algoritma *Hello world* hampir selalu ditemukan pada buku-buku bahasa pemrograman. Pelajaran pemrograman dan buku-buku bahasa pemrograman sering memulai contoh program pertamanya dengan memberikan contoh bagaimana mencetak "*Hello, world*" ke layar. Program "*Hello, world*" pertama kali dikemukakan oleh Brian W. Kernighan, yaitu seorang penemu Bahasa *C* [KER88].

Algoritma untuk menulis "*Hello, world*":

```
PROGRAM HelloWorld
{ Program untuk mencetak "Hello world"
  Masukan: -
  Keluaran: string 'Hello, world'
}

DEKLARASI
   { tidak ada }

ALGORITMA:
   write("Hello, world")
```

Algoritma 3.1 Mencetak "Hello, world"

(2) Algoritma kedua adalah program untuk mencetak tabel konversi suhu dalam derajat Fahrenheit (F) dan suhu ekivalennya dalam derajat Celcius (C) dengan menggunakan rumus

```
C = 5/9 (F - 32)
```

Data masukan adalah suhu awal (x) dan suhu akhir (y). Tabel keluaran memuat suhu mulai dari x sampai y °F dengan kenaikan sebesar *step*. Misalnya jika $x = 0$, $y = 100$, dan *step* = 20, tabel yang ingin dihasilkan kira-kira seperti di bawah ini:

0	−17.8
20	− 6.7
40	4.4
60	15.6
80	26.7
100	37.8

Maka, algoritma untuk mencetak tabel konversi Fahrenheit-Celcius sebagai berikut:

```
PROGRAM FahrenheitCelcius
{ Program untuk mencetak tabel Fahrenheit-Celcius dari x sampai y
  dengan kenaikan seebsar step.
  Masukan: suhu awal, suhu akhir, step
  Keluaran: tabel konversi suhu dalam C dan F
}

DEKLARASI
   F, C : real
   x, y, step : integer

ALGORITMA:
   read(x, y, step)
   F ← x
   while F ≤ y do
      C = 5/9 * (F - 32)
      write(F, C)
```

```
    F ← F + step
  endwhile
```

Algoritma 3.2 Mencetak tabel konversi Fahrenheit-Celcius

Dengan memperhatikan kedua contoh algoritma sederhana di atas, maka teks algoritma selalu disusun oleh tiga bagian (blok), yaitu:

1. bagian judul (*header*),
2. bagian deklarasi (*declaration*),
3. bagian algoritma.

Ketiga bagian tersebut secara umum ditunjukkan dalam notasi algoritmik pada Algoritma 3.3.

```
PROGRAM Nama Program
{ Penjelasan tentang algoritma, yang berisi uraian singkat mengenai
masalah apa yang akan diselesaikan, termasuk masukan dan keluarannya }

DEKLARASI
{ semua nama yang dipakai, meliputi nama tipe, nama konstanta, nama
peubah, nama prosedur dan nama fungsi diumumkan di sini }

ALGORITMA:
{ berisi langkah-langkah penyelesaian masalah}
```

Algoritma 3.3 Struktur teks algoritma

Pada setiap bagian sangat dianjurkan menuliskan komentar untuk memperjelas maksud dari pernyataan. Komentar adalah kalimat yang diapit oleh pasangan tanda kurung kurawal ('{' dan '}'). Komentar membuat algoritma lebih mudah untuk dibaca dan dipahami. Di dalam program komputer, komentar tidak dieksekusi, hanya dilewati saja.

Penjelasan masing-masing bagian di dalam teks algoritma diberikan pada upa-bab di bawah ini.

3.2.1 Bagian Judul

Judul adalah bagian yang terdiri atas nama program dan penjelasan (spesifikasi) tentang program tersebut. Judul diawali dengan kata kunci PROGRAM dan nama program X. Kata PROGRAM ini bukan menyatakan program dalam bahasa komputer, tetapi menyatakan bahwa kita sedang menulis algoritma untuk program pemecahan masalah.

Nama program sebaiknya singkat namun cukup menggambarkan apa yang dilakukan oleh program. Di bawah nama program sebaikya disertai dengan penjelasan singkat tentang apa masalah diprogram dan apa masukan dan

keluarannya. Dua contoh program terdahulu mempunyai judul sebagai berikut:

```
PROGRAM HelloWorld
{ Program untuk mencetak "Hello, world"
Masukan program ini tidak ada. Keluarannya adalah tulisan 'Hello,
world' tercetak di layar.
}
```

```
PROGRAM FahrenheitCelcius
{ Program untuk mencetak tabel Fahrenheit-Celcius dari x sampai y
dengan kenaikan sebesar step. Masukan program ini adalh suhu awal,
suhu akhir, step, dan keluarannya adalah tabel konversi suhu dalam C
dan F
}
```

3.2,2 Bagian Deklarasi

Bagian ini digunakan untuk mengumumkan semua nama yang dipakai di dalam algoritma beserta propertinya (misalnya tipe). Nama tersebut dapat berupa nama konstanta, nama peubah, nama tipe, nama prosedur dan nama fungsi. Semua nama yang dipakai di dalam algoritma harus dikenali sebelum mereka digunakan. Bagian deklarasi mungkin kosong jika tidak ada penggunaan nama. Program `HelloWorld` mengandung bagian deklarasi yang kosong, sedangkan program `FahrenheitCelcius` berisi deklarasi nama-nama peubah yang digunakan di dalam program beserta tipenya sebagai berikut:

```
DEKLARASI
   F, C : real
   x, y, step : integer
```

`F`, `C`, `x`, `y`, dan `step` adalah nama-nama peubah yang digunakan di dalam bagian algoritma. Nama-nama peubah ini beserta tipenya harus diumumkan di bagian deklarasi sebelum mereka digunakan. Penjelasan lebih lanjut mengenai nama dan tipe akan diberikan di dalam Bab 4.

Contoh-contoh deklarasi yang lain (deklarasi nama konstanta, nama tipe, nama peubah, nama prosedur dan fungsi) adalah seperti di bawah ini:

```
DEKLARASI
   { nama konstanta }
   const NPeg = 100    { jumlah pegawai }
   const phi = 3.14    { nilai π }
   { nama tipe }
   type Titik : record      { koordinat titik di bidang kartesian }
                < x:integer,    { absis }
                  y:integer >   { ordinat }
   { nama peubah (variable) }
```

```
c     : char                    { karakter yang dibaca}
Q     : Titik                   { titik dalam koordinat kartesian}
ketemu : boolean                { keadaan hasil pencarian}

FUNCTION Apakah_A(input c : char) → boolean
{ Mengembalikan nilai true bila c adalah karakter 'A', atau
  false bila sebaliknya }


PROCEDURE Tukar(input/output A : integer, input/output B : integer)
{ Mempertukarkan nilai A dan B. Parameter A dan B sudah terdefinisi
  nilainya. Setelah pertukaran, A berisi nilai B semula, B berisi
  nilai A semula.}
```

Npeg dan phi adalah nama konstanta, Titik adalah nama tipe, c, q, dan ketemu adalah nama peubah, Apakah_A adalah nama fungsi, dan Tukar adalah nama prosedur.

Oleh karena nama adalah satu kesatuan leksikal, maka nama tidak boleh mengandung spasi, tanda baca (seperti titik, koma, dan sebagainya), operator (seperti +, -, :, dan sebagainya). Karakter garis bawah ("_") dapat digunakan untuk mengganti spasi. Jadi, menuliskan nama fungsi sebagai

```
Apakah_A
```

adalah benar, tetapi

```
Apakah A
```

salah karena mengandung spasi.

Selain spasi, kita juga tidak membedakan huruf besar (kapital) dan huruf kecil di dalam notasi algoritmik kita. Namun, pada bahasa pemrograman yang *case sensitive* seperti bahasa C, huruf besar dan huruf kecil adalah dua karakter yang dianggap berbeda.

3.2.3 Bagian Algoritma

Inilah bagian inti dari sebuah program. Bagian ini berisi instruksi-instruksi pemecahan masalah dalam notasi *pseudo-code*.

Program HelloWorld mengandung bagian algoritma yang hanya berisi satu baris instruksi saja:

```
ALGORITMA:
  write("Hello, world")
```

Program FahrenheitCelcius mengandung bagian algoritma yang berisi beberapa baris instruksi:

```
ALGORITMA:
   read(x, y, step)
   F ← x
   while F ≤ y do
      C = 5/9 * (F - 32)
      write(F, C)
      F ← F + step
   endwhile
```

Untuk sementara ini, Anda tidak perlu memikirkan terlalu serius *pseudo-code* seperti `write`, `while`, dan lambang "←". Anda akan mempelajari *pseudo-code* lebih mendalam di dalam bab-bab selanjutnya.

3.3 Translasi Notasi Algoritmik ke Bahasa Pascal dan C

Teks algoritma merupakan desain penyelesaian masalah. Agar dapat dieksekusi oleh komputer, algoritma harus diterjemahkan terlebih dahulu ke dalam notasi bahasa pemrograman. Penerjemahan dari notasi algoritmik ke notasi bahasa pemrograman dinamakan **translasi**.

Di dalam buku ini, notasi algoritma hanya akan ditranslasi ke dalam bahasa *Pascal* dan bahasa *C* saja. Bahasa *Pascal* dipilih karena ia relatif mudah dibaca, notasinya mirip dengan bahasa Inggris standard. Bahasa *C* dipilih karena bahasa ini *powerfull*, notasinya ringkas, dan dipakai secara luas di dalam industri perangkat lunak.

Buku ini tidak bertujuan mengajarkan Anda bahasa *Pascal* dan *C*, tetapi mengajarkan cara menerjemahkan notasi algoritmik ke dalam bahasa *Pascal* dan *C*. Jika Anda belum pernah belajar bahasa *Pascal* atau *C*, sebaiknya Anda perlu membaca buku-buku tentang kedua bahasa tersebut.

ALGORITMIK:

```
PROGRAM Nama Program
{ Penjelasan tentang algoritma, yang berisi uraian singkat mengenai
masalah apa yang akan diselesaikan, apa masukan dan keluarannya }

DEKLARASI
{ semua nama yang dipakai, meliputi nama tipe, nama konstanta, nama
peubah, nama prosedur dan nama fungsi diumumkan di sini }

ALGORITMA:
{ berisi langkah-langkah penyelesaian masalah}
```

PASCAL:

```
program Nama Program;
{ Penjelasan tentang algoritma, yang berisi uraian singkat mengenai
masalah apa yang akan diselesaikan, apa msukan dan keluarannya }

(* DEKLARASI *)
[const]
{ semua nama tetapan dan harga tetapannya didefinisikan di sini }
[type]
{ semua nama tipe bentukan didefenisikan di sini}
[var]
{ semua nama peubah global didefinisikan di sini }

{ deklarasi prosedur dan fungsi ditulis di sini }

(* ALGORITMA: *)
begin
   { semua instruksi program dituliskan di sini}
end.
```

C:

```
/* PROGRAM Nama Program */
/* Penjelasan tentang algoritma, yang berisi uraian singkat mengenai
masalah apa yang akan diselesaikan, apa msukan dan keluarannya */

/* DEKLARASI */
/* semua nama yang penggunaannya global didefinisikan di sini    */

/* purwarupa prosedur dan fungsi didefinisikan di sini    */

/* ALGORITMA: */
main()
{
  /* di sini juga mungkin ada DEKLARASI lokal */

  /* semua instruksi program dituliskan di sini */

}
```

Beberapa hal penting yang harus diketahui dari bahasa *C* adalah:

1. Bahasa *C* bersifat *case sensitive*. Ini berarti bahasa *C* membedakan huruf besar (kapital) dengan huruf kecil. Nama yang ditulis dengan huruf besar dianggap berbeda kalau ia ditulis dengan huruf kecil. Misalnya,

   `N` tidak sama dengan `n`
   `nama_orang` tidak sama dengan `NAMA_ORANG`, `Nama_orang`, `Nama_Orang`
   `HitKar` tidak sama dengan `hitkar`

   Hal ini berbeda dengan bahasa *Pascal*. *Pascal* tidak membedakan nama yang ditulis dalam huruf besar atau huruf kecil. Jadi,

`N` sama saja dengan `n`
`nama_orang` sama saja dengan `NAMA_ORANG` atau `Nama_ORANG`
`HitKar` sama saja dengan `hitkar`

2. Dalam bahasa *C*, deklarasi yang didefinisikan sebelum kata `main()` adalah deklarasi global, artinya semua nama yang didefinisikan di dalam deklarasi global dikenal di seluruh bagian program, termasuk di dalam fungsi atau prosedur yang ada di dalam program itu. Apabila deklarasi didefinisikan di dalam `main()`, maka nama di dalam bagian deklarasi (disebut deklarasi lokal) hanya dikenal oleh program utama saja, tetapi tidak dikenal oleh fungsi atau prosedur.

3. Dalam bahasa *C*, komentar ditulis di antara "/*" dan "*/". Sedangkan dalam bahasa *Pascal* komentar dapat ditulis di antara "{" dan "}" atau di antara "(*" dan "*)".

Untuk mengilustrasikan cara mentranslasi, berikut ini kita berikan contoh translasi program `HelloWorld` dan program `FahrenheitCelcius` dalam notasi *Pascal* dan *C*. Kita sengaja tidak menghilangkan kata "DEKLARASI" dan kata "ALGORITMA" di dalam kode program *Pascal* dan *C*, agar pembaca tidak kehilangan 'jejak' dalam proses translasi. Kita menuliskan kata "DEKLARASI" dan kata "ALGORITMA" sebagai komentar saja. Anda boleh membuangnya jika itu dianggap merepotkan. Cara-cara mentranslasi setiap notasi algoritma akan dijelaskan pada setiap bab yang membahas notasi tersebut.

ALGORITMIK:

1. Program mencetak *Hello World*

```
PROGRAM HelloWorld
{ Program untuk mencetak "Hello world". Masukan program ini tidak ada.
Keluarannya adalah tulisan 'Hello, world' tercetak di layar. }

DEKLARASI
   { tidak ada }

ALGORITMA:
   write("Hello, world")
```

2. Program konversi suhu Fahrenheit-Celcius

```
PROGRAM FahrenheitCelcius
{ Program untuk mencetak tabel Fahrenheit-Celcius dari x sampai y
dengan kenaikan sebesar step. Masukan program ini adalh suhu awal,
suhu akhir, step, dan keluarannya adalah tabel konversi suhu dalam C
dan F
}
```

```
DEKLARASI
   F, C : real
   x, y, step : integer

ALGORITMA:
   read(x, y, step)
   F ← x
   while F ≤ y do
      C = 5/9 * (F - 32)
      write(F, C)
      F ← F + step
   endwhile
```

PASCAL:

1. Program mencetak *Hello World*

```
PROGRAM HelloWorld;
{ Program untuk mencetak "Hello world". Masukan program ini tidak ada.
Keluarannya adalah tulisan 'Hello, world' tercetak di layar.
}

(* DEKLARASI *)
   { tidak ada }

begin
(* ALGORITMA: *)
   write('Hello, world');
end.
```

2. Program konversi suhu Fahrenheit-Celcius

```
PROGRAM FahrenheitCelcius;
{ Program untuk mencetak tabel Fahrenheit-Celcius dari x sampai y
dengan kenaikan sebesar step. Masukan program ini adalah suhu awal,
suhu akhir, step, dan keluarannya adalah tabel konversi suhu dalam C
dan F }

(* DEKLARASI *)
var
   F, C : real;
   x, y, step : integer;

begin
(* ALGORITMA: *)
   read(x);
   read(y);
   read(step);

   F := x;
   while F <= y do
    begin
      C:= 5/9 * (F - 32);
      writeln(F, '  ', C);
      F := F + step;
```

```
    end;
end.
```

C:

```
/* PROGRAM HelloWorld */
/* Program untuk mencetak "Hello world". Masukan program ini tidak
ada. Keluarannya adalah tulisan 'Hello, world' tercetak di layar.*/

#include <stdio.h>
main()
{
 /* DEKLARASI */
     /* tidak ada */

 /* ALGORITMA: */
   printf("Hello, world");
}
```

```
/* PROGRAM FahrenheitCelcius */
/* Program untuk mencetak tabel Fahrenheit-Celcius dari x sampai y
dengan kenaikan sebesar step. Masukan program ini adalah suhu awal,
suhu akhir, step, dan keluarannya adalah tabel konversi suhu dalam C
dan F */

#include <stdio.h>

main()
{
 /* DEKLARASI */

   float F, C ;
   int x, y, step;



/* ALGORITMA: */
   scanf("%d", &x);
   scanf("%d", &y);
   scanf("%d", &step);

   F = x;
   while (F <= y)
    {
      C = (5.0/9.0) * (F - 32);
      printf("%3.0f  %6.1f \n, F, C);
      F = F + step;
    }
}
```

Keterangan: Pernyataan

```
#include <stdio.h>
```

adalah pernyataan untuk melibatkan (*include*) pustaka standar yang berisi operasi masukan/keluaran (operasi baca dan operasi tulis). Pustaka masukan/keluaran ini terdapat di dalam *file header* `stdio.h`. Di dalam Bahasa *C* terdapat sejumlah *file header* yang mungkin diperlukan jika program menggunakan beberapa fungsi standar (salah satu fungsi standar di dalam program di atas adalah printf).

3.4 Kompilator Bahasa Pascal dan C

Lihatlah kembali Gambar 1.8 di dalam Bab 1 mengenai tahapan pelaksanan program oleh komputer. Program sumber (*source program*) dalam bahasa pemrograman harus dikompilasi terlebih dahulu oleh program khusus yang disebut kompilator (*compiler*). Kompilasi program bertujuan memeriksa kebenaran sintaks (tata bahasa) program, kemudian, jika tidak ada kesalahan, program diterjemahkan ke dalam bahasa mesin sehingga siap dieksekusi.

Saat ini terdapat cukup banyak kompilator bahasa *Pascal* dan bahasa *C*. Pada era komputer 16-bit, kompilator bahasa *Pascal* dan bahasa *C* yang populer adalah *Turbo Pascal* dan *Turbo C*. Sekarang kedua kompilator ini relatif sudah jarang digunakan sejak digunakannya sistem operasi *Windows* dan *UNIX* (termasuk *Linux*).

Saat ini, kakas (*tool*) pengembangan program berbasis *GUI* (*Graphical user Interface*) yang populer digunakan adalah *Borland Delphi*, *Visual C*, *Borland C++*, dan sebagainya. *Borland Delphi* adalah kakas pengembangan program yang berbasis pada bahasa *Pascal Object*, sedangkan *Visual C* dan berbasis pada bahasa *C*. Kedua kompilator ini masih dapat digunakan untuk mengompilasi program *Pascal* dan *C*.

Sejak diberlakukannya keharusan menggunakan *software* legal (sebagai konsekuensi diberlakukannya UU No 19/2002 tentang HaKI (Hak Kekayaan Intelektual), maka kita harus mengeluarkan biaya yang cukup besar untuk membeli kakas pengembangan program yang berlisensi. Namun, bukan berarti tidak ada cara yang murah untuk tetap bisa memprogram. Di internet terdapat beberapa kompilator bahasa *Pascal* dan bahasa *C* yang dapat diunduh (*download*) secara gratis, misalnya *Free Pascal* (untuk bahasa *Pascal*) dan *GCC* untuk (untuk bahasa *C*).

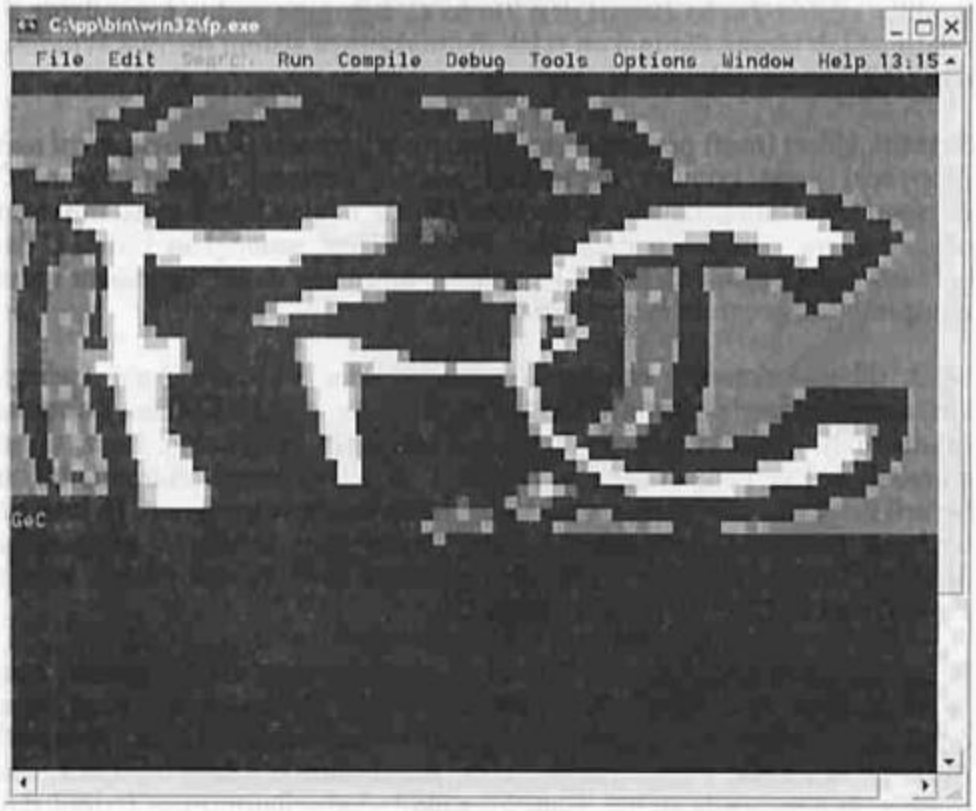
Free Pascal adalah kompilator *Pascal* 32-bit. Ia tersedia untuk prosesor *Intel x86*, *Motorola 680x0*, dan *PowerPC* (dari 1.9.2). Sistem operasi yang dapat digunakan untuk mengoperasikan *Free Pascal* adalah *Linux*, *FreeBSD*, *NetBSD*, *MacOSX*, *DOS*, *Win32*, *OS/2*, *BeOS*, *SunOS* (*Solaris*), *QNX*, dan *Classic Amiga*. Anda dapat men-*download* kompilator *Free Pascal* dari

www.freepascal.org. Tersedia banyak pilihan kompilator *Free Pascal* yang dapat Anda *download* (baik versi standar maupun versi yang lengkap). Instalasilah kompilator *Pascal* ini ke komputer Anda. Program *Pascal* dapat dikompilasi dari *prompt DOS* (*Disk Operating System*) atau dari *IDE* (*Integrated Development Environment)* yang disediakan oleh *Free Pascal* ini.

GCC (*GNU C compiler*) adalah kompilator *freeware* untuk bahasa *C*. Kompilator ini dapat disertakan bersamaan pada instalasi sistem operasi *Linux*. Kita dapat men-*download GCC* dari internet untuk di-instalasi pada sistem operasi selain *Linux* (seperti *Window* atau *DOS* misalnya). Informasi mengenai kompilator *GCC* dapat dilihat di http://gcc.gnu.org. Anda juga dapat men-*download GCC* dari situs *web* tersebut, lalu menginstalasinya di komputer Anda

(a) Mengompilasi Program dengan *Free Pascal*

Program dalam bahasa *Pascal* terlebih dahulu disunting dengan editor teks. *Free Pascal* sendiri menyediakan editor yang menyatu dengan *IDE* (*Integrated Development Environment*). *IDE* ini mirip dengan *IDE* pada kompilator *Turbo Pascal 7* ke atas (lihat Gambar 3.1).



Gambar 3.1 IDE pada *Free* Pascal

Suntinglah program Pascal *Hello, world*, lalu simpan kode programnya ke dalam berkas dengan nama `Hello.pas`. Kompilasi program tersebut dari *IDE*. Jika tidak ada kesalahan sintaks di dalam program, maka kompilasi program berlangsung sukses. *Run* program dari *IDE*, maka hasil *running* program akan tampak di layar seperti ini:

```
Hello, world
```

Selain dari *IDE*, program *Pascal* juga dapat dikompilasi dari *prompt DOS*. Pindahkan direktori Anda ke `C:/pp/bin/win32`, lalu kompilasi program `Hello.pas` yang disimpan di direktori ini dengan mengetikkan perintah berikut dari *prompt*:

```
ppc386 Hello.pas
```

Jika tidak ada kesalahan sintaks di dalam program, maka kompilasi program berlangsung sukses, dan kompilator menghasilkan *executable file* yang berupa berkas biner bernama `Hello.exe`. Run *executable file* ini dengan mengetikkan

```
Hello
```

dari *prompt*, maka *running* program akan memberikan hasil yang sama seperti dari *IDE* di atas.

Selain di lingkungan *DOS* atau *Windows*, *Free Pascal* juga dapat dioperasikan di lingkungan *Linux*. Cara mengompilasi program Pascal di *Linux* sama seperti mengompilasi dari *prompt DOS* di atas (`ppc386` *nama_file.pas*).

(b) Mengompilasi Program dengan *GCC*

GCC tidak menyediakan editor teks seperti halnya *FreePascal*, tetapi kita dapat menggunakan editor teks biasa seperti *Notepad*, *Ultra Edit*, atau editor yang tersedia pada sistem operasi *Linux* seperti *vi*, *joe*, dan sebagainya. Pelajari cara menggunakan editor-editor ini. Program sumber disimpan di dalam berkas dengan ekstensi ".c". Misalnya program C *Hello, world* disimpan dengan nama `Hello.c`. Kompilasi berkas `Hello.c` ini dengan mengetikkan *command line*:

```
gcc Hello.c
```

dari *prompt* (*prompt* Linux atau *prompt DOS*) Jika tidak ada kesalahan sintaks di dalam program, maka kompilasi program berlangsung sukses, dan kompilator menghasilkan *executable file* yang berupa berkas biner bernama `a.out`. dan Anda mengetikkan perintah

```
a.out
```

dari *prompt*, maka hasil *running* program akan tampak di layar seperti ini:

```
Hello, world
```

Catatan:
Jika nama executable file yang kita inginkan sama dengan nama berkas program, maka kompilasi program `Hello.c` *sebagai berikut:*

```
gcc Hello.c -o Hello.out
```

4

Tipe, Operator, dan Ekspresi

Program komputer pada dasarnya memanipulasi objek (data) di dalam memori. Peubah (*variable*) dan konstanta (*constant*) adalah objek data dasar yang dimanipulasi di dalam program. Deklarasi daftar peubah digunakan untuk menyatakan tipe peubah. Operator menspesifikasikan operasi apa yang dapat dilakukan terhadap peubah dan konstanta. Ekspresi mengombinasikan peubah-peubah dan konstanta untuk menghasilkan nilai baru.

Tipe sebuah objek menentukan himpunan nilai yang dapat dimilikinya dan operasi yang dapat dilakukan pada objek tersebut. Nilai-nilai yang dicakup oleh tipe tersebut dinyatakan di dalam ranah (*domain*) nilai. Operasi-operasi (beserta operator) yang dapat dilakukan terhadap tipe tersebut juga didefinisikan.

Tipe data dapat dikelompokkan menjadi atas dua macam: **tipe dasar** dan **tipe bentukan**. Tipe dasar adalah tipe yang dapat langsung dipakai, sedangkan tipe bentukan dibentuk dari tipe dasar atau dari tipe bentukan lain yang sudah didefinisikan sebelumnya.

4.1 Tipe Dasar

Tipe dasar sudah dikenal dalam kehidupan sehari-hari. Tipe ini sudah ada sejak zaman dahulu (*predefined data type*). Kita hampir setiap hari berbicara tentang angka-angka dan karakter. Dalam dunia pemrograman, yang termasuk ke dalam tipe dasar adalah: bilangan lojik, bilangan bulat, karakter, bilangan riil, dan *string*. Tiga tipe dasar yang pertama disebut juga

tipe ordinal karena setiap konstanta nilainya dapat ditransformasi ke suatu nilai *integer*.

4.1.1 Bilangan Lojik

Nama Tipe
Nama tipe bilangan lojik adalah *boolean* (diambil dari nama seorang matematikawan Inggris, George Boole).

Ranah Nilai
Bilangan lojik hanya mengenal dua buah nilai: benar (*true*) atau salah (*false*). Istilah "bilangan" pada "bilangan lojik" dikarenakan kita dapat menyatakan "benar" dengan angka 1 dan "salah" dengan angka 0 (atau sebaliknya, bergantung konvensi yang kita buat).

Konstanta
Karena ranah nilai tipe *boolean* hanya beranggotakan dua buah nilai, maka konstanta (*constant*) yang terdapat pada tipe ini adalah `true` dan `false`

Operasi
Operasi-operasi yang dapat dilakukan terhadap tipe *boolean* dikenal dengan **operasi logika** atau **operasi boolean**. Operasi logika menghasilkan nilai *true* atau *false*. Operator logika yang umum digunakan untuk operasi logika adalah: *not*, *and*, *or*, dan *xor*.

Jika *a* dan *b* adalah peubah (*variable*) yang bertipe *boolean*, maka hasil operasi *a* dan *b* dengan keempat operator *boolean* tersebut diberikan oleh masing-masing tabel (yang disebut tabel kebenaran – *truth table*) berikut:

a	not a
true	false
false	true

a	b	a and b	a or b	a xor b
true	true	true	true	false
true	false	false	true	true
false	true	false	true	true
false	false	false	false	false

Cara mengingat hasil operasi dengan operator *boolean* sangat mudah. Ingatlah bahwa operasi dengan operator and hanya akan bernilai benar bila *a* dan *b* keduanya bernilai benar. Operasi dengan operator or hanya akan bernilai salah bila *a* dan *b* keduanya bernilai salah. Sedangkan operasi

dengan operator xor akan bernilai benar bila *a* dan *b* saling berlawanan nilai kebenarannya.

Contoh operasi logika: misalkan *X*, *Y*, dan *Z* adalah peubah bertipe *boolean*, *X* bernilai true, *Y* bernilai false, dan *Z* bernilai true, maka:

Operasi logika	Hasil
(X and Y) or Z	true
X and (Y or Z)	true
not (X and Z)	false
(Y xor Z) and Y	false

4.1.2 Bilangan Bulat

Bilangan bulat sudah umum digunakan dalam kehidupan sehari-hari. Bilangan bulat adalah bilangan yang tidak mengandung pecahan desimal, misalnya 34, 8, 1203, 0, -17, 34567892901, dan sebagainya.

Nama Tipe
Nama tipe bilangan bulat adalah *integer*.

Ranah Nilai
Secara teoritis, tipe bilangan bulat mempunyai ranah nilai yang tidak terbatas. Rentang nilainya adalah dari minus tidak berhingga sampai plus tidak berhingga. Tetapi, di dalam komputer tipe *integer* mempunyai ranah nilai yang terbatas. Ranah nilai tipe *integer* pemrograman bergantung pada mesin (komputer) dan kompilator yang digunakan. Pada kompilator Pascal yang digunakan pada komputer 16-bit, rentang nilai *integer* 16-bit adalah dari -32768 sampai +32767. Kompilator ini menyediakan empat macam tipe untuk *integer*, yaitu *byte*, *shortint*, *word*, *integer*, dan *longint*. Rentang nilai untuk kelima tipe tersebut adalah:

Tipe	Rentang nilai	Format
byte	0 .. 255	*Unsigned 8-bit*
shortint	-128 .. 127	*Signed 8-bit*
word	0 .. 65535	*Unsigned 16-bit*
integer	-32768 .. 32767	*Signed 16-bit*
longint	-2147483648 .. 2147483647	*Signed 32-bit*

Dalam bahasa *C*, hanya ada dua macam tipe untuk *integer* yaitu `char` dan `int`. Tipe *char* berukuran 1 *byte*, sedangkan `int` adalah tipe *integer* yang secara tipikal mencerminkan ukuran alami dari *integer* pada kebanyakan mesin. Selain dua tipe tersebut, terdapat beberapa macam *qualifier* yang dapat diterapkan pada tipe *int*, yaitu `short` dan `long`, sehingga kedua tipe ini dapat divariasikan menjadi `short int` dan `long int`. *Qualifier* `short` dan

`long` memberikan panjang yang berbeda pada tipe *int*; *int* normalnya berukuran yang natural pada mesin tertentu. `short` umumnya 16-bit, `long` 32 bit, dan *int* salah satu dari 16-bit atau 32-bit. Tiap-tiap kompilator bahasa *C* bebas memilih ukuran *integer* yang cocok untuk perangkat kerasnya dengan batasan hanya pada `short` dan `int` paling sedikit 16-bit, `long` paling sedikit 32-bit, `short` tidak lebih panjang dari `int`, dan `int` tidak lebih panjang dari `long`. *Qualifier* `signed` dan `unsigned` dapat digunakan pada `char` dan `int`. *Qualifier* `signed` menyebabkan nilai *integer* yang dipakai dapat bertanda negatif, sedangkan `unsigned` menyebabkan nilai *integer* selalu positif atau nol [KER88]. Ekivalensi antara tipe *integer* di dalam kompilator *Turbo Pascal* dan kompilator *Turbo C* dinyatakan dalam tabel berikut:

Pascal	Rentang nilai	C
byte	0 .. 255	unsigned char
shortint	-128 .. 127	signed char
word	0 .. 65535	unsigned int
integer	-32768 .. 32767	int, short int
longint	-2147483648 .. 2147483647	long int

Sebagai contoh, misalkan dua buah peubah *X* dan *Y* masing-masing dideklarasikan bertipe *byte* dan *integer* dalam bahasa *Pascal*:

```
var
  X : byte;
  Y : integer;
```

Dengan mendeklarasikan *X* bertipe *byte* sedangkan *Y* bertipe *integer*, maka peubah *X* tidak dapat dioperasikan untuk nilai-nilai di atas 255 atau di bawah 0. Begitu juga peubah *Y* tidak dapat manampung nilai di atas 32767. Pemilihan implementasi bilangan bulat ke dalam tipe-tipe *integer* yang berbeda lebih disebabkan pada faktor penghematan memori. Sebagai contoh, tipe *word* hanya membutuhkan 2 *byte* memori, sedangkan *integer* membutuhkan 4 *byte* memori.

Tipe bilangan bulat adalah tipe yang memiliki keterurutan. Ini artinya, bila sebuah nilai bilangan bulat diketahui, nilai sebelumnya (*predecessor*) dan nilai sesudahnya (*successor*) dapat ditentukan. Contohnya, *predecessor* dari 8 adalah 7, sedangkan *succesor*-nya adalah 9. Secara formal keterurutan itu didefinisikan sebagai berikut: jika a adalah peubah bertipe bilangan bulat, maka $predecessor(a) = a - 1$, dan $successor(a) = a + 1$.

Konstanta

Konstanta nilai bertipe bilangan bulat harus ditulis tanpa mengandung titik desimal. Contoh beberapa konstanta bertipe bilangan bulat misalnya:

```
78    -14    7654    0    5    9999
```

Operasi

Operasi yang dilakukan terhadap bilangan bulat ada dua macam, yaitu **operasi aritmetika** dan **operasi perbandingan**. Operator yang digunakan pada masing-masing operasi disebut operator aritmetika dan operator perbandingan (atau operator relasional).

a. Operasi Aritmetika

Operasi aritmetika terhadap bilangan bulat dengan sembarang operator aritmetika menghasilkan nilai yang bertipe bilangan bulat juga. Operator aritmetika yang didefinisikan pada bilangan bulat adalah:

```
+          (tambah)
-          (kurang)
*          (kali)
div        (bagi)
mod        (sisa hasil bagi)
```

Operator `div` (div = *divide*) adalah operator bagi yang memberikan hasil pembagian berupa bilangan bulat, sedangkan `mod` (mod = *modulo*) memberikan sisa hasil bagi.

Contoh-contoh operasi aritmetika bilangan bulat beserta hasil operasinya:

```
3 + 10             (hasil: 13)
87 - 31            (hasil: 56)
5 * 10             (hasil: 50)
10 div 3           (hasil: 3)
10 mod 3           (hasil: 1)
20 div 5           (hasil: 4)
20 mod 5           (hasil: 0)
```

Objek yang dioperasikan disebut *operand*. Misalnya pada operasi $a + b$, masing-masing a dan b adalah *operand*, sedangkan "+" adalah operatornya. Semua operator di atas membutuhkan dua buah *operand* dalam pengoperasiannya, sehingga mereka disebut juga **operator biner**. Khusus untuk operator "– ", ia juga merupakan **operator uner**, karena ia dapat dioperasikan dengan satu buah *operand*, misalnya –10, -17, dan sebagainya.

b. Operasi Perbandingan

Operasi perbandingan terhadap bilangan bulat dengan salah satu operator relasional menghasilkan nilai *boolean* (*true* atau *false*). Operator perbandingan untuk bilangan bulat adalah:

```
<          (lebih kecil)
≤          (lebih kecil atau sama dengan)
>          (lebih besar)
≥          (lebih besar atau sama dengan)
=          (sama dengan)
1          (tidak sama dengan)
```

Contoh operasi perbandingan:

```
3 < 8                (true)
74 > 101             (false)
9 ≤ 9                (true)
9 < 9                (false)
17 = 17              (true)
(24 div 3) ≠ 8       (false)
```

4.1.3 Bilangan Riil

Bilangan riil adalah bilangan yang mengandung pecahan desimal, misalnya 3.65, 0.0003, 29.0, 2.60240000E-6, .24, dan lain-lain. Semua konstanta bilangan riil harus mengandung "." (titik). Konstanta "18" dianggap sebagai bilangan bulat, tetapi "18.0" dianggap sebagai bilangan riil. Bilangan riil dapat juga dituliskan dengan notasi E yang artinya perpangkatan sepuluh. Misalnya, 2.60240000E-6 artinya $2.60240000 \times 10^{-6}$. Baik tipe bilangan bulat maupun tipe bilangan riil, keduanya dinamakan juga tipe numerik (*numeric* = angka).

Nama Tipe
Nama tipe bilangan bulat adalah *real* (beberapa literatur menyebutnya *floating-point*)

Ranah Nilai
Sebagaimana halnya pada tipe bilangan bulat, secara teoritis tipe bilangan riil memiliki ranah nilai yang tidak terbatas. Rentang nilainya adalah dari minus tidak hingga sampai plus tidak hingga.

Di dalam komputer, tipe *real* mempunyai rentang nilai yang terbatas, bergantung pada *processor* dan kompilator yang digunakan. Di dalam kompilator *Pascal* misalnya, tipe *real* dapat direpresentasikan ke dalam empat macam tipe, yaitu *real*, *single*, *double*, dan *extended*. Rentang nilai positif untuk keempat tipe tersebut adalah:

Tipe	Rentang Nilai	Format
real	$2.9 \times 10^{-39} .. 1.7 \times 10^{38}$	6 byte
single	$1.5 \times 10^{-45} .. 3.4 \times 10^{38}$	4 byte
double	$5.0 \times 10^{-324} .. 1.7 \times 10^{308}$	8 byte
extended	$3.4 \times 10^{-4933} .. 1.1 \times 10^{4932}$	10 byte

Di dalam bahasa *C*, hanya ada dua tipe untuk bilangan riil, yaitu `float` dan `double`; `float` adalah bilangan riil berpresisi tunggal (*single-precision*), sedangkan `double` adalah bilangan riil berpresisi ganda (*double-precision*). *Qualifier* `long` dapat digunakan untuk menghasilkan ukuran tipe bilangan

riil yang berbeda; long `double` menspesifikasikan tipe bilangan riil yang berpresisi-*extended*.

Konstanta
Setiap konstanta yang bertipe bilangan riil harus ditulis dengan tanda titik desimal. Contoh konstanta bertipe bilangan riil misalnya:

```
0.78    -14.2376    7.654000+E8    0.0    .5    99.0
```

Operasi
Seperti pada tipe bilangan bulat, operasi yang dilakukan terhadap bilangan riil ada dua macam, yaitu operasi aritmetika dan operasi perbandingan. Operator yang digunakan pada masing-masing operasi disebut operator aritmetika dan operator perbandingan.

a. Operasi Aritmetika
Operasi aritmetika terhadap bilangan riil dengan sembarang operator aritmetika menghasilkan nilai yang bertipe bilangan riil juga. Operator aritmetika yang berlaku pada bilangan riil adalah:

```
+    (tambah)
-    (kurang)
*    (kali)
/    (bagi)
```

Umumnya bahasa pemrogramanan membolehkan operasi campuran (*mixed operation*), yaitu operasi aritmetika dengan salah satu *operand*-nya bertipe bilangan riil sedangkan *operand* lainnya bilangan bulat. Jika salah satu *operand* bertipe riil dan *operand* yang lain bertipe bilangan bulat, maka *operand* bilangan bulat akan otomatis dikonversi ke bilangan riil.

Contoh-contoh operasi aritmetika bilangan riil:

```
6.4 + 5.7          (hasil: 12.1)
8.0 - 2.8          (hasil: 5.2)
10.0/3.0           (hasil: 3.333...)
10/2.5             (hasil: 4.0 → operasi bilangan campuran)
7.2*0.5            (hasil: 3.6)
```

b. Operasi Perbandingan
Operasi perbandingan terhadap bilangan riil dengan salah satu operator relasional menghasilkan nilai *boolean* (*true* atau *false*). Operator perbandingan untuk bilangan riil adalah:

```
<    (lebih kecil)
≤    (lebih kecil atau sama dengan)
>    (lebih besar)
≥    (lebih besar atau sama dengan)
≠    (tidak sama dengan)
```

Contoh-contoh operasi perbandingan:

```
0.003 < 0.3          (hasil: false)
8.0 ≥ 5              (hasil: true)
3.0 ≠ 3.5            (hasil: true)
```

Perhatikanlah bahwa kita tidak mengenal perbandingan kesamaan dua buah bilangan riil dengan operator "=". Hal ini dikarenakan bilangan riil tidak dapat disajikan secara eksak di dalam komputer. Misalnya, 1/3 tidak sama dengan 0.33333, sebab 1/3 = 0.33333... (dengan angka tiga yang tidak pernah berhenti), begitu juga 2/3 tidak sama dengan 0.666666666. Karena alasan ini maka operator "=" dianggap tidak ada untuk bilangan riil, sehingga operasi kesamaan dua buah nilai riil tidak kita definisikan.

4.1.4 Karakter

Yang termasuk ke dalam karakter adalah semua huruf abjad, semua tanda baca, angka '0', '1', ..., '9', dan karakter-karakter khusus seperti '&', '^', '%', '#', '@', dan sebagainya. Karakter kosong (*null*) adalah karakter yang panjangnya nol, dan dilambangkan dengan ''.

Nama Tipe
Nama tipe untuk karakter adalah *char*.

Ranah Nilai
Ranah karakter adalah semua huruf di dalam alfabet ('a'..'z', 'A'..'Z'), angka desimal (0..9), tanda baca ('.', ':', '!', '?', ',', dan lain-lain), operator aritmetik ('+', '-', '*', '/'), dan karakter-karakter khusus seperti ('$', '#', '@', '^', '~', dan lain-lain). Daftar karakter baku yang lengkap dapat dilihat di dalam buku-buku yang memuat tabel ASCII.

Konstanta
Konstanta karakter harus diapit oleh tanda petik tunggal. Contoh konstanta karakter misalnya:

```
'h'  'Y'   '.'  ' '  'p' '+'  '9'  '0'  '$'
```

Hati-hati menuliskan bahwa '9' adalah karakter, tetapi 9 adalah *integer*!

Operasi
Operasi yang dapat dilakukan pada tipe karakter adalah operasi perbandingan. Operator perbandingan yang berlaku untuk tipe karakter adalah:

```
=       (sama dengan)
≠       (tidak sama dengan)
<       (lebih kecil)
>       (lebih besar)
≥       (lebih besar atau sama dengan)
```

Seperti halnya pada tipe bilangan bulat, tipe karakter juga mempunyai keterurutan (*successor* dan *predecessor*) yang ditentukan oleh cara pengodeannya di dalam komputer, misalnya pengodean ASCII. Karena adanya keterurutan tersebutlah maka kita dapat mendefinisikan operator <, ≤, >, dan ≥ pada tipe karakter. Operasi dengan operator perbandingan menghasilkan nilai *boolean*. Misalnya, bila *a* dan *b* adalah peubah bertipe karakter, maka operasi *a* < *b* akan menghasilkan nilai benar atau salah bergantung pada keterurutan harga *a* dan *b* pada sistem pengodeannya.

Contoh:

```
'a' = 'a'       (hasil: true)
'T' = 't'       (hasil: false)
'y' ≠ 'Y'       (hasil: true)
'm' < 'z'       (hasil: true)
'Q' > 'Z'       (hasil: false)
```

4.1.5 *String*

String adalah untaian karakter dengan panjang tertentu. *String* sebenarnya bukan tipe dasar murni karena ia disusun dari elemen-elemen bertipe karakter. Namun, karena tipe *string* sering dipakai dalam pemrograman, maka *string* dapat diperlakukan sebagai tipe dasar.

Nama Tipe
Nama tipe *string* adalah *string*.

Ranah Nilai
Ranah nilai untuk tipe *string* adalah deretan karakter yang telah didefinisikan pada ranah karakter.

Konstanta
Semua konstanta *string* harus diapit oleh tanda petik tunggal. Contoh-contoh konstanta *string* misalnya:

```
'BANDUNG'
'ganesha'
'Jl. Pahlawan No. 76'
' Jurusan Teknik Informatika'
'.......................'
'ABCD765'
'K7685302'
'm'
```

String kosong (*null*) adalah *string* yang panjangnya nol, dan dilambangkan dengan ''. Dengan kata lain, *string* kosong sama dengan karakter kosong.

Operasi
Operasi terhadap data bertipe *string* didefinisikan dua macam:

a. Operasi Penyambungan (*Concatenation*)
Operator: +
Operator "+" yang dimaksudkan di sini berarti penyambungan. Bila *a* dan *b* adalah peubah bertipe *string*, maka *a* + *b* sama dengan *ab*.

Contoh:

```
'Teknik' + 'INFORMATIKA' = 'TeknikINFORMATIKA'
'Teknik' + ' INFORMATIKA' = 'Teknik INFORMATIKA'
'aaa' + '  bbb' + 'cc' = 'aaa  bbbcc'
'1' + '2'= '12' dan '2' + '1' = '21'     (namun 1 + 2 = 3, mengapa?)
```

b. Operasi Perbandingan
Operator:

```
=          (sama dengan)
≠          (tidak sama dengan)
<          (lebih kecil)
>          (lebih besar)
≤          (lebih kecil atau sama dengan)
≥          (lebih besar atau sama dengan)
```

Operasi perbandingan, seperti halnya pada karakter, menghasilkan nilai *boolean* (*true* atau *false*). Bila dua *string* dibandingkan, maka yang terjadi adalah perbandingan elemen-elemen karakter dari kedua *string* pada posisi yang sama.

Contoh:

```
'abcd' = 'abc'     (hasil: false)
'aku ' < 'AKU'     (hasil: true)
```

String yang disusun oleh gabungan numerik dan karakter sering dinamakan *alfanumerik*. Misalnya 'K7685302', 'D5432AB', dan sebagainya.

4.2 Tipe Bentukan

Tipe bentukan adalah tipe yang didefinisikan sendiri oleh pemrogram (*user-defined type data*). Tipe bentukan disusun oleh satu atau lebih tipe dasar. Ada dua macam tipe bentukan:
1. tipe dasar yang diberi nama dengan nama tipe baru,
2. tipe terstruktur.

4.2.1 Tipe Dasar yang Diberi Nama Tipe Baru

Kadang-kadang pemrogram ingin memberi nama baru terhadap tipe dasar yang sudah dikenal. Alasan pemberian nama baru mungkin agar nama baru tersebut lebih "akrab" dan lebih mudah diinterpretasi oleh orang yang membaca teks algoritma.

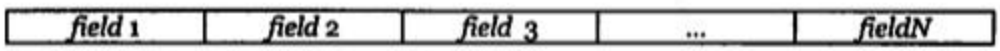
Kita dapat memberi nama baru untuk tipe dasar tersebut dengan kata kunci `type`. Ranah nilai, cara menulis konstanta, dan operasi-operasi terhadap tipe baru tersebut tidak berubah, tetap sama dengan tipe dasar aslinya.

Contoh:
```
type BilanganBulat : integer
```

`BilanganBulat` adalah tipe bilangan bulat yang sama saja dengan tipe *integer*. Apabila kita mempunyai sebuah peubah (*variable*) yang bernama `P` dan bertipe `BilanganBulat`, peubah `P` tersebut sama saja bertipe *integer*.

4.2.2 Tipe Terstruktur

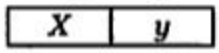
Tipe terstruktur adalah tipe yang berbentuk rekaman (*record*). Rekaman disusun oleh satu atau lebih *field* (Gambar 4.1). Tiap *field* menyimpan data dari tipe dasar tertentu atau dari tipe bentukan lain yang sudah didefinisikan sebelumnya. Nama rekaman ditentukan sendiri oleh pemrogram.



Gambar 4.1 Tipe terstruktur dengan N Buah *Field*

Contoh-contoh di bawah ini memperlihatkan bagaimana mendefenisikan tipe bentukan untuk tipe terstruktur.

1) Titik dalam koordinat kartesian dinyatakan sebagai (x, y), dengan x adalah nilai absis dalam arah sumbu-X dan y adalah nilai ordinat dalam arah sumbu-Y. Kita dapat mendefinisikan titik sebagai tipe bentukan baru dengan x dan y sebagai nama *field*-nya.



Cara menuliskan tipe `Titik`:

```
type Titik : record < x : real, y : real>
```

atau

```
type Titik : record < x, y : real>
```

Kata kunci `record` menyatakan bahwa `Titik` adalah tipe terstruktur. *Jika* dideklarasikan `P` adalah peubah (*variable*) bertipe `Titik`, maka cara mengacu tiap *field* pada `P` adalah:

```
P.x
P.y
```

Tipe `Titik` didefinisikan sebagai berikut:

Nama tipe	: `Titik`
Ranah nilai	: `(real,real)`

Contoh konstanta	: `<2.7,13.2>`
	`<-1.4,-6.5>`
	`<3.034,-10.12>`

Operasi	: operasi aritmetik bilangan riil terhadap x dan y operasi perbandingan terhadap x dan y

2) Bilangan kompleks adalah bilangan yang dapat dinyatakan sebagai $a + bi$ dengan a dan b riil sedangkan $i = \sqrt{-1}$. Misalnya $4.0 + 3.8i$, $9.4 - 2.5i$, dan sebagainya. Kita dapat menyatakan sebuah bilangan kompleks sebagai tipe rekaman dengan a dan b sebagai nama *field*-nya.

a	*b*

Cara menuliskan tipe `Kompleks`:

```
type Kompleks : record <a:real, b:real>
```

Jika dideklarasikan `K` adalah peubah bertipe `Kompleks`, maka cara mengacu tiap *field* pada `K` adalah:

```
K.a
K.b
```

Definisi tipe `Kompleks` sebagai berikut:

Nama tipe	: `Kompleks`
Ranah nilai	: `(real, real)`
Contoh konstanta	: `<4.0,3.8>, <9.4,2.5>, <23.034, -100.0>`
Operasi	: - operasi aritmetik bilangan riil terhadap a dan b - operasi perbandingan terhadap masing-masing *field*

3) Didefinisikan tipe terstruktur yang mewakili tanggal dalam kalender Masehi. Hari dinyatakan sebagai tanggal (*dd*), bulan (*mm*), dan tahun (*yy*), misalnya 10-8:1999. Misalkan tipe bentukan tersebut diberi nama *Tanggal*:

dd	*mm*	*yy*

Cara menuliskan tipe `Tanggal`:

```
type Tanggal : record
               <dd : integer,      {1..31}
                mm : integer,      {1..12}
                yy : integer       { > 0 }
               >
```

Jika `D` adalah peubah bertipe `Tanggal`, maka cara mengacu tiap *field*:

```
D.dd
D.mm
D.yy
```

Tipe `Tanggal` didefinisikan sebagai berikut:

Nama tipe : `Tanggal`
Ranah nilai : sesuai ranah masing-masing *field*

Contoh konstanta : `<12,7,1997>, <31,12,1980>, <29,2,1980>`
Operasi : - operasi aritmetik bilangan bulat terhadap tiap *field*
- operasi perbandingan terhadap masing-masing *field*

4) Didefinisikan tipe terstruktur yang mewakili jam. Jam dinyatakan sebagai jam (*hh*), menit (*mm*), dan detik (*ss*), contohnya 12:45:10 (jam 12 lewat 45 menit lewat 10 detik). Misalkan tipe bentukan tersebut diberi nama `Jam`.

dd	*mm*	*ss*

Cara menuliskan tipe `Jam`:

```
type Jam : record
           <hh    : integer,     {0..23}
            mm    : integer,     {0..59}
            yy    : integer      {0..59}
           >
```

Jika `J` adalah peubah bertipe `Jam`, maka cara mengacu tiap *field*:

```
J.hh
J.mm
J.ss
```

Tipe `Jam` didefinisikan sebagai berikut:

Nama tipe : `Jam`
Ranah nilai : sesuai ranah masing-masing *field*
Contoh konstanta : `<12,45,10>, <23,12,19>, <9,17,8>`
Operasi : - operasi aritmetik bilangan bulat terhadap tiap *field*
- operasi perbandingan terhadap masing-masing *field*

5) Tipe terstruktur untuk jadwal kereta api. Jadwal keberangkatan kereta api terdiri atas informasi nomor kereta api (*NoKA*), kota asal (*KotaAsal*), kota tujuan (*KotaTujuan*), jam berangkat (*JamBerangkat*), dan jam tiba (*JamTiba*).

Cara menuliskan tipe `Jadwal_KA`:

```
type Jadwal_KA : record
                 <NoKA          : string,
                  KotaAsal      : string,
                  JamBerangkat  : Jam,
                  KotaTujuan    : string,
                  JamTiba       : Jam
                 >
```

Jika `JKA` adalah peubah bertipe `Jadwal_KA`, maka cara mengacu tiap-tiap *field*:

```
JKA.NoKA
JKA.KotaAsal
JKA.JamBerangkat                { mengacu jam keberangkatan }
JKA.JamBerangkat.hh             { mengacu jam (hh) keberangkatan }
JKA.JamBerangkat.mm             { mengacu menit (mm) keberangkatan }
JKA.JamBerangkat.ss             { mengacu detik (ss) keberangkatan }
JKA.KotaTujuan
JKA.JamTiba.hh
JKA.JamTiba.mm
JKA.JamTiba.ss
```

Tipe `Jadwal_KA` didefinisikan sebagai berikut:

Nama tipe : `Jadwal_KA`
Ranah nilai : sesuai ranah masing-masing *field*
Contoh konstanta :
```
<'KA01', 'Jakarta', <17,2,0>, 'Semarang',
<05,54,0> >
<'KA24'>, 'Bandung', <9,40,12>, 'Solo',
<14,10,50> >
```
Operasi : sesuai operasi untuk masing-masing tipe *field*.

6) `NilMhs` adalah nama tipe terstruktur yang menyatakan nilai ujian seorang mahasiswa untuk suatu mata kuliah (*MK*) yang ia ambil. Data setiap mahasiswa adalah *NIM* (Nomor Induk Mahasiswa), nama mahasiswa, kode mata kuliah yang ia ambil, dan nilai mata kuliah tersebut.

NIM	*NamaMhs*	*KodeMK*	*Nilai*

Cara menuliskan tipe `NilMhs`:

```
type NilMhs : record
              < NIM     : integer,  {Nomor Induk Mahasiswa }
                NamaMhs : string,   {nama mahasiswa }
                KodeMK  : string,   {kode mata kuliah }
                Nilai   : char      {indeks nilai MK (A/B/C/D/E)}
              >
```

Jika dideklarasikan M adalah peubah bertipe NilMhs, maka Cara mengacu tiap *field* pada rekaman M adalah:

```
M.NIM
M.NamaMhs
M.KodeMK
M.Nilai
```

Tipe NilMhs didefinisikan sebagai berikut:

Nama tipe : NilMhs
Ranah nilai : sesuai ranah masing-masing *field*
Contoh konstanta : <13596001,'Amir Ali','FI324','A'>
<13597010,'Wati Siregar','MA211,'C'>
Operasi : - operasi aritmetik bilangan bulat terhadap NIM
- operasi *string* terhadap NamaMhs, KodeMK, Nilai

7) Contoh berikut menyajikan struktur tipe bentukan yang cukup kompleks. Misalkan kita ingin menyimpan data nilai-nilai mata kuliah yang diambil setiap mahasiswa. Data yang disimpan di dalam struktur rekaman sebagai berikut:
   1. *NIM* (Nomor Induk mahasiswa)
   2. *NamaMhs* (nama mahasiswa)
   3. Mata kuliah yang diambil mahasiswa tersebut, yang terdiri atas:
      - Kode mata kuliah
      - Nama mata kuliah
      - Nilai mata kuliah

Cara menuliskan tipe MataKuliah:

```
type MataKuliah : record
                  <KodeMK  : string,    { kode mata kuliah }
                   NamaMK  : string,    { nama mata kuliah }
                   Nilai   : char       { indeks nilai MK (A/B/C/D/E) }
                  >

type NilMhs  : record
               <NIM     : string,    { Nomor Induk Mahasiswa }
                NamaMhs : string,    { nama mahasiswa }
                MK      : MataKuliah
               >
```

4.3 Nama

Setiap objek di alam mempunyai nama. Nama itu diberikan oleh manusia agar objek tersebut mudah diidentifikasi, diacu, dan dibedakan dari objek lainnya. Di dalam algoritma, nama dipakai sebagai pengidentifikasi "sesuatu" dan pemrogram mengacu "sesuatu" itu melalui namanya. Karena itu, tiap nama haruslah unik, tidak boleh ada dua buah nama yang sama.

Beberapa aturan mengenai penamaan:

1. Nama harus dimulai dengan huruf alfabet, tidak boleh dimulai dengan angka, spasi, atau karakter khusus lainnya;
2. Huruf besar atau huruf kecil tidak dibedakan. Jadi, suatu nama yang ditulis dalam huruf besar atau huruf kecil dianggap sama;
3. Karakter penyusun nama hanya boleh huruf alfabet, angka, dan "_". Karakter garis bawah (*underscore*) dihitung sebagai sebuah huruf; karakter ini kadang-kadang berguna untuk memudahkan membaca algoritma.
4. Nama tidak boleh mengandung operator aritmetika, operator relasional, tanda baca, dan karakter khusus lainnya;
5. Karakter-karakter di dalam nama tidak boleh dipisah dengan spasi. Hal ini mengingat bahwa nama adalah kesatuan leksikal, maka setiap nama harus ditulis secara utuh. Spasi dapat diganti dengan karakter "_"
6. Panjang nama tidak dibatasi.

Nama sebaiknya interpretatif yaitu mencerminkan nilai intrinsik atau fungsi yang dikandungnya. Kita sebagai pemrogram sangat dianjurkan memberikan keterangan pada nama yang didefinisikan. Keterangan tersebut dinyatakan sebagai komentar yang diapit oleh sepasang tanda kurung kurawal.

Contoh nama yang salah:

```
6titik          { karena dimulai dengan angka }
nilai ujian     { karena dipisahkan dengan spasi }
PT-1            { karena mengandung operator kurang }
hari!           { karena mengandung karakter khusus }
A 1             { karena mengandung spasi }
```

Contoh nama yang benar:

```
titik6  atau  titik_6
nilai_ujian  atau  nilaiujian
PT_1   atau  PT1
hari
A1
```

Semua nama yang digunakan di dalam algoritma harus dideklarasikan di dalam bagian DEKLARASI sebelum mereka digunakan. Deklarasi menspesifikasikan nama-nama peubah beserta tipenya, nama konstanta beserta nilainya, nama tipe bentukan, dan nama prosedur/ fungsi.

Di dalam algoritma, objek yang diberi nama dapat berupa:

1. **Peubah (*variable*);**
   Peubah adalah tempat menyimpan nilai yang isinya dapat diubah.

   Contoh:

```
DEKLARASI
  X, nilai_ujian, jumlah : real
```

```
k : integer
c : char
```

X, nilai_ujian, jumlah, k, dan c adalah nama-nama peubah. Nilai-nilai peubah dapat dimanipulasi melalui pernyataan-pernyataan di dalam algoritma.

2. **Konstanta (*constant*);**
Konstanta adalah tempat penyimpanan di dalam memori yang nilainya tetap selama pelaksanaan program dan tidak boleh diubah. Notasi untuk menyatakan konstanta adalah const.

Contoh:

```
DEKLARASI
  const phi = 3.14
  const Nmaks = 100
  const sandi = 'xyz'
```

phi, Nmaks, dan sandi adalah nama konstanta. Nilai konstanta yang sudah diisikan ke dalam nama konstanta tidak dapat diubah oleh instruksi di dalam algoritma.

3. **Tipe bentukan, seperti yang telah dijelaskan di atas;**
Nama tipe bentukan diberikan oleh pemrogram;

Contoh:

```
DEKLARASI
  type Titik : record <x : real, y : real>
  type Jam : record
             <hh : integer, {0..23}
              mm : integer, {0..59}
              ss : integer  {0..59}
             >
  P : Titik
  J1, J2 : Jam
```

Titik dan Jam adalah nama tipe, sedangkan P adalah nama peubah yang bertipe Titik, J1 dan J2 adalah nama peubah yang bertipe Jam.

4. Nama fungsi (akan dijelaskan kemudian) yang digunakan;

Contoh:

```
DEKLARASI
  FUNCTION Maksimum(input A, B : integer) → integer
( mengembalikan nilai terbesar antara A dan B )
```

Maksimum adalah nama fungsi.

5. Nama prosedur (akan dijelaskan kemudian) yang digunakan.

Contoh:

```
DEKLARASI
    PROCEDURE Tukar(input/output A, B : integer)
    { Mempertukarkan nilai A dan B }
```

Tukar adalah nama prosedur.

4.4 Tabel Translasi dari Algoritmik ke Pascal dan C

Kelompok	Algoritmik	Pascal	C	Ket.
1. Tipe Dasar	boolean	boolean	(Lihat ket. di bawah tabel)	
	integer	byte shortint word integer longint	unsigned char signed char unsigned int int, short int long int	1 byte 1 byte 2 byte 2 byte 4 byte
	real	real double extended	float double long double	(pada komputer 16-bit)
	char	char	Char	
	string	string string[n]	(Lihat ket.) char[n]	
	record <field1:type, field2:type, ... fieldN:type >	record field1: type; field2: type; ... fieldN: type; end;	struct {type field1; type field2; ... type fieldN; }	
2. Operator a. aritmetik	+ - * / div mod	+ - * / div mod	+ - * / / %	
b. perbandingan	<	<	<	

	≤ > ≥ = ≠	<= > >= = <>	<= > >= == !=	
c. logika	not and or xor	not and or xor	! && || tidak ada	
d. string	+ < ≥ > ≤ = ≠	+ < >= > <= = <>	tidak ada tidak ada tidak ada tidak ada tidak ada tidak ada tidak ada	
3. Komentar	{ komentar }	{ komentar } (* komentar *)	/* komentar */	
4. Lain-lain	const type true false	const type true false	const typedef dapat diganti 1 dapat diganti 0	

Keterangan Tambahan:

1. Bahasa *C* tidak menyediakan tipe *boolean* secara eksplisit. Tipe *boolean* dapat diganti dengan tipe *integer*, yang dalam hal ini *true* = 1, *false* = 0. Namun demikian, kita dapat membentuk tipe baru yang bernama *boolean* dengan typedef sebagai berikut:

```
typedef enum {false = 0, true = 1} Boolean;
```

   Tipe `Boolean` di atas selanjutnya dapat digunakan untuk mendeklarasikan peubah yang bertipe *boolean*:

```
Boolean ketemu;   /* ketemu adalah peubah bertipe boolean */
```

   Selanjutnya, `false` dan `true` dapat digunakan sebagai konstanta *boolean* di dalam program *C*.

2. Sebuah karakter atau *string* dalam bahasa Pascal diapit dengan sepasang tanda petik tunggal ('...') , sedangkan dalam bahasa *C* karakter

2. Sebuah karakter atau *string* dalam bahasa Pascal diapit dengan sepasang tanda petik tunggal ('...') , sedangkan dalam bahasa *C* karakter diapit oleh sepasang tanda petik tunggal ('...') dan sebuah *string* diapit oleh sepasang tanda petik ganda ("...").

   Contoh dalam bahasa *Pascal*:

```
'a'                          (karakter)
'Teknik Informatika'         (string)
```

   Contoh dalam bahasa *C*:

```
'a'                          (karakter)
"Teknik Informatika"         (string)
```

   Perhatikan bahwa di dalam bahasa *C*, `'a'` adalah sebuah karakter, sedangkan `"a"` adalah sebuah *string*.

3. Dalam bahasa *Pascal*, *string* yang tidak disertai dengan panjang maka diasumsikan panjangnya adalah *default*, yaitu 255. Apabila panjangnya ditentukan, panjang tersebut harus disebutkan.

   Contoh: `kota : string[20];`

4. Bahasa *C* tidak menyediakan tipe *string* secara khusus. *String* di dalam bahasa *C* direpresentasikan dengan larik (*array* – materi larik diberikan di dalam Bab 13)) Namun, kita dapat membuat tipe bentukan baru untuk *string* dengan menggunakan larik dan fasilitas typedef dan panjang *string* maksimum 255 (seperti nilai *default* panjang *string* di bahasa *Pascal*)

```
typedef char String[255];
```

   Tipe `String` ini selanjutnya dapat digunakan untuk mendeklarasikan peubah yang bertipe *string*:

```
String p, kota;   /* p dan kota adalah peubah bertipe string */
```

   Jika panjang *string* ditetapkan, maka nama peubah yang bertipe *string* didefinisikan sebagai *array* karakter disertai panjang *string* tersebut:

```
char nama_peubah[n];
```

   dengan n adalah panjang *string*.

   Contoh: `char kota[20];`

5. Dua buah *string* di dalam bahasa *C* tidak dapat dibandingkan secara langsung, karena *string* pada dasarnya adalah *array of character* (lihat Bab tentang *array*, Bab 13). Namun, kita dapat membandingkan kesamaan dua buah *string* dengan menggunakan fungsi strcmp. Fungsi ini menghasilkan nilai perbandingan dua buah *string*, *s1* dan *s2* dengan format

```
nilai = strcmp(s1,s2)
```
yang dalam hal ini, *nilai* = 0 jika s1 = s2, *nilai* < 0 jika s1 < s2, dan *nilai* > 0 jika s1 > s2. Lihat contoh penggunaan fungsi strcmp ini di dalam Bab 6.

6. Pendeklarasian konstanta dalam bahasa *C*, selain menggunakan kata kunci `const`, dapat juga menggunakan pernyataan baris `#define`. Pernyataan `#define` mendefinisikan konstanta simbolik dengan cara penulisan:

```
#define  nama   nilai
```

Pernyataan di atas berarti, setiap kemunculan *nama* di dalam program akan diganti dengan *nilai*.

Contoh:

```
/* nama konstanta */
const float phi = 3.14;        /* konstanta π */
const int N = 100;             /* jumlah mahasiswa */
const char sandi[]= "xyz";     /* kunci rahasia */
```

atau

```
/* nama konstanta */
#define PHI   3.14       /* konstanta π */
#define N     100        /* jumlah mahasiswa */
#define SANDI "xyz"      /* kunci rahasia */
```

Pernyataan `#define` harus diletakkan di luar blok `main()`. Perhatikan bahwa tidak diperlukan karakter ";" pada akhir pernyaatan `#define`. Nama-nama seperti `PHI`, `N`, dan `SANDI` adalah konstanta simbolik, bukan peubah, sehingga mereka tidak muncul di bagian deklarasi. Nama-nama konstanta simbolik menurut konvensi ditulis dengan huruf besar untuk membedakannya dengan nama peubah.

Contoh-contoh translasi

Contoh translasi bagian DEKLARASI ke dalam bahasa *Pascal* dan *C* diberikan di bawah ini. Untuk prosedur dan fungsi, pembaca dapat membaca Bab 10 dan Bab 11 tentang cara mentranslasikan prosedur dan fungsi ke dalam bahasa *Pascal* dan *C*. Kata-kata yang dicetak tebal di dalam bahasa *Pascal* dan *C* adalah *reserved words* (kata cadangan). Kata cadangan tidak boleh dijadikan sebagai nama tipe, nama peubah, nama tetapan, nama fungsi, dan nama prosedur.

ALGORITMIK

```
DEKLARASI
   { nama konstanta }
   const phi = 3.14        { konstanta π }
   const Nmaks = 100       { jumlah mahasiswa }
```

```
const sandi = 'xyz'  { kunci rahasia }

{ nama tipe }

type MHS: record        { data mahasiswa }
            < NIM  : integer,
              nama : string,
              usia :integer
            >

type Titik : record <x:real, y:real>      { titik di dalam bidang
                                            kartesian }
type Kompleks : record <a:real, b:real>   { bilangan kompleks }

type Jam : record
            <hh : integer, {0..23}
             mm : integer, {0..59}
             ss : integer  {0..59}
            >

type JadwalKA : record
                  < NoKA          : string,
                    KotaAsal      : string,
                    JamBerangkat  : Jam,
                    KotaTujuan    : string,
                    JamTiba       : Jam
                  >

{ nama peubah }
   luasL : real          { luas lingkaran }
   psw : string          { password }
   indeks : char         { indeks nilai ujian }
   ketemu : boolean      { hasil pencarian, true jika data yang
                           dicari ditemukan, false jika sebaliknya}
   P : Titik             { koordinat titik pusat lingkaran }
   F : Kompleks          { frekuensi, hasil transformasi Fourier }
   JKA : JadwalKA        { jadwal kereta api ekspres }

   { nama fungsi dan prosedur }
   PROCEDURE HitungTitikTengah(input P1:Titik, input P2:Titik,
                               output Pt:Titik)

   {Menghitung titik tengah garis dengan titik ujung P1 dan P2. }

   FUNCTION Faktorial(input n:integer)→integer
   { mengembalikan nilai faktorial dari n > 0 }

   FUNCTION Cari(input x:integer)→ boolean
   { true bila x ditemukan, false bila sebaliknya }

   PROCEDURE CetakJadwalKA(input kota : string)
   { Mencetak jawal semua kereta api yang berangkat dari kota tertentu
}
```

PASCAL:

```
(* DEKLARASI *)

{ nama konstanta }
const phi = 3.14;       { konstanta π }
      Nmaks = 100;      { jumlah mahasiswa }
      sandi = 'xyz';    { kunci rahasia }


{ nama tipe }
type
    MHS = record        { data mahasiswa }
            NIM  : integer;
            nama : string;
            usia : integer;
          end;

    Titik = record     { titik di dalam bidang kartesian }
              x:real;
              y:real;
            end;

    Kompleks = record     { bilangan kompleks }
                 a:real;
                 b:real;
               end;

    Jam  = record
             hh : integer;  {0..23}
             mm : integer;  {0..59}
             ss : integer;  {0..59}
           end;

    JadwalKA = record
                 NoKA : string[5];
                 KotaAsal : string[15];
                 JamBerangkat : Jam;
                 KotaTujuan : string;
                 JamTiba : Jam;
               end;

var
   { nama peubah }
   luasL : real;        { luas lingkaran }
   psw : string;        { password }
   indeks : char;       { indeks nilai ujian }


   ketemu : boolean;  { hasil pencarian, true jika data yang
                        dicari ditemukan, false jika sebaliknya}
   P : Titik;         { koordinat titik pusat lingkaran }
   F : Kompleks;      { frekuensi, hasil transformasi Fourier }
   JKA : JadwalKA;    { jadwal kereta api ekspres }


   { nama fungsi dan prosedur }
```

```
   PROCEDURE HitungTitikTengah(P1:Titik; P2:Titik; var Pt:Titik);
   {Menghitung titik tengah garis dengan titik ujung P1 dan P2. }

   FUNCTION Faktorial(n:integer):integer;
   { mengembalikan nilai faktorial dari n > 0 }

   FUNCTION Cari(x:integer): boolean;
   { true bila x ditemukan, false bila sebaliknya }

   PROCEDURE CetakJadwalKA(kota : string[15]);
   { Mencetak jawal semua kereta api yang berangkat dari kota tertentu
}
```

C:

```
/* DEKLARASI */

/* nama konstanta */
const float phi = 3.14;       /* konstanta π */
const int N = 100;            /* jumlah mahasiswa */
const char sandi[]= "xyz";    /* kunci rahasia */

/* nama tipe */
typedef struct {int NIM;
                char nama[25];
                int usia;
               } MHS;  /* data mahasiswa */

typedef struct {float x; float y;} Titik;    /*titik di bidang
kartesian */

typedef struct {float a; float b;} Kompleks;/*bilangan kompleks*/

typedef struct {int hh;
                int mm;
                int ss;
               }Jam;

typedef struct { char NoKA[5];
                 char KotaAsal[15];
                 Jam JamBerangkat;
                 char KotaTujuan[15];
                 Jam JamTiba;} JadwalKA;
typedef enum {false=0, true=1} boolean; /* tipe bentukan untuk boolean
*/

/* nama peubah */
float luasL;        /* luas lingkaran */
char psw[255];      /* password */
char indeks;        /* indeks nilai ujian */


boolean ketemu;     /* hasil pencarian, true jika data
                       dicari ditemukan, false jika sebaliknya */
Titik P;           /* koordinat titik pusat lingkaran  */
Kompleks F;        /* frekuensi, hasil transformasi Fourier */
JadwalKA JKA;      /* jadwal kereta api ekspres */
```

```
/* nama fungsi dan prosedur */

void HitungTitikTengah(Titik P1, Titik P2, Titik *P);
/* Menghitung titik tengah garis dengan titik ujung P1 dan P2. */

int Faktorial(int n);
/* mengembalikan nilai faktorial dari n > 0 */

boolean Cari(int x);
/* true bila x ditemukan, false bila sebaliknya */

void CetakJadwalKA(char kota[15]);
/* Mencetak jawal semua kereta api yang berangkat dari kota tertentu
*/
```

4.5 Nilai

Nilai adalah besaran dari tipe data yang terdefinisi (tipe dasar atau tipe bentukan). Nilai dapat berupa data yang disimpan di dalam peubah atau konstanta, nilai dari hasil perhitungan, atau nilai yang dikirim oleh fungsi. Algoritma pada dasarnya memanipulasi nilai yang disimpan di dalam peubah. Memanipulasi nilai di dalam peubah misalnya: mengisikannya ke peubah lain yang bertipe sama, dipakai untuk perhitungan, atau dituliskan ke piranti keluaran. Masing-masing cara memanipulasi ini dibahas satu-per satu di bawah ini.

4.5.1 Pengisian Nilai ke dalam Nama Peubah

Peubah menyimpan sebuah nilai bertipe sama dengan tipe peubah. Sebuah nilai dapat dimasukkan ke dalam nama peubah dengan dua buah cara:

1. pengisian nilai secara langsung (*assignment*),
2. pembacaan.

4.5.1.1 Pengisian Nilai Secara Langsung

Pengisian nilai secara langsung (*assignment*) adalah mengisi sebuah nilai ke dalam peubah secara langsung di dalam algoritma. Nilai yang diisikan harus bertipe sama dengan tipe peubah.

Notasi pengisian nilai secara langsung: ←

Arti notasi ←: nilai di sebelah kanan tanda panah diisikan ke dalam peubah di sebelah kirinya.

(a) Nilai yang diisikan ke dalam peubah dapat berupa konstanta,

```
peubah ← konstanta    { nilai konstanta diisikan ke dalam peubah }
```

Contoh: `A ← 10` `{ Nilai A sama dengan 10 }`

(b) atau suatu peubah diisi dengan nilai dari peubah lain:

```
peubah2 ← peubah1    { nilai peubah1 disalin (copy) ke dalam peubah2 }
```

Contoh: `A ← B` `{ Nilai A disalin ke dalam B }`

(c) atau suatu peubah diisi dengan hasil evaluasi dari sebuah ekspresi:

```
peubah ← ekspresi    { hasil perhitungan diisikan ke dalam peubah }
```

Contoh: `A ← (B+C)/2` `{ A berisi hasil evaluasi (B+C)/2 }`

Setiap peubah menempati sebuah lokasi di memori komputer. Sifat memori adalah jika ia diisi dengan nilai baru, maka nilai yang disimpan sebelumnya hilang. Jadi, jika suatu peubah diisi dengan nilai baru, maka nilai lama yang disimpan oleh peubah tersebut "ditimpa" dengan nilai yang baru. Prinsip yang dipakai adalah: nilai yang dikandung oleh peubah adalah nilai yang terakhir kali diisikan ke dalamnya.

Misalkan peubah `k` dideklarasikan bertipe *integer*, `jarak` bertipe *real*, `ketemu` bertipe *boolean*, `NamaKota` bertipe *string*, dan `J` bertipe `Jam`.

```
DEKLARASI
   k        : integer
   jarak    : real
   ketemu   : boolean
   NamaKota : string
   type Jam : record
                <hh : integer, {0..23}
                 mm : integer, {0..59}
                 ss : integer  {0..59}
                >
   J        : Jam
```

Contoh pengisian peubah-peubah tersebut dengan nilai-nilai konstanta adalah:

```
ALGORITMA:
   k ← 5
   jarak ← 0.03
   ketemu ← false

   NamaKota ← 'SOLO'

   { Mengisi J dengan jam 6:12:39 }
   J ← <6,12,39>
```

```
{ atau, kita bisa mengisi setiap field dari J:
  J.hh ← 6
  J.mm ← 12
  J.ss ← 39
}
```

Tetapi, pengisian langsung nilai di bawah ini salah:

```
k ← 0.5
```

karena `k` yang bertipe *integer* diisi dengan nilai bertipe riil.

Misalkan `B`, `awal`, `suhu1`, `suhu2`, `ulang`, dan mulai adalah nama peubah yang didefinisikan bertipe *integer*. Contoh di bawah ini adalah mengisi sebuah peubah dengan nilai dari peubah lain.

```
DEKLARASI
   B, awal, suhu1, suhu2, ulang : integer

ALGORITMA:
   B ← awal
   suhu1 ← suhu2
   ulang ← mulai
```

Peubah di ruas kanan harus sudah berisi nilai terlebih dahulu. Jadi, jika `suhu1` berisi `70` dan `suhu2` sudah berisi nilai `80`, maka pengisian nilai

```
suhu1 ← suhu2
```

menyebabkan peubah `suhu1` sekarang berisi nilai `80`. Nilai yang dikandung `suhu1` sebelumnya (70) hilang digantikan 80.

Perhatikan bahwa pernyataan pengisian nilai

```
nama_var1 ← nama_var2
```

sama artinya dengan menyalin (*copy*) nilai *nama_var2* ke dalam *nama_var1*. Penyalinan itu tidak menyebabkan nilai *nama_var2* hilang. Nilai *nama_var2* tetap seperti semula.

Contoh berikut ini adalah pengisian peubah dengan ekspresi. Hasil ekspresi harus bertipe sama dengan tipe peubah di ruas kiri.

```
DEKLARASI
   P, Q, R, nisbah, jumlah, banyak, i, j, k : integer
   a, b, c, determinan : real
   h : boolean

ALGORITMA:
   C ← A + B
   nisbah ← jumlah/banyak * 5
   determinan ← b*b - 4*a*c
```

```
k ← (i+j) div 2
h ← (i > 1) and (i ≤ 100)
```

Bila A didefinisikan dideklarasikan bertipe *integer*, maka pernyataan:

```
A ← A + 1
```

berarti nilai A yang baru adalah nilai A yang lama ditambah satu, atau

$A_{baru} = A_{lama} + 1$

Misalkan A bernilai 8, maka setelah pernyataan A←A+1, nilai A sekarang adalah 9 (yaitu 8 + 1).

Berapa nilai M dan P setelah runtunan penugasan di bawah ini selesai dilaksanakan oleh komputer?

```
DEKLARASI
   M, P : integer

ALGORITMA:
   M ← 16
   P ← M * 2
   M ← P + 100
   P ← P + M
```

Jawabnya adalah: di akhir instruksi, M = 132 dan P = 164. Periksa kebenarannya.

4.5.1.2 Pembacaan

Nilai peubah dapat diisi secara eksternal dari piranti masukan, misalnya dari papan ketik, dari sebuah berkas (*file*), dari tetikus (*mouse*), dan sebagainya. Memasukkan nilai dari dari piranti masukan dinamakan operasi *pembacaan* data, karena komputer seolah-olah membaca apa yang kita masukkan.

Notasi algoritmik untuk pembacaan nilai dari piranti masukan:

```
read (nama 1, nama 2, ..., nama n)
```

dengan syarat bahwa $nama_1$, $nama_2$, ..., $nama_n$ adalah nama-nama peubah yang sudah dideklarasikan tipenya.

Misalkan M, a1, a2, a3, bertipe *real*, nama_kota dan nama_mhs bertipe *string*, NRP dan nilai bertipe *integer*, dan P adalah rekaman bertipe Titik. Cara membaca nilai untuk semua peubah tersebut adalah:

```
DEKLARASI
   M, a1, a2, a3 : real
   nama_mhs      : string
```

```
   NIM            : integer
   type Titik     : record <x : real, y : real>
   P : Titik

ALGORITMA:
   read(M)
   read(a1, a2, a3)
   read(nama_mhs, NIM)
   read(P.x, P.y)
```

Ketika pernyatan

```
read(M)
```

dieksekusi oleh komputer, komputer membaca nilai `M` yang dimasukkan dari luar (misalnya dari papan ketik). Jika nilai yang diketikkan adalah `78`, maka lokasi memori yang bernama `M` sekarang berisi `78`.

Begitu juga bila pernyataan

```
read(nama_mhs, NIM)
```

dieksekusi, komputer meminta pengguna mengetikkan nilai untuk `nama_mhs`, `NRP`, dan `nilai`. Bila pengguna mengetikkan

4.5.2 Ekspresi

Suatu nilai dipakai untuk proses transformasi menjadi keluaran yang diinginkan. Transformasi nilai menjadi keluaran dilakukan melalui suatu perhitungan (komputasi). Cara perhitungan itu dinyatakan dalam suatu ekspresi. Ekspresi terdiri atas *operand* dan *operator*. *Operand* adalah nilai yang dioperasikan dengan *operator* tertentu. *Operand* dapat berupa konstanta, nama peubah, nama konstanta, atau hasil dari suatu fungsi. Hasil evaluasi dari sebuah ekspresi adalah nilai di dalam ranah yang sesuai dengan tipe *operand* yang dipakai. Dikenal tiga macam ekspresi: eskpresi aritmetik, ekspresi relasional, dan ekspresi *string*.

4.5.2.1 Ekspresi Aritmetik

Ekspresi numerik adalah ekspresi yang baik *operand*-nya bertipe numerik dan hasilnya juga bertipe numerik. Misalkan dideklarasikan nama-nama peubah sebagai berikut:

```
DEKLARASI
   a, b, c : real
   d       : integer
   i, j, k : integer
```

Contoh ekspresi numerik misalnya,

```
a * c
```

Hasil evaluasi ekspresi tersebut juga bertipe *real*. Pada ekspresi ini, *operand*-nya adalah a dan b, sedangkan operatornya adalah "*". Bila hasil evaluasi disimpan ke dalam peubah, maka peubah tersebut haruslah bertipe sama dengan tipe hasil. Pengisian hasil evaluasi ekspresi a*c ke dalam nama peubah c melalui pernyataan

```
c ← a * b
```

adalah benar, tetapi pernyataan

```
d ← a * b
```

tidak benar karena d bertipe *integer* sedangkan hasil evaluasi ekspresi bertipe *real*.

Contoh lainnya adalah penyimpanan eskpresi (i+j) div 2 ke dalam peubah k:

```
k ← (i + j) div 2
```

Contoh ekspresi numerik lainnya:

```
a/2.5) + c *b
i+(j*k) - 10*(d+k)
100.0/(a+c) - 2.5
```

Yang harus diperhatikan dalam penulisan ekspresi numerik adalah tingkatan (hirarkhi) operator. Operator yang mempunyai tingkatan lebih tinggi lebih dahulu dikerjakan daripada operator yang tingkatannya lebih rendah. Misalnya ekspresi

```
a/c + b
```

akan dikerjakan dengan urutan a/c lebih dahulu, lalu hasilnya ditambah dengan b.

Urutan evaluasi ekspresi dapat berubah karena penggunaan tanda kurung. Misalnya eskpresi

```
a/(c + b)
```

akan dikerjakan sebagai c + b lebih dahulu, baru kemudian a dibagi dengan hasil dari c + b.

Tingkatan operator aritmetika adalah (dari tertinggi ke terendah):

```
(i)     div, mod
(ii)    /
(iii)   *
(iv))   +, -
```

Operator "/" dan "*" dapat saja mempunyai urutan prioritas yang sama, misalnya

```
a*b/c
```

sama saja dengan

```
(a*b)/c
```

atau

```
a*(b/c)
```

Tetapi, `a/b*c` diinterpretasikan sebagai `(a/b)*c`, hasilnya tidak sama dengan `a/(b*c)`. Penggunaan tanda kurung dapat menghilangkan kesalahan interpretasi.

Operator +, - mempunyai urutan perioritas yang sama. Apabila ada bagian ekspresi yang diapit oleh pasangan tanda kurung, maka ekspresi di dalam tanda kurung mempunyai prioritas tertinggi untuk dievaluasi terlebih dahulu.

Contoh-contoh ekspresi yang sudah kita kemukakan merupakan eskpresi *biner*, yaitu ekspresi yang *operator*-nya membutuhkan dua buah *operand* (sehingga operator-nya disebut juga *operator* biner). Di samping eskpresi biner, terdapat juga ekspresi *uner*, yaitu ekspresi dengan satu buah *operand*, misalnya

```
-a
-a * (b+c)
```

(`-a` adalah ekspresi uner). Operator "-", selain merupakan operator biner, juga adalah satu-satunya operator uner dari seluruh operator arimetika.

Ekpresi biner pada seluruh contoh di atas ditulis dalam notasi *infix*, yaitu notasi yang kita gunakan dalam kehidupan sehari-hari. Notasi *infix* adalah notasi dengan susunan:

operand1 operator operand2

Selain notasi *infix*, ekspresi biner juga dapat ditulis dalam notasi *prefix*:

operator operand1 operand2

atau dalam notasi *postfix* (*suffix* atau *polish*):

operand1 operand2 operator

Contoh ekspresi *infix*, *prefix*, dan *postfix*:

infix :
```
6*8
a+b/c*d-e*f
```
prefix :
```
*68
*+a/bc-d*ef
```
postfix :
```
68*
abc/+def*-*
```

Umumnya bahasa pemrograman mengevaluasi ekspresi dalam notasi *infix*, namun ada beberapa bahasa pemrograman tertentu yang dapat menghitung ekspresi dalam notasi *postfix* dan *prefix*.

Rumus-rumus di bawah ini dalam notasi algoritmik:

$$T = \frac{5}{9}(C + 32) \quad ; \quad z = \frac{2x + y}{5w} \quad ; \quad y = 5(\frac{a+b}{cd} + \frac{m}{p+q})$$

```
T ← 5/9*(C + 32)
z ← (2*x + y)/(5*w)
y ← 5((a + b)/(c*d) + m/(p + q))
```

4.5.2.2 Ekspresi Relasional

Ekspresi relasional adalah ekspresi dengan operator <, ≤, >, ≥, =, dan ≠, not, and, or, dan xor. Hasil evaluasi ekspresinya adalah nilai bertipe *boolean* (*true* atau *false*), sehingga ekspresi relasional kadang-kadang disebut juga ekspresi *boolean*. Misalkan dideklarasikan nama-nama peubah sebagai berikut ini:

```
DEKLARASI
   ada, ketemu, besar  : boolean
   x, y : integer
```

Misalkan `ketemu` berisi nilai `false`, ada bernilai `true`, `x` bernilai `8` dan `y` bernilai `12`, maka. Contoh-contoh ekspresi *boolean* adalah:

```
not ada          (hasil: false)
ada or ketemu    (hasil: true)
ada and true     (hasil: true)
x < 5            (hasil: false)
ada or (x = y)   (hasil: true)
```

4.5.2.3 Ekspresi *String*

Ekspresi *string* adalah ekspresi dengan operator "+" (operator penyambungan /*concatenation*). Misalkan dideklarasikan nama-nama peubah sebagai berikut ini:

```
DEKLARASI
     Kar  : char
     s    : string
```

maka contoh-contoh ekspresi *string* misalnya:

```
s + kar) + 'c'              { operasi penyambungan karakter/string }
'Jl Ganesha ' + 'No 10'
```

4.5.3 Menuliskan Nilai ke Piranti Keluaran

Nilai konstanta, peubah, dan hasil ekspresi dapat ditampilkan ke piranti keluaran (umumnya layar peraga). Instruksi penulisan nilai dilakukan dengan notasi write:

```
write(nama 1, nama 2, ..., nama n)
write(konstanta)
write(ekspresi)
write(nama, konstanta, ekspresi)
```

$nama_1$, $nama_2$, ..., $nama_n$ dapat berupa nama peubah atau nama konstanta. Menampilkan nilai ke piranti keluaran diistilahkan dengan mencetak nilai tersebut.

Contoh:

```
write('A')      { menulis karakter 'A' }
write(A)        { menulis nilai yang disimpan peubah A}
write(A + 2)    { menulis ekspresi, yaitu nilai A ditambah 2}
write(A, 'dikali 2 = ', A*2)  { menulis nilai A, string 'dikali 2 = ',
                                dan nilai dari ekspresi A * 2}
```

Misalkan nama_mhs, nrp, nilai dan J dideklarasikan tipenya di dalam bagian DEKLARASI. Algoritma untuk mencetak nilai yang disimpan oleh nama-nama tersebut adalah seperti di bawah ini:

```
DEKLARASI
    nrp       : integer
    nama_mhs  : string
    nilai     : real
    type Jam  : record
                  < hh:integer, {0..23}
                    mm:integer, {0..59}
                    ss:integer  {0..59}
                  >
    J         : Jam

ALGORITMA:
    nama_mhs ← 'Achmad Baiquni'
    nrp ← 13290056
    nilai ← 90.8
    J.hh ← 6
    J.mm ← 12
    J.ss ← 45

    write('Jurusan Teknik Informatika ITB')
    write(nama_mhs, nrp)
    write('Nilai = ', nilai)
    write('Dihitung pada pukul:', J.hh,':',J.mm,':',J.ss)
```

Keluaran yang ditampilkan ke piranti keluaran adalah seperti ini:

```
Jurusan Teknik Informatika ITB
Ahmad Baiquni      10290056
Nilai = 90.8
Dihitung pada pukul: 6:12:45
```

Perhatikanlah bahwa `write(A)` tidak sama dengan `write('A')`. Yang pertama menuliskan nilai A, sedangkan yang kedua menuliskan karakter 'A'.

4.6 Tabel Translasi Notasi Algoritmik Pengisian Nilai, Pembacaan, dan Penulisan ke dalam Notasi Bahasa Pascal dan C

Pernyataan	Algoritmik	*Bahasa Pascal*	*Bahasa C*
1. Penugasan	←	:=	=
2. Pembacaan	read	read readln	scanf
3. Penulisan	write	write writeln	printf

Keterangan:

`read` : membaca masukan dari papan kunci, kursor tetap di baris yang sama.

`readln` : membaca masukan dari papan kunci, kursor kemudian pindah ke baris berikutnya.

`write` : menulis keluaran ke layar, kursor tetap di baris yang sama.

`writeln` : menulis keluaran ke layar, kursor kemudian pindah ke baris berikutnya.

`scanf` : membaca dengan menggunakan penentu format.

`printf` : mencetak nilai dengan penentu format.

Tabel penentu format dalam bahasa *C*:

Tipe data	Penentu format
`int`	`%d`
`unsigned int`	`%u`
`long int`	`%ld`
`long unsigend int`	`%lu`
`float`	`%f`
`double`	`%lf`
`char`	`%c`
`char[n] (string)`	`%s`

Di dalam bahasa *C*, pengisian konstanta *string* secara langsung ke dalam peubah tidak dapat dilakukan. Sebagai contoh, jika dideklarasikan *s* sebagai peubah *string*:

```
char s[15];
```

maka, pernyataan

```
s = "ini string";
```

adalah salah. Cara mengisi konstanta *string* ke dalam peubah adalah dengan perantara fungsi strcpy (*string copy*) sebagai berikut:

```
strcpy(s, "ini string");
```

Sebaliknya, bahasa *Pascal* membolehkan pengisian konstanta *string* secara langsung ke dalam peubah. Sebagai contoh, jika dideklarasikan *s* sebagai peubah *string*:

```
var s : string[15];
```

maka, pernyataan

```
s := 'ini string';
```

adalah benar.

Contoh-contoh translasi:

ALGORITMIK:

```
DEKLARASI
   type Titik : record <x:real, y:real>
   P    : Titik
   a, b : integer
```

```
   NamaArsip, h  : string
   nilai : real
   C  : char

ALGORITMA:
   nilai←1200.0
   read(P.x, P.y)
   read(NamaArsip)
   h ← NamaArsip
   read(a, b)
   read(C)

   write('Nama arsip: ',NamaArsip)
   write('Koordinat titik adalah: ',P.x, ',', P.y)
   write(b, nilai)
   write('Karakter yang dibaca adalah ', C)
```

PASCAL:

```
(*DEKLARASI*)
   type
      Titik = record
                 x: real;
                 y: real;
              end;
   var
      P : Titik;
      a, b : integer;
      NamaArsip, h : string[12];
      nilai : real;
      C  : char;

(* ALGORITMA *)
begin
   nilai:=1200.0;
   readln(P.x, P.y);
   readln(NamaArsip);
   h : = NamaArsip;
   readln(a, b);
   readln(C);
   writeln('Nama arsip: ',NamaArsip);
   writeln('Koordinat titik adalah ',P.x, ',', P.y);
   writeln(b, nilai);
   writeln('Karakter yang dibaca adalah ', C);
end.
```

C:

```
/*DEKLARASI*/
   typedef struct{
             float x;
             float y;
           }Titik;

   Titik P;
   int a, b;
```

```
  char NamaArsip[12], h[12];
  float nilai;
  char C;

/* ALGORITMA */
main()
{
  nilai = 1200.0;
  scanf("%d %d", &P.x, &P.y);
  scanf("%s", NamaArsip);
  strcpy(h, NamaArsip);
  scanf("%d %d", &a, &b);
  scanf("%c",&C);
  scanf("Nama arsip: %s \n", NamaArsip);
  printf("Koordinat titik adalah %d %c %d \n",P.x, ',', P.y);
  printf("%d %f \n", b, nilai);
  printf("Karakter yang dibaca adalah %c \n", C);
}
```

Keterangan:

1. Perintah pembacaan scanf untuk nama peubah yang bertipe *string* tidak perlu lagi diberi awalan "&" karena *string* di dalam bahasa *C* sebenarnya adalah larik (*array*) (baca buku acuan tentang larik dalam bahasa *C*). Nama peubah yang tidak diikuti dengan indeks menunjukkan alamat dari elemen pertama larik tersebut.
2. `scanf` tidak dapat membaca *string* yang mengandung spasi, sebab `scanf` mengabaikan karakter-karakter setelah spasi. Untuk itu, untuk membaca *string* yang mengandung spasi, gunakan perintah `gets` (lihat pejelasan di bawah).
3. Simbol `\n` menyatakan bahwa setelah pencetakan, kursor pindah ke baris di bawahnya. Jadi, efeknya sama dengan `writeln` di *Pascal*.

Terdapat beberapa perintah lain di dalam bahasa *C* untuk membaca masukan dari papan ketik, dan perintah untuk mencetak *string*, yaitu:

(a) *getche()* : membaca data karakter diikuti penekanan tombol **Enter**; karakter yang dibaca ditampilkan (*echo*) ke layar;
include : `#include <conio.h>`
contoh : `x = getche();`

(b) *getch()* : membaca data karakter tanpa penekanan tombol **Enter**, karakter yang dibaca tidak ditampilkan ke layar)
include : `#include <conio.h>`
contoh : `x = getch();`

(c) *gets()* : membaca *string* dari papan ketik
include : `#include <stdio.h>`
contoh : `gets(NamaArsip)`

(d) *puts()* : mencetak *string* ke layar
include : `#include <stdio.h>`
contoh : `puts(NamaPegawai)`

Soal Latihan Bab 4

1. Definisikan sebuah tipe terstruktur untuk menyatakan data nasabah di sebuah bank. Data nasabah terdiri atas *field*: nomor *account*, nama nasabah, alamat nasabah, kota nasabah, nomor telepon nasabah. Untuk setiap *field*, definisikan tipe data yang cocok.
2. Definisikan sebuah tipe terstruktur untuk menyatakan data penerbangan di sebuah bandara. Data penerbangan terdiri atas: nomor penerbangan (misal GA101), bandara (kota) asal, bandara tujuan, tanggal keberangkatan, jam keberangkatan (*departure time*), jam datang (*arrival time*). Untuk setiap *field*, definisikan tipe data yang cocok.
3. Tuliskan rumus berikut dalam notasi algoritmik:

   (a) $V = \frac{4}{3}\pi r^3$

   (b) $x = \frac{-b + 2c^2 + 4ab}{2c}$

   (c) $m = \frac{a-b}{3ac}(1 - \frac{b}{cd})$

4. Apa perbedaan n ← n + 2 dari sudut pandang algoritma dan $n = n + 2$ dari sudut pandang matematika?
5. Sebuah mobil melaju dengan kecepatan tetap v km/jam. Jika mobil tersebut berjalan selama t jam, tulislah algoritma untuk menghitung jarak yang sudah ditempuh mobil (dalam km). Algoritma tersebut membaca masukan berupa v dan t, menghitung *jaeak* dengan rumus $s = vt$, lalu mencetak jarak tersebut.
6. Translasikan algoritma pada soal nomor 5 ke dalam program dalam bahasa *Pascal* dan *C*, lalu tes program dengan bermacam-macam nilai v dan t.

5
Runtunan

Melangkah anak tangga satu per satu secara sekuensial

Runtunan adalah struktur algoritma paling dasar yang berisi rangkaian instruksi yang diproses secara sekuensial, satu per satu, mulai dari instruksi pertama sampai instruksi terakhir. Bab 5 ini membahas runtunan dan contoh-contoh algoritma yang hanya berisi runtunan.

5.1 Pendahuluan

Algoritma merupakan runtunan (*sequence*) satu atau lebih instruksi, yang berarti bahwa [GOL88]:

1. Tiap instruksi dikerjakan satu per satu;
2. Tiap instruksi dilaksanakan tepat sekali; tidak ada instruksi yang diulang;
3. Urutan instruksi yang dilaksanakan pemroses sama dengan urutan instruksi sebagaimana yang tertulis di dalam teks algoritmanya;
4. Akhir dari instruksi terakhir merupakan akhir algoritma.

Setiap instruksi di dalam runtunan ditulis dalam satu baris, atau beberapa instruksi dalam baris yang sama tetapi antara setiap instruksii dipisahkan dengan tanda titik koma (;). Contoh-contoh masalah di bawah ini memperlihatkan algoritma yang hanya berisi runtunan.

5.2 Contoh-contoh Runtunan

Contoh 5.1. Bagaimana mencetak pesan "*Hello, world*"ke layar? (Catatan: contoh pertama ini sudah pernah diberikan sebagai ilustrasi di dalam Bab 3).

Penyelesaian

Algoritma untuk mencetak "*Hello, world*":

```
PROGRAM Hello_World
{ Program untuk mencetak "Hello, world" }

DEKLARASI
   { tidak ada }

ALGORITMA:
   write('Hello, world')
```

Algoritma 5.1 Mencetak "Hello, world"

Runtunan di dalam Algoritma 5.1 hanya berisi satu pernyataan saja, yaitu write('hello, world'). Kita dapat membuat beragam versi program Hello_World, tidak hanya satu baris instruksi seperti algoritma di atas. Beberapa versi algoritma untuk mencetak "*Hello, world*" dituliskan di bawah ini:

Versi 2:
String "*Hello, world* " disimpan di dalam sebuah peubah yang bertipe *string*, baru kemudian dicetak.

```
PROGRAM Hello_World2
{ Program untuk mencetak "Hello, world" }

DEKLARASI
   pesan : string

ALGORITMA:
   Pesan ← 'Hello, world'
   write(pesan)
```

Versi 3:
String "*Hello, world*" disimpan sebagai konstanta., baru kemudian dicetak.

```
PROGRAM Hello_World3
{ Program untuk mencetak "Hello, world" }

DEKLARASI
   const pesan = 'Hello, world'
```

```
ALGORITMA:
   write(pesan)
```

Contoh 5.2. Tulislah algoritma yang membaca nama seseorang dari papan ketik, lalu menampilkan ucapan "Halo" diikuti dengan nama orang tersebut.

Penyelesaian

```
PROGRAM Halo_Nama
{ Mencetak string 'Halo ' dan diikuti dengan nama orang. Nama orang
  dibaca dari papan ketik }

DEKLARASI
   nama : string

ALGORITMA:
   read(nama)
   write('Halo', nama)
```

Algoritma 5.2 Mencetak ucapan "Halo" dan nama

Contoh 5.3. Tuliskan algoritma yang membaca dua buah nilai-nilai untuk peubah A dan B, lalu mempertukarkan nilai kedua peubah tersebut. Misalnya, sebelum pertukaran nilai A = 8, nilai B = 5, maka setelah pertukaran, nilai A = 5 dan nilai B = 8. (Catatan: contoh ketiga ini sudah pernah diberikan sebagai ilustrasi di dalam Bab 2).

Penyelesaian

Algoritma pertukaran dua buah nilai sama seperti algoritma mempertukarkan isi dari dua buah ember. Kita membutuhkan sebuah peubah bantu sebagai tempat tempat penampungan sementara.

```
PROGRAM Pertukaran
{ Mempertukarkan nilai A dan B. Nilai A dan B dibaca terlebih dahulu.
}

DEKLARASI
   A, B, C : integer

ALGORITMA:
   { asumsikan A dan B sudah terdefinisi dengan nilai, misalnya
     melalui pengisian langsung atau dibaca nilainya dari papan ketik }

  { pertukarkan nilai A dan B }
   C ← A    {simpan  nilai A di tempat penampungan sementara, C}
   A ← B    {sekarang A dapat diisi dengan nilai B}
   B ← C    {isi B dengan nilai A semula yang tadi disimpan di C}
  {Tulis nilai A dan B setelah pertukaran, jika diperlukan }
```

Algoritma 5.3 Mempertukarkan nilai dari dua buah peubah

Algoritma `Pertukaran` ke dalam program *Pascal* dan *C* adalah seperti di bawah ini:

PASCAL:

```
program Pertukaran;
{ Mempertukarkan nilai A dan B. Nilai A dan B dibaca terlebih dahulu. }

(* DEKLARASI *)
var
   A, B, C : integer;

(* ALGORITMA: *)
begin
   { baca nilai A dan B, misalnya dengan isntruksi:
     write('A = ?'); readln(A);
     write('B = ?'); readln(B);
   }

   { pertukarkan nilai A dan B }
   C := A;   {simpan  nilai A di tempat penampungan sementara, temp}
   A := B;   {sekarang A dapat diisi dengan nilai B}
   B := C;   {isi B dengan nilai A semula yang tadi disimpan di temp}

   { Tulis nilai A dan B setelah pertukaran, jika diperlukan,
     misalnya:}
     writeln('A = ', A);
     writeln('B = ', B);
   }
end.
```

C:

```
/* PROGRAM Pertukaran */
/* Mempertukarkan nilai A dan B. Nilai A dan B dibaca terlebih dahulu.
*/

#include <stdio.h>

main()
{
/* DEKLARASI */
   int A, B, C;

/* ALGORITMA: */
   /* baca nilai A dan B, misalnya dengan isntruksi:
      printf("A = ?"); scanf("%d", &A);
      printf("B = ?"); scanf("%d", &B);
  */
   /* proses pertukaran */
   C = A;    /*simpan  nilai A di tempat penampungan sementara, C */
   A = B;    /*sekarang A dapat diisi dengan nilai B */
   B = C;    /*isi B dengan nilai A semula yang tadi disimpan di C */

   /* Tulis nilai A dan B setelah pertukaran, jika diperlukan: */
        printf("A = %d \n", A);
        printf("B = %d \n", B);
   */
}
```

Contoh 5.4. Misalkan *A* dan *B* adalah dua buah peubah yang bertipe *integer*. Apa yang dilakukan oleh potongan algoritma di bawah ini?

```
ALGORITMA:
   ...
   A ← A + B
   B ← A - B
   A ← A - B
   ...
```

Algoritma 5.4 Apa yang dilakukan algoritma ini?

Penyelesaian

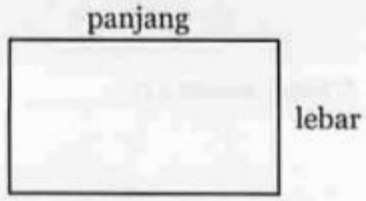
Coba lakukan tes untuk $A = 10$ dan $B = 6$. Anda akan mendapatkan hasil setelah pelaksanaan runtunan tersebut bahwa $A = 6$ dan $B = 10$. Jadi, potongan algoritma di atas adalah cara lain untuk mempertukarkan nilai dari dua buah peubah yang bertipe bilangan bulat *tanpa* menggunakan peubah bantu (bandingkan dengan Algoritma 5.3 yang menggunakan peubah bantu, *temp*). Namun, Algoritma 5.3 dapat digunakan untuk mempertukarkan nilai dari dua peubah yang bertipe sembarang (asalkan keduanya bertipe sama), sedangkan Algoritma 5.4 hanya dapat digunakan untuk mempertukarkan nilai dari dua peubah *integer* saja.

Algoritma pertukaran nilai dari dua buah peubah *integer* tanpa peubah bantu selengkapnya sebagai berikut:

```
PROGRAM Tukar
{ Mempertukarkan nilai A dan B yang bertipe bilangan bulat tanpa peubah
bantu. Nilai A dan B dibaca terlebih dahulu dari piranti masukan. }

DEKLARASI
   A     : integer    { nilai pertama }
   B     : integer    { nilai kedua }
   temp  : integer    { peubah bantu }

ALGORITMA:
   read(A,B)    { baca nilai A dan B terlebih dahulu }

   write(A,B)   { cetak nilai A dan B sebelum pertukaran }

   { proses pertukaran }
   A ← A + B
   B ← A - B
   A ← A - B

   write(A,B) { cetak nilai A dan B setelah pertukaran }
```

Algoritma 5.5 Mempertukarkan nilai dari dua buah peubah integer tanpa peubah bantu

Contoh 5.5. Tulis algoritma masing-masing dalam notasi algoritmik, *Pascal*, dan *C*, yang membaca panjang dan lebar dari sebuah empat persegi panjang, lalu menghitung luas segiempat yang berbentuk empat persegi panjang (Gambar 5.1). Luas segiempat adalah panjang dikali lebar. Luas segiempat dicetak ke piranti keluaran. Tuliskan!



Gambar 5.1 Luas segiempat yang berbentuk empat persegi panjang adalah panjang × lebar.

Penyelesaian

ALGORITMIK:

```
PROGRAM Luas_Empat_Persegi_Panjang
{ Membaca panjang dan lebar segiempat yang berbentuk empat persegi panjang,
menghitung luasnya, lalu mencetak luas tersebut ke piranti keluaran. }

DEKLARASI
  panjang : real        { panjang segiempat, dalam satuan cm }
  lebar   : real        { lebar segiempat, dalam satuan cm }
  luas    : real        { luas segiempat, dalam satuan cm² }

ALGORITMA:
  read(panjang, lebar)
  luas ← panjang * lebar
  write(luas)
```

Algoritma 5.6 Menghitung luas empat persegi panjang

PASCAL:

```
program Luas_Empat_Persegi_Panjang;
{ Membaca panjang dan lebar segiempat yang berbentuk empat persegi panjang,
menghitung luasnya, lalu mencetak luas tesrebut ke piranti keluaran. }

(* DEKLARASI *)
var
  panjang  : real;        { panjang segiempat, dalam satuan cm }
  lebar    : real;        { lebar segiempat, dalam satuan cm }
  luas     : real;        { luas segiempat, dalam satuan cm² }

(* ALGORITMA: *)
begin
  write('Berapa panjang segiempat? '); readln(panjang);
  write('Berapa lebar segiempat? '); readln(lebar);
  luas := panjang * lebar;
```

```
   writeln('Luas segiempat = ', luas);
end.
```

C:

```
/* PROGRAM LUAS SEGIEMPAT
/*Membaca panjang dan lebar segiempat yang berbentuk empat persegi panjang,
menghitung luasnya, lalu mencetak luas tesrebut ke piranti keluaran. */

#include <stdio.h>

main()
{
   /* DEKLARASI */
   float panjang;     /* panjang segiempat, dalam satuan cm */
   float lebar;       /* lebar segiempat, dalam satuan cm */
   float luas;        /* luas segiempat, dalam satuan cm^2 */

   /* ALGORITMA: */
   printf("Berapa panjang segiempat? "); scanf("%f", &panjang);
   printf("Berapa lebar segiempat? "); scanf("%f", &lebar);
   luas = panjang * lebar;
   printf("Luas segiempat = %f \n", luas);
}
```

Contoh 5.6. Tulislah algoritma untuk menghitung komisi yang diterima *salesman* berdasarkan nilai penjualan yang dicapainya. *Salesman* itu mendapat komisi 5% dari hasil penjualannya. Algoritma menerima data nama *salesman* dan nilai penjualan yang dicapainya, menghitung komisi, lalu mencetak nama *salesman*, dan besar komisi yang diperolehnya.

Penyelesaian

```
 PROGRAM Komisi_Salesman
{ Menghitung besar komisi yang diterima seorang salesman. Besar komisi
  adalah 5% dari nilai penjualan yang dicapainya. Data masukan adalah
  nama salesman dan nilai penjualannya. Keluaran algoritma adalah
besarkomisi yang diterima salesman tersebut.}

DEKLARASI
   NamaSalesman   : string
   NilaiPenjualan : real   { nilai penjualan yang dicapai, dalam Rp }
   komisi         : real   { besar komisi, dalam Rp }
ALGORITMA:
   read(NamaSalesman, NilaiPenjualan)
   komisi ← 0.05 * NilaiPenjualan
   write(komisi)
```

Algoritma 5.7 Menghitung komisi salesman

Contoh 5.7. Tulislah algoritma yang membaca nama karyawan dan gaji pokok bulanannya dan menghitung gaji bersih karyawan tersebut. Gaji bersih yang diterima pegawai adalah:

```
Gaji bersih = gaji pokok + tunjangan - pajak
```

Tunjangan karyawan dihitung 20% dari gaji pokok, sedangkan pajak adalah 15% dari gaji pokok ditambah tunjangan. Nama karyawan dan gaji bersihnya dicetak ke piranti keluaran.

Penyelesaian

```
PROGRAM Gaji_Bersih_Karyawan
{ Menghitung gaji bersih karyawan. Data masukan adalah nama karyawan dan
gaji pokok bulanannya. Gaji bersih = gaji pokok + tunjangan - pajak.
  Tunjangan adalah 20% dari gaji pokok, sedangkan pajak adalah 15% dari
gaji pokok. Keluaran adalah nama karyawan dan gaji bersihnya
}

DEKLARASI
   const PersenTunjangan = 0.2    { persentase tunjangan gaji }
   const PersenPajak = 0.15       { persentase potongan pajak }
   NamaKaryawan    : string
   GajiPokok,tunjangan,pajak,GajiBersih : real

ALGORITMA:
   read(NamaKaryawan, GajiPokok)
   tunjangan ← PersenTunjangan * GajiPokok
   pajak ← PersenPajak * (GajiPokok + tunjangan)
   GajiBersih ← GajiPokok + tunjangan - pajak
   write(NamaKaryawan, GajiBersih)
```

Algoritma 5.8 Menghitung gaji bersih seorang karyawan

Contoh 5.8. Tulislah algoritma yang membaca dua buah titik $P1 = (x_1, y_1)$ dan $P2 = (x_2, y_2)$, lalu menghitung titik tengah dari $P1$ dan $P2$.

Titik tengah dari dua $P1$ dan $P2$ adalah $P3 = (x_3, y_3)$ yang dihitung dengan rumus:

$$x_3 = \frac{x_1 + x_2}{2} \quad \text{dan} \quad y_3 = \frac{y_1 + y_2}{2}$$

Sebagai contoh, jika $P1 = (4, 8)$ dan $P2 = (2, 6)$, maka $P3(3, 7)$.

Penyelesaian

```
PROGRAM Titik_Tengah
{ Menghitung titik-tengah dari dua buah titik P1 = (x1,y1) dan P2 =
(x2,y2). Titik tengah dituliskan ke piranti keluaran }

DEKLARASI
   type Titik : record <x:real,
                        y:real
```

```
  P1, P2, P3 : Titik

ALGORITMA:
  read(P1.x,P1.y)              { baca titik P1 }
  read(P2.x,P2.y)              { baca titik P2 }
  P3.x ← (P1.x + P2.x)/2
  P3.y ← (P1.y + P2.y)/2
  write(P3.x,P3.y)
```

Algoritma 5.9 Menghitung titik tengah dari dua buah titik.

Contoh 5.9. Seorang pelari maraton menempuh waktu lari yang diukur dalam satuan jam, menit, dan detik. Tulislah algoritma yang membaca waktu tempuh seorang pelari maraton, lalu mengonversi waktu tempuh tersebut ke dalam detik. Ingatlah bahwa

1 menit = 60 detik
1 jam = 3600 detik

Misalnya, jika waktu tempuh seorang pelari maraton adalah 1 jam, 5 menit, 40 detik. Dalam detik, waktu tempuh seluruhnya adalah (1 × 3600) + (5 × 60) + + 40 = 3940 detik.

Penyelesaian

```
PROGRAM Konversi_ke_detik
{ Membaca waktu tempuh pelari maraton dalam jam, menit, detik, lalu
mengkonversi waktu tempuh tersebut ke dalam detik. Hasil konversi
ditampilkan ke piranti keluaran }

DEKLARASI
  type Jam = record
             <
               hh : integer   {jam}
               mm : integer   {menit}
               ss : integer   {detik}
             >
  J : Jam
  TotalDetik : integer

ALGORITMA:
  read(jam, menit, detik)
  TotalDetik ← (J.hh*3600) + (J.mm*60) + J.ss
  write(TotalDetik)
```

Hati-hati jika Anda mentranslasi algoritma `Konversi_ke_detik` ke dalam bahasa *Pascal* atau *C*, sebab Anda harus memperhatikan tipe bilangan bulat yang digunakan. Karena ranah nilai tipe *integer* terbatas, maka ada kemungkinan hasil konversi jam-menit-detik ke total detik bernilai negatif, sebab nilai (J.hh*3600) + (J.mm*60) + J.ss berada di luar rentang tipe *integer*. Tipe *longint* (*long integer*) di dalam bahasa *Pascal* dan bahasa *C* mempunyai

ranah yang lebih besar sehingga dapat dipakai untuk masalah ini. Jadi, program `Konversi_ke_detik` dalam bahasa *Pascal* dan *C* sebagai berikut:

PASCAL:

```
program Konversi_ke_detik;
{ Membaca waktu tempuh pelari maraton dalam jam, menit, detik, lalu
mengkonversi waktu tempuh tersebut ke dalam detik. Hasil konversi
ditampilkan ke piranti keluaran }

(* DEKLARASI *)
type Jam = record
             hh:longint;    {jam}
             mm:longint;    {menit}
             ss:longint;    {detik}
           end;

var
  J : Jam;
  TotalDetik : longint;

(* ALGORITMA: *)
begin
   write('Jam  :'); readln(J.hh);
   write('Menit:'); readln(J.mm);
   write('Detik:'); readln(J.ss);
   TotalDetik:= (J.hh*3600) + (J.mm*60) + J.ss;
   writeln('Total detik = ', TotalDetik);
end.
```

C:

```
/* PROGRAM Konversi_ke_detik */
/* Membaca waktu tempuh pelari maraton dalam jam, menit, detik, lalu
mengkonversi waktu tempuh tersebut ke dalam detik. Hasil konversi
ditampilkan ke piranti keluaran. */

#include <stdio.h>

main()
{
  /* DEKLARASI */
  typedef struct {long int hh;   /*jam*/
                  long int mm;   /*menit*/
                  long int ss;   /*detik*/
                 }Jam;
  Jam J;
  long int total_detik;

  /* ALGORITMA: */
  printf("Jam  :"); scanf("%ld",&J.hh);
  printf("Menit:"); scanf("%ld",&J.mm);
  printf("Detik:"); scanf("%ld",&J.ss);
  TotalDetik = (J.hh*3600) + (J.mm*60) + J.ss;
  printf("Total detik = %ld", TotalDetik);
}
```

Contoh 5.10. Tulislah algoritma yang membaca lama sebuah percakapan telepon dalam satuan detik, lalu mengonversinya ke dalam jam, menit, dan detik. Sebagai contoh, misalkan lama percakapan adalah 4000 detik, maka 4000 detik = 1 jam + 6 menit + 40 detik, ini diperoleh dengan cara:

```
4000    div     3600    = 1(jam)
4000    mod     3600    = 400(sisa detik)
400     div     60      = 6(menit)
400     mod     60      = 40(detik)
```

Masalah pada contoh 5.10 ini merupakan kebalikan masalah pada Contoh 5.9.

Penyelesaian

```
PROGRAM Konversi_detik_ke_JamMenitDetik
/* Membaca lama percakapan telepon dalam detik, lalu mengkonversinya ke
dalam jam-menit-detik. Hasil konversi ditampilkan ke piranti keluaran. */

DEKLARASI
   type Jam : record <hh:integer , {0..23}
                      mm:integer , {0..59}
                      ss:integer   {0..59}
                     >
   J           : Jam
   TotalDetik  : integer
   sisa        : integer    { peubah pembantu }

ALGORITMA:
   read(TotalDetik)
   J.hh ← total_detik div 3600      { mendapatkan jam }
   Sisa ← TotalDetik mod 3600
   J.mm ← sisa div 60               { mendapatkan menit }
   J.ss ← sisa mod 60               { mendapatkan detik }
   write(J.hh,J.mm,J.ss)
```

Algoritma 5.10 Mengonversi detik ke jam-menit-detik

Banyak masalah dalam kehidupan sehari-hari yang memiliki kemiripan dengan konversi jumlah detik ke jam-menit-detik. Misalnya mengonversi panjang dalam satuan cm ke kilometer-meter-sentimeter dan mengonversi jumlah hari dalam tahun-bulan-hari (lihat soal Latihan Bab 5).

Contoh 5.11. Tulislah algoritma yang membaca dua buah jam, yang pertama waktu $J1$ (*hh*:*mm*:*ss*) dan waktu kedua $J2$ (*hh*:*mm*:*ss*) dengan syarat $J2 > J1$, lalu menghitung selisih jam (durasi) dari $J1$ sampai $J2$.

Sebagai contoh,

```
     J2          J1          J3 = J2 - J1
  12:10:56    10:08:14      02:02:42
  12:18:40    10:12:50      02:05:50
  12:10:40    10:40:55      01:29:45
```

Penyelèsaian

Dengan mengingat bahwa sebuah waktu dalam format *hh:mm:ss* dapat dipandang sebagai rentang waktu dimulai dari 0:0:0, maka kita dapat menggunakan Algoritma 5.10 untuk menghitung selisih dua buah jam. Secara garis besar, algoritma menghitung selisih dua buah jam sebagai berikut:

1. Konversi jam pertama ke dalam jumlah detik
2. Konversi jam kedua ke dalam jumlah detik
3. Hitung selisih hasil langkah 2 dengan hasil langkah 1
4. Konversi hasil langkah 3 ke jam-menit-detik.

Algoritma selengkapnya adalah:

```
PROGRAM SelisihWaktu
{ Menghitung durasi (selisih waktu) dari dua buah waktu, J1 (hh:mm:ss)
dan J2 (hh:mm:ss). Selisih waktu J2 dan J1 adalah J3 = J2 - J1. }

DEKLARASI
   type Jam : record < hh:integer , {0..23}
                       mm:integer , {0..59}
                       ss:integer   {0..59}
                     >

   J1, J2, J3   : Jam
   TotalDetik1, TotalDetik2, SelisihDetik : integer
   sisa : integer   { peubah bantu untuk menampung sisa pembagian }

ALGORITMA:
   read(J1.hh, J1.mm, J1.ss)     { jam pertama }
   read(J2.hh, J2.mm, J2.ss)     { jam kedua }

   { konversi masing-masing jam ke total detik }
   TotalDetik1 ← (J1.hh*3600) + (J1.mm*60) + J1.ss
   TotalDetik2 ← (J2.hh*3600) + (J2.mm*60) + J2.ss

   { hitung selisih total detik }
   SelisihDetik ← TotalDetik2 - TotalDetik1

   {konversi selisih detik ke dalam jam-menit-detik }
   J3.hh ← SelisihDetik div 3600     { mendapatkan jam }
   sisa ← SelisihDetik mod 3600
   J3.mm ← sisa div 60                { mendapatkan menit }
   J33.ss ← sisa mod 60               { mendapatkan detik }

   write(J3.hh, J3.mm, J3.mm)
```

Algoritma 5.11 Menghitung selisih (durasi) dari dua buah waktu

Contoh 5.12. Misalkan seorang penelepon di warung telekomunikasi memulai percakapan pada pukul *J1* dan selesai pada pukul *J2*. Andaikan 1 pulsa = 5 detik dan biaya per pulsa Rp150, tulislah algoritma untuk menghitung lama percakapan (dalam jam-menit-detik) dan biaya yang harus dibayar penelpon. Untuk

menyederhanakan masalah, andaikanlah wartel itu tutup tepat pada pukul 0:0:0 malam.

Penyelesaian

```
PROGRAM Wartel
{ Menghitung biaya percakapan di warung telekomunikasi. Masukan adalah
waktu awal dan waktu selesai percakapan (hh:mm:ss). Keluaran adalah
lama dan biaya percakapan. Satu pulsa = 5 detik dan ongkos per pulsa
adalah Rp150. }

DEKLARASI
   const BiayaPerPulsa  = 150        { biaya per pulsa }
   const LamaPulsa = 5               { 1 pulsa = 5 detik }

   type Jam : record <hh:integer, {0..23}
                      mm:integer, {0..59}
                      ss:integer  {0..59}
                     >

   J1       : Jam                   { jam awal percakapan }
   J2       : Jam                   { jam selesai percakapan }
   J3       : Jam                   { lama percakapan }
   TotalDetik1, TotalDetik2 : integer
   sisa     : integer      { peubah bantu untuk menampung sisa pembagian }
   durasi   : integer
   pulsa    : real
   biaya    : real

ALGORITMA:
   read(J1.hh, J1.mm, J1.ss)         { jam awal percakapan }
   read(J2.hh, J2.mm, J2.ss)         { jam selesai percakapan }

   { konversi masing-masing jam ke total detik }
   TotalDetik1 ← (J1.hh*3600) + (J1.mm*60) + J1.ss
   TotalDetik2 ← (J2.hh*3600) + (J2.mm*60) + J2.ss

   { hitung lama percakapan }
   durasi←TotalDetik2 - TotalDetik1

   { hitung jumlah pulsa dan biaya untuk seluruh pulsa }
   pulsa ← durasi/LamaPulsa
   biaya ← pulsa * BiayaPerPulsa

   {konversi durasi ke dalam jam-menit-detik }
   J3.hh ← durasi div 3600       { mendapatkan jam }
   sisa ← durasi mod 3600
   J3.mm ← sisa div 60           { mendapatkan menit }
   J3.ss ← sisa mod 60           { mendapatkan detik }

   Write(J3.hh, J3.mm, J3.ss, biaya)
```

Algoritma 5.12 Menghitung lama dan biaya percakapan di Wartel

5.3 Membaca/Menulis dari/ke Arsip

Contoh-contoh program *Pascal* dan *C* yang diberikan di dalam buku ini mengasumsikan bahwa data masukan dibaca dari papan ketik (*keyboard*) dan keluarannya dicetak ke layar (*monitor*). Adakalanya data masukan dibaca dari arsip (*file*) atau keluaran (*output*) program ditulis ke arsip.

Pembacaan data dari arsip bergantung pada format (cara menyimpan) data di dalam arsip. Di dalam buku ini kita mengasumsikan bahwa data masukan dibaca dari arsip dengan format teks (*text*). Antara satu data dengan data lainnya dipisahkan oleh separator spasi. Tipe peubah yang digunakan untuk menampung hasil pembacaan harus sama dengan tipe data yang dibaca; jika datanya bertipe *integer* maka peubah pembacaan juga harus bertipe *integer*; jika data riil maka peubahnya juga harus bertipe riil; begitu seterusnya.

Data masukan disimpan ke dalam arsip dengan menggunakan program pengolah kata yang menghasilkan karakter-karakter *ASCII*, seperti *Notepad* pada *Windows*, *joe* dan *vi* pada *Linux*, atau editor teks yang terintegrasi di dalam *IDE* (*Integrated Development Environment*) kompilator bahasa pemrograman (seperti *IDE* pada *Free Pascal*, lihat Bab 3).

Sebelum dibaca/ditulis, arsip harus dibuka (*open*) terlebih dahulu. Begitu pula, setelah arsip dibaca/ditulis, arsip tersebut harus ditutup (*close*).

(a) Membuka arsip (bahasa *Pascal*) untuk dibaca

Misalkan data masukan sudah disimpan di dalam arsip yang bernama data.txt. Cara membuka arsip tersebut di dalam bahasa *Pascal* adalah:

```
var
   Fin : text;     { nama peubah arsip }
begin
   assign(Fin, 'data.txt');
   reset(Fin);
   ...       { instruksi selanjutnya }
```

(b) Membuka arsip (bahasa *C*) untuk dibaca

Misalkan data masukan sudah disimpan di dalam arsip yang bernama `data.txt`. Cara membuka arsip tersebut di dalam bahasa *C* adalah:

```
main()
{
   FILE *Fin;     /* nama peubah arsip */
   Fin = fopen("data.txt", "r");  /* r adalah argumen mode, yang
                                     berarti file dibuka untuk
                                     dibaca (read) */
    ...       { instruksi selanjutnya }
}
```

Catatan:
Beberapa kompilator bahasa C membedakan antara arsip teks dan arsip biner. Untuk arsip biner ditambahkan "b" pada argumen mode (menjadi "rb" atau "r+b"), sedangkan untuk arsip teks ditambahkan "t" pada argumen mode (menjadi "rt" atau "r+t").

(c) Membuka arsip (bahasa *Pascal*) untuk ditulis
Misalkan keluaran akan disimpan di dalam arsip yang bernama `hasil.txt`. Cara membuka arsip tersebut untuk penulisan di dalam bahasa *Pascal* adalah:

```
var
   Fout : text;     { nama peubah arsip }

begin
   assign(Fout, 'hasil.txt');
   rewrite(Fout);
   ...       { instruksi selanjutnya }
```

(d) Membuka arsip (bahasa *C*) untuk ditulis
Misalkan keluaran akan disimpan di dalam arsip yang bernama `hasil.txt`. Cara membuka arsip tersebut untuk penulisan di dalam bahasa *C* adalah:

```
main()
{
   FILE *Fout;     /* nama peubah arsip */

   Fout = fopen("data.txt", "w");   /* w menyatakan bahwa file F
                                       dibuka untuk dibaca (read), t
                                       menyatakan File F adalah file
                                       text */
   ...       /* instruksi selanjutnya */
```

(e) Menutup arsip (bahasa *Pascal*)
Misalkan arsip `data.txt` sudah selesai dibaca. Cara menutup arsip tersebut di dalam bahasa *Pascal* adalah:

```
close(Fin);
```

(f) Membuka arsip (bahasa *C*)
Misalkan arsip `data.txt` sudah selesai dibaca. Cara menutup arsip tersebut di dalam bahasa *C* adalah:

```
flose(Fin);
```

(g) Pernyataan untuk membaca data dari arsip (bahasa *Pascal*)
Misalkan data masukan sudah disimpan di dalam arsip yang bernama `data.txt`. Instruksi untuk membaca sebuah data dalam bahasa *Pascal* dan menyimpan hasil pembacaan di dalam peubah *x* adalah:

```
read(Fin, x);
```

(h) Pernyataan untuk membaca data dari arsip (bahasa *C*)

Misalkan data masukan sudah disimpan di dalam arsip yang bernama `data.txt`. Bahasa *C* memiliki banyak instruksi untuk membaca data dari arsip. Salah satunya adalah sebagai berikut: instruksi untuk membaca sebuah data dalam bahasa *C* dan menyimpan hasil pembacaan di dalam peubah *x* adalah:

```
fread(x, sizeof(x), 1, Fin);
```

Pernyataan `fread` di atas berarti membaca nilai untuk x dengan ukuran (*sizeof*) sebesar ukuran tipe x sebanyak 1 elemen dari arsip `Fin`.

(i) Pernyataan untuk menulis data ke arsip (bahasa *Pascal*)

Misalkan data keluaran akan disimpan di dalam arsip yang bernama `hasil.txt`. Instruksi untuk menuliskan sebuah data (x) dalam bahasa *Pascal* adalah:

```
write(Fout, x);
```

(j) Pernyataan untuk membaca data dari arsip (bahasa *C*)

Misalkan data keluaran akan disimpan di dalam arsip yang bernama `hasil.txt`. Bahasa *C* memiliki banyak instruksi untuk menulis keluaran ke arsip. Salah satunya sebagai berikut: instruksi untuk menuliskan sebuah data (x) dalam bahasa *C* adalah:

```
fwrite(x, sizeof(x), 1, Fout);
```

Pernyataan *fwrite* di atas berarti memulis nilai x dengan ukuran (*sizeof*) sebesar ukuran tipe x sebanyak 1 elemen dari *file Fout*.

Contoh Pembacaan Data Masukan dari Arsip (1)

Tinjau kembali program *Pascal* dan *C* di dalam Contoh 5.5. Program tersebut membaca panjang dan lebar empat persegi panjang, menghitung luasnya, dan mencetak luasnya. Misalkan data panjang dan lebar disimpan di dalam arsip `data.txt`.

Misalkan arsip `data.txt` sudah berisi panjang dan lebar empat persegi panjang sebagai berikut (antara data dipisahkan oleh spasi):

```
15 20
```

Keluaran program, yaitu luas empat persegi panjang, ditulis ke arsip `hasil.txt`. Program *Pascal* dan *C* untuk spesifikasi masalah ini adalah seperti di bawah ini:

PASCAL:

```
program Luas_Empat_Persegi_Panjang;
{ Membaca panjang dan lebar empat persegi panjang dari arsip, menghitung
luasnya, lalu mencetak luas tersebut ke arsip "hasil.txt". }

(* DEKLARASI *)
var
```

```
   panjang  : real;        { panjang segiempat, dalam satuan cm }
   lebar    : real;        { lebar segiempat, dalam satuan cm }
   luas     : real;        { luas segiempat, dalam satuan cm² }
   Fin, Fout: text;        { pointer ke arsip masukan dan keluaran }

(* ALGORITMA: *)
begin
   { buka arsip masukan }
   assign(Fin, 'data.txt');
   reset(Fin);

   { buka arsip keluaran }
   assign(Fout, 'hasil.txt');
   rewrite(Fout);

   { baca panjang dan lebar dari arsip Fin }
   read(Fin, panjang, lebar);

   luas := panjang*lebar;

   { tulis luas empat persegi ke arsip Fout }
   writeln(Fout, 'Luas segiempat = ', luas);

   { tutup arsip }
   close(Fin);
   close(Fout);
end.
```

C:

```
/* PROGRAM LUAS SEGIEMPAT
/* Membaca panjang dan lebar empat persegi panjang dari arsip "data.txt",
menghitung luasnya, lalu mencetak luas tesrebut ke arsip "hasil.txt". */

#include <stdio.h>

main()
{
   /* DEKLARASI */
   float panjang;     /* panjang segiempat, dalam satuan cm */
   float lebar;       /* lebar segiempat, dalam satuan cm */
   float luas;        /* luas segiempat, dalam satuan cm² */

   FILE *Fin, *Fout;     /* pointer ke arsip masukan dan keluaran */

   /* ALGORITMA: */

   /* buka arsip masukan */
   Fin = fopen("data.txt", "r");

   /* buka arsip keluaran */
   Fout = fopen("hasil.txt", "w");

   /* baca panjang dan lebar dari arsip Fin */
   fscanf(Fin, "%f %f", &panjang, &lebar);

   luas = panjang * lebar;
```

```
  /* tulis luas empat persegi ke arsip Fout */
   fprintf(Fout, "Luas segiempat = %f \n", luas);

   /* tutup arsip */
   fclose(Fin);
   fclose(Fout);
}
```

Contoh Pembacaan Data Masukan dari Arsip (2)

Modifikasi program *Pascal* dan *C* dari Contoh 1 di atas sehingga nama arsip data masukan dan nama arsip keluaran tidak selalu harus `data.txt` maupun `hasil.txt`, tetapi merupakan masukan dari pengguna program. Program *Pascal* dan *C* untuk spesifikasi masalah ini adalah seperti di bawah ini:

PASCAL:

```
program Luas_Empat_Persegi_Panjang;
{ Membaca panjang dan lebar empat persegi panjang dari arsip "data.txt",
menghitung luasnya, lalu mencetak luas tersebut ke arsip "hasil.txt". }

(* DEKLARASI *)
var
   panjang  : real;        { panjang segiempat, dalam satuan cm }
   lebar    : real;        { lebar segiempat, dalam satuan cm }
   luas     : real;        { luas segiempat, dalam satuan cm² }
   Fin, Fout: text;        { pointer ke arsip masukan dan keluaran }

   NamaArsip1, NamaArsip2  : string[12];   { nama arsip masukan
                                             dan keluaran }
(* ALGORITMA: *)
begin
   write('Nama arsip masukan: '); readln(NamaArsip1);
   write('Nama arsip keluaran: '); readln(NamaArsip2);

   { buka arsip masukan }
   assign(Fin, NamaArsip1);
   reset(Fin);

   { buka arsip keluaran }
   assign(Fout, NamaArsip2);
   rewrite(Fout);

   { baca panjang dan lebar dari arsip Fin }
   read(Fin, panjang, lebar);

   luas := panjang * lebar;

   { tulis luas empat persegi ke arsip Fout }
   writeln(Fout, 'Luas segiempat = ', luas);

   { tutup arsip }
   close(Fin);
   close(Fout);
end.
```

C:

```
/* PROGRAM LUAS SEGIEMPAT
/* Membaca panjang dan lebar empat persegi panjang dari arsip "data.txt",
menghitung luasnya, lalu mencetak luas tersebut ke arsip "hasil.txt". */

#include <stdio.h>

main()
{
   /* DEKLARASI */
   float panjang;     /* panjang segiempat, dalam satuan cm */
   float lebar;       /* lebar segiempat, dalam satuan cm */
   float luas;        /* luas segiempat, dalam satuan cm² */

   char NamaArsip1[12], NamaArsip2[12];   /* nama arsip masukan
                                             dan keluaran */
   FILE *Fin, *Fout;     /* pointer ke arsip masukan dan keluaran */

   /* ALGORITMA: */

   printf("Nama arsip masukan: "); scanf("%s", NamaArsip1);
   printf("Nama arsip keluaran: "); scanf("%s", NamaArsip2);

   /* buka arsip masukan */
   Fin = fopen(NamaArsip1, "r");

   /* buka arsip keluaran */
   Fout = fopen(NamaArsip2, "w");

   /* baca panjang dan lebar dari arsip Fin */
   fscanf(Fin, "%f %f", &panjang, &lebar);

   luas = panjang * lebar;

   /* tulis luas empat persegi ke arsip Fout */
   fprintf(Fout, "Luas segiempat = %f \n", luas);

   /* tutup arsip */
   fclose(Fin);
   fclose(Fout);
}
```

Soal Latihan Bab 5

1. Buatlah sebuah algoritma dengan spesifikasi sebagai berikut:
   - menampilkan tulisan "Halo, siapa namamu?", lalu
   - meminta pengguna memasukkan namanya, dan akhirnya
   - menuliskan pesan "Senang berteman denganmu," <nama>, yang dalam hal ini <nama> adalah *string* yang dibaca berdasarkan (b).

2. Tulislah algoritma untuk menghitung luas bangun geometri yang lain (lingkaran, bujursangkar, segitiga, trapesium, dan sebagainya). Data masukan dibaca dari piranti masukan dan luas bangun ditampilkan sebagai keluaran.

3. Dibaca durasi waktu dalam detik. Tulislah algoritma untuk mengonversi durasi waktu tersebut ke dalam hari, jam, menit, detik.

4. Sebuah proyek dikerjakan selama x hari. Tulislah algoritma untuk mengonversi berapa tahun, berapa bulan, dan berapa hari proyek tersebut dikerjakan. Asumsikan: 1 tahun = 365 hari, 1 bulan = 30 hari. Keluaran (tahun, bulan, hari) ditampilkan ke piranti keluaran.

5. Dibaca dua buah tanggal (*dd:mm:yy*). Tulislah algoritma untuk menghitung berapa hari jarak kedua tanggal tersebut. Asumsikan: 1 tahun = 365 hari, 1 bulan = 30 hari. Keluaran (tahun, bulan, hari) ditampilkan ke piranti keluaran.

6. Dibaca tiga buah bilangan bulat x, y, dan z. Tulislah algoritma untuk mempertukarkan tripel (x, y, z) menjadi (y, z, x).

7. Buatlah algoritma yang membaca nilai uang (rupiah) dalam kelipatan 25, lalu menentukan berapa nilai tukaran pecahan. Pecahan yang tersedia adalah Rp1000, Rp500, Rp100, Rp50, dan Rp 25. Sebagai contoh, uang senilai Rp2775 setara dengan 2 buah pecahan Rp1000 ditambah 7 buah pecahan Rp100 ditambah 1 buah pecahan Rp50 ditambah 1 buah pecahan Rp25.

8. Seekor semut menempuh perjalanan sejauh x cm. Tulislah algoritma untuk mengonversi jarak x ke dalam kilometer-meter-sentimeter. Ingat bahwa 1 m = 100 cm dan 1 km = 1000 m = 100.000 cm. Misal x = 261341 cm, ini berarti semut menempuh jarak sejauh 2 km + 63 m + 141 cm.

9. Tuliskan algoritma yang membaca panjang sebuah benda dalam satuan meter, lalu mengonversinya ke dalam satuan inchi, kaki, dan *yard* (1 *inchi* = 25.4 mm, 1 kaki = 30.48 cm, dan 1 yard = 0.9144 m).

10. Berat badan ideal ada hubungannya dengan tinggi badan seseorang. Untuk menentukan berat badan ideal, tinggi badan dikurangi 100, lalu dikurangi lagi dengan 10% dari hasil pengurangan pertama. Tulislah algoritma yang membaca tinggi badan lalu menentukan berat badan yang ideal untuk tinggi tersebut.

6
Pemilihan

Pilih jalan yang mana?

Program yang hanya berisi runtunan instruksi biasanya terdapat pada masalah sederhana. Seringkali suatu instruksi hanya bisa dikerjakan jika ia memenuhi suatu persyaratan tertentu. Oleh karena itu, komputer tidak lagi mengerjakan instruksi secara sekuensial seperti pada runtunan, tetapi berdasarkan syarat yang dipenuhi. Struktur pemilihan memungkinkan kita melakukan aksi jika suatu syarat dpenuhi. Di dalam Bab 6 ini kita membahas konstruksi algoritma yang penting yaitu struktur pemilihan (*selection*).

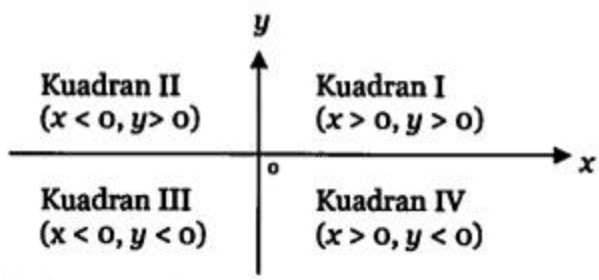
6.1 Menganalisis Kasus

Sebuah persoalan harus kita analisis untuk menentukan kasus-kasus yang mungkin terdapat di dalamnya. Untuk setiap kasus ada aksi tertentu yang dilakukan. Adanya analisis kasus menyebabkan terjadinya pemilihan instruksi (atau pencabangan) di dalam algoritma, bergantung pada kasus mana yang dipenuhi.

Sebagai contoh, kita ingin menentukan di kuadran mana terletak sebuah titik di bidang kartesian. Yang dimaksud dengan kuadran adalah seperempat bidang datar bidang yang terbagi oleh sumbu-sumbu koordinat (sumbu-X dan sumbu-Y). Ada empat kuadran di bidang kartesian, yaitu Kuadran I, Kuadran II, Kuadran III, dan Kuadran IV (Gambar 6.1). Suatu titik $P(x,y)$ dapat terletak pada salah satu dari empat kuadran tersebut, bergantung pada tanda x dan y (apakah positif atau negatif). Nilai a disebut positif jika $a > 0$, dan sebaliknya disebut negatif jika $a < 0$. Kita tidak mendefinisikan kuadran titik P jika titik tersebut terletak pada sumbu-sumbu koordinat (salah satu dari x atau $y = 0$).

Ada lima kasus yang harus kita analisis untuk menentukan kuadran titik $P(x,y)$, yaitu:
Kasus 1 : jika $x > 0$ dan $y > 0$, maka titik P terletak di kuadran I
Kasus 2 : jika $x < 0$ dan $y > 0$, maka titik P terletak di kuadran II
Kasus 3 : jika $x < 0$ dan $y < 0$, maka titik P terletak di kuadran III
Kasus 4 : jika $x > 0$ dan $y < 0$, maka titik P terletak di kuadran IV
Kasus 5 : selain Kasus 1, 2, 3, dan 4, maka titik P tidak terletak di kuadran manapun

Contohnya, titik (1, 3) terletak di kuadran I karena $1 > 0$ dan $3 > 0$, titik (-1, 3) di kuadran II, titik (-1, -3) di kuadran III, dan titik (3, -1) di kuadran IV.



Gambar 6.1 Penetapan kuadran untuk titik-titik di bidang kartesian. Kuadran adalah bidang yang merupakan seperempat dari bidang kartesian.

Perhatikan Kasus 1. Pada kasus ini ada pernyataan:

jika $x > 0$ dan $y > 0$, maka $P(x,y)$ terletak di kuadran I

Kondisi atau syarat yang harus dipenuhi agar $P(x, y)$ terletak di kuadran I adalah

$$x > 0 \text{ dan } y > 0$$

Kondisi bernilai *boolean* (*true* atau *false*) dan menentukan aksi yang dilakukan jika kondisi tersebut berlaku (memenuhi). Penting dicatat bahwa setiap kasus tidak boleh beririsan, dan analisis kasus harus mencakup semua kemungkinan kasus.

Kondisi bernilai *boolean* adalah ekspresi *boolean* yang bernilai *true* atau *false* bergantung pada nilai masing-masing *operand* yang terlibat di dalamnya. Ekspresi *boolean* dibentuk dengan mengombinasikan *operand* yang bertipe sama dengan salah satu dari operator relasional: $=, \neq, <, \leq, >, \geq$, dan operator *uner* not.
Contoh-contoh ekspresi *boolean*:

```
x > y
a ≠ 10
m = n
p ≤ q
a + b > 1
str = 'itb'
k mod 4 = 0
ketemu = true
not berhenti
(x > 0) and (y < 0)
```

Aksi yang dikerjakan jika kondisi *boolean* dipenuhi bergantung pada masalahnya, misalnya berupa pernyataan pengisian nilai (*assignment*), kalkulasi, baca, tulis, dan sebagainya. Penentuan kondisi *boolean* dan aksi yang dilakukan bergantung pada jumlah kasus yang terdapat pada masalah tersebut: satu kasus, dua kasus, atau lebih dari dua kasus. Masing-masing dijelaskan terpisah seperti yang dijelaskan berikut ini.

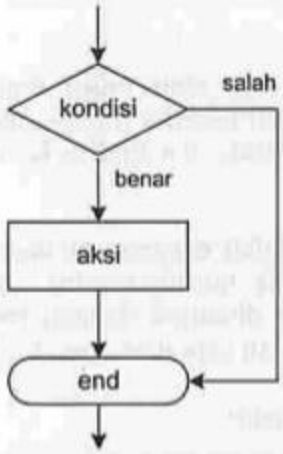
6.2 Satu Kasus

Notasi algoritmik untuk analisis dengan satu kasus adalah dengan menggunakan konstruksi *IF-THEN* (jika-maka) dalam bentuk pernyataan:

```
if kondisi then
   aksi
endif
```

Pernyataan di atas berarti bahwa *aksi* hanya dilaksanakan bila *kondisi* bernilai benar (*true*). Bila *kondisi* bernilai salah (*false*), tidak ada aksi apa pun yang dikerjakan. Kata endif sengaja kita tambahkan untuk mempertegas awal dan akhir struktur *IF-THEN*. Aksi yang dikerjakan bisa

satu, dua, atau lebih. Bagan-alir pada Gambar 6.2 membantu memperlihatkan visualisasi pemilihan dengan hanya satu kasus ini.



Gambar 6.2 Bagan alir yang memperlihatkan pemilihan dengan hanya satu kasus.

Contoh-contoh:

(a)
```
if x > 100 then
    x ← x + 1
endif
```

(b)
```
if kar = '*' then
    stop ← true
endif
```

(c)
```
if (a ≠ 0) or (p = 1) then
   q ← a * p
   write(q)
endif
```

(d)
```
if ada = false then
   read(cc)
   write(cc)
endif
```

Catatan:

Contoh (d) di atas dapat juga kita tulis sebagai berikut:

```
if not ada then
  read(cc)
  write(cc)
endif
```

karena aksi sesudah kata `then` hanya akan dikerjakan hanya jika kondisi bernilai *true* (dalam hal ini, kondisi adalah `not ada`, yang bernilai *true* bila `ada` bernilai `false`).

6.3 Contoh-contoh Masalah dengan Satu Kasus

Contoh 6.1. Buatlah algoritma yang membaca sebuah bilangan bulat, lalu mencetak pesan "genap" jika bilangan tersebut adalah genap.

Penyelesaian

Bilangan genap adalah bilangan yang habis dibagi dengan 2 (sisa pembagian = 0). Oleh karena itu, kita perlu membagi data masukan dengan 2. Jika data masukan habis dibagi 2, maka kita tulis bahwa bilangan tesrebut bilangan genap.

```
PROGRAM Genap
{ Mencetak pesan "bilangan genap" jika sebuah bilangan bulat yang
dibaca dari piranti masukan merupakan bilangan genap}

DEKLARASI
   x : integer

ALGORITMA:
   read(x)
   if x mod 2 = 0 then
      write('genap')
   endif
```

Algoritma 6.1 Mencetak "genap" jika data masukan adalah bilangan genap.

Contoh 6.2. Tulislah algoritma yang membaca sebuah karakter, lalu menuliskan pesan "huruf hidup" jika karakter tersebut merupakan salah satu dari huruf vokal.

Penyelesaian

Huruf vokal ada lima, yaitu *a*, *i*, *u*, *e*, dan *o*. Bandingkan karakter yang dibaca dengan kelima huruf tersebut. Jika karakter masukan sama dengan salah satu huruf vokal tersebut, maka tuliskan pesan bahwa karakter tersebut adalah huruf hidup.

```
PROGRAM Huruf_Vokal
{ Mencetak pesan "huruf vokal" bila sebuah karakter yang dibaca merupakan
huruf hidup, Asumsikan karakter yang dibaca adalah huruf kecil }

DEKLARASI
   c : char

ALGORITMA:
   read(c)
   if (c='a') or (c='i') or (c='u') or (c='e') or (c='o') then
     write('huruf vokal')
   endif
```

Algoritma 6.2 Mencetak "huruf hidup" jika data masukan adalah huruf vokal.

Contoh 6.3. Di dalam matematika kita mengenal nilai mutlak (*absolute value*). Untuk sembarang bilangan riil x, maka nilai mutlak, dilambangkan dengan $|x|$, adalah besaran x tanpa memperhitungkan tandanya. Nilai mutlak selalu positif. Jadi, jika x bernilai negatif, maka nilai mutlak dari x diperoleh dengan mengalikannya dengan negatif. Sebagai contoh, $|10.2| = 10.2$ dan $|-21| = -(-21) = 21$. Buatlah algoritma yang membaca sebuah bilangan riil lalu menentukan nilai mutlaknya.

Penyelesaian

Misalkan bilangan riil tersebut adalah x. Jika $x < 0$, maka nilai mutlaknya adalah $-x$. Tidak ada perubahan yang dilakukan jika $x \geq 0$, sebab nilai mutlak selalu positif atau nol.

```
PROGRAM NilaiMutlak
{ Menentukan nilai mutlak dari sebuah bilangan riil }

DEKLARASI
   x : real

ALGORITMA:
   read(x)
   if x < 0 then
     x ← -x
   endif
   write(x)
```

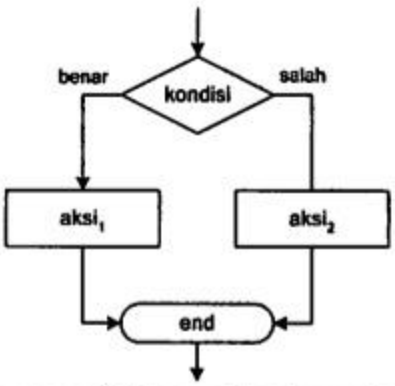
Algoritma 6.3 Menentukan nilai mutlak dari sebuah bilangan riil.

6.4 Dua Kasus

Konstruksi *IF-THEN* hanya menyediakan satu alternatif aksi jika suatu persyaratan (kondisi) dipenuhi. Kadang-kadang kita perlu memilih melakukan aksi alternatif jika suatu kondisi tidak memenuhi. Jadi, ada dua kasus, tetapi hanya salah satu dari keduanya yang harus dipilih satu untuk dikerjakan. Notasi algoritmik untuk masalah dengan dua buah kasus adalah dengan menggunakan konstruksi *IF-THEN-ELSE* (jika-maka-kalau tidak):

```
if kondisi then
  aksi1
else
  aksi2
endif
```

Pernyataan di atas berarti bahwa $aksi_1$ dikerjakan jika *kondisi* bernilai benar, sebaliknya jika *kondisi* bernilai salah, maka $aksi_2$ yang akan dilaksanakan. Perhatikanlah bahwa `else` menyatakan ingkaran (*negation*) dari *kondisi*. Bagan-alir pada Gambar 6.3 membantu memperlihatkan visualisasi pemilihan dengan dua kasus-kasus.



Gambar 6.3 Bagan alir yang memperlihatkan pemilihan dengan dua kasus.

Contoh-contoh:

```
(a) if a > 0 then
        write('bilangan positif')
    else
        write('bilangan bukan positif')
    endif

(b) if (k > 4) and (k div 2 = 4) then
        read(n)
        z ← n * k
    else
        read(m)
        z ← m/k
    endif

(c) if (n > 0) then
      if a > b then
         z ← a
      else
         z ← b
      endif
    else
      write('n harus positif')
    endif
```

6.5 Contoh-contoh Masalah dengan Dua Kasus

Contoh 6.4. Tulislah algoritma yang membaca sebuah bilangan bulat, lalu menuliskan pesan "genap" jika bilangan tersebut adalah genap, atau "ganjil" jika bilangan tersebut adalah bilangan ganjil.

Penyelesaian

Misalkan bilangan bulat yang dibaca adalah x. Hanya ada dua kemungkinan untuk x, yaitu genap atau ganjil. Bilangan genap adalah bilangan yang habis dibagi dengan 2 (memberikan sisa = 0), sedangkan bilangan ganjil bersisa 1 bila dibagi dengan 2. Contohnya, 10 adalah bilangan genap karena 10 mod 2 = 0, tetapi 9 adalah bilangan ganjil karena 9 mod 2 = 1.

Analisis kasus:

Kasus 1: jika x mod 2 = 0, maka tulis pesan "genap"
Kasus 2: jika x mod 2 ≠ 0, maka tulis pesan "ganjil"

```
PROGRAM GenapGanjil
{ Mencetak pesan "genap" jika sebuah bilangan bulat yang dibaca
merupakan bilangan genap, atau "ganjil" jika bilangan tersebut ganjil }

DEKLARASI
   x : integer

ALGORITMA:
   read(x)
   if x mod 2 = 0 then
      write('genap')
   else
      write('ganjil')
   endif
```

Algoritma 6.4 Menentukan genap atau ganjil.

Contoh 6.5. Buatlah algoritma yang membaca dua buah bilangan bulat, lalu menentukan bilangan yang terbesar.

Penyelesaian

Misalkan kedua bilangan tersebut A dan B. Hanya ada dua kemungkinan bilangan terbesar, A atau B. Kita harus membandingkan kedua bilangan tersebut lalu menetukan yang terbesar.

Analisis kasus:

Kasus 1: jika $A > B$, maka tulis pesan "Bilangan terbesar = A"
Kasus 2: jika $B \geq A$, maka tulis pesan "Bilangan terbesar = B"

```
PROGRAM Maksimum
{ Menentukan bilangan terbesar dari dua buah bilangan bulat }

DEKLARASI
   A, B : integer

ALGORITMA:
   read(A,B)
   if A > B then
     write('Bilangan terbesar = ', A)
   else { berarti B ≥ A }
```

```
    write('Bilangan terbesar = ', B)
  endif
```

Algoritma 6.5 Menentukan bilangan terbesar dari dua buah bilangan.

Contoh 6.6. Tuliskan algoritma yang membaca tiga buah bilangan bulat, lalu menentukan dari ketiga bilangan itu yang terbesar!

Penyelesaian

Ada dua cara penyelesaian masalah yang mungkin untuk masalah ini. Kedua versi penyelesaian masalah diberikan di bawah ini.

Versi 1:
Misalkan kedua bilangan tersebut *A*, *B*, dan *C*.
Analisis kasus:

Kasus 1: jika $A > B$, maka set *maks* = *A*
Kasus 2: jika $B \geq A$, maka set *maks* = *B*
Bandingkan *maks* dengan *C*: jika $C >$ *maks*, maka set *maks* = *C*

```
PROGRAM Maksimum3
{ Menentukan bilangan terbesar dari tiga buah bilangan bulat }

DEKLARASI
   A, B, C, maks : integer

ALGORITMA:
   read(A,B,C)

   { cari maks antara A dan B }
   if A > B then
     maks ← A
   else  { B ≥ A }
     maks ← B
   endif

   { bandingkan maks dengan C }
   if C > maks then
     maks ← C
   endif
   write(maks)
```

Algoritma 6.6 Menentukan bilangan terbesar dari tiga buah bilangan (versi 1).

Versi 2:
Misalkan kedua bilangan tersebut *A*, *B*, dan *C*.
Analisis kasus:

Kasus 1: jika $A > B$, maka
  Analisis kasus (*A*, *C*):
    Kasus 1.1: jika $A > C$, maka tulis "Bilangan terbesar = *A*"
    Kasus 1.2: jika $C \geq A$, maka tulis "Bilangan terbesar = *C*"

Kasus 2: jika $B \geq A$, maka
Analisis kasus (B, C):
Kasus 2.1: jika $B > C$, maka tulis "Bilangan terbesar =B"
Kasus 2.2: jika $C \geq B$, maka tulis "Bilangan terbesar = C"

```
PROGRAM Maksimum3
{ Menentukan bilangan terbesar dari tiga buah bilangan bulat }

DEKLARASI
   A, B, C : integer

ALGORITMA:
   read(A,B,C)
   if A > B then
      { bandingkan A dengan C }
      if A > C then
         write('Bilangan terbesar =',A)
      else
         write('Bilangan terbesar =', C)
      endif
   else  { berarti B ≥ A }
       { bandingkan B dengan C }
       if B > C then
          write('Bilangan terbesar =', B)
       else
          write('Bilangan terbesar =', C)
      endif
   endif
```

Algoritma 6.7 Menentukan bilangan terbesar dari tiga buah bilangan (versi 2).

atau dalam kondisi yang lebih ringkas:

```
PROGRAM Maksimum3
{ Menentukan bilangan terbesar dari tiga buah bilangan bulat }

DEKLARASI
   A, B, C : integer

ALGORITMA:
   read(A,B,C)
   if (A > B) and (A > C) then
     write('Bilangan terbesar =',A)
   else
     if (B > A) and ( B > C) then
       write('Bilangan terbesar =', B)
     else
        write('Bilangan terbesar =', C)
     endif
   endif
```

Algoritma 6.8 Menentukan bilangan terbesar dari tiga buah bilangan (versi 3).

Contoh 6.7. Karyawan honorer di PT "ABC" digaji berdasarkan jumlah jam kerjanya selama satu minggu. Upah per jam misalkan Rp2000,00. Bila jumlah jam kerja lebih besar dari 48 jam, maka sisanya dianggap sebagai jam lembur. Upah lembur misalkan Rp3000,00/jam. Tulislah algoritma yang membaca jumlah jam kerja seorang karyawan selama satu minggu, lalu menentukan upah mingguannya.

Penyelesaian

Misalkan jumlah jam kerja karyawan adalah *JJK*.
Analisis kasus:

Kasus 1: jika $JJK \leq 48$, maka upah = $JJK * 2000$
Kasus 2: jika $JJK > 48$, maka
lembur = $JJK - 48$
upah = $48 * 2000 + \text{lembur} * 3000$

```
PROGRAM Upah_Karyawan
{ Menentukan upah mingguan seorang karyawan. Masukan yang dibaca dari
papan kunci adalah nama karyawan, golongan, dan jumlah jam kerja.
Keluaran program adalah nama karyawan dan upahnya. }

DEKLARASI
   nama   : string   { nama karyawan }
   JJK    : integer  { Jumlah Jam Kerja }
   lembur : real     { jumlah jam lembur }
   upah   : real     { upah karyawan }

ALGORITMA:
   read(nama,JJK)
   if JJK ≤ 48 then
      upah ← JJK * 2000
   else  { berarti JJK > 48 }
      lembur ← JJK - 48
      upah ← 48 * 2000 + lembur * 3000
   endif
   write(nama,upah)
```

Algoritma 6.9 Menentukan honor karyawan dengan kemungkinan lembur (versi 1).

Angka-angka seperti 48 jam, Rp2000, dan Rp3000 adalah konstanta yang suatu saat mungkin berubah. Pada algoritma `Upah_Karyawan` di atas, angka-angka tersebut dinyatakan langsung di dalam algoritma. Bila sewaktu-waktu ada perubahan ketetapan, kita harus mengubah seluruh angka yang muncul di dalam instruksi. Hal ini merupakan pekerjaan yang kurang mangkus terutama bila suatu konstanta terdapat pada lebih dari satu instruksi. Untuk mengatasi masalah ini, pemrogram sangat dianjurkan menyatakan angka-angka tersebut sebagai konstanta (`const`), sehingga bila ada perubahan, kita cukup mengubahnya di bagian konstanta saja.

Dengan demikian, algoritma Upah_Karyawan ditulis lebih elegan seperti berikut ini:

```
PROGRAM Upah_Karyawan
{ Menentukan upah mingguan seorang karyawan. Masukan yang dibaca dari
papan kunci adalah nama karyawan, golongan, dan jumlah jam kerja.
Keluaran program adalah nama karyawan dan upahnya. }

DEKLARASI
   const JamNormal  = 48      { jumlah jam kerja normal per minggu }
   const UpahPerJam = 2000    { upah per jam, Rp 2000 }
   const UpahLembur = 3000    { upah per jam lembur, Rp 3000 }

   nama   : string   { nama karyawan }
   JJK    : integer  { Jumlah Jam Kerja }
   lembur : real     { jumlah jam lembur }
   upah   : real     { upah karyawan }

ALGORITMA:
   read(nama,JJK)
   if JJK ≤ JamNormal then
      upah ← JJK * UpahPerJam
   else
      lembur ← JJK - JamNormal
      upah ← JamNormal * UpahPerJam + lembur * UpahLembur
   endif
   write(nama,upah)
```

Algoritma 6.10 Menentukan honor karyawan dengan kemungkinan lembur (versi 2).

Contoh 6.8. Buatlah algoritma yang membaca angka tahun masehi dari papan ketik, lalu menentukan apakah tahun tersebut merupakan tahun kabisat. Secara sederhana, tahun kabisat adalah tahun yang habis dibagi dengan 4. Pada tahun kabisat, bulan Februari berjumlah 29 hari. Contoh tahun kabisat adalah 1996 dan 2000. Tahun 2002 bukan tahun kabisat karena tidak habis dibagi 4.

Penyelesaian

Misalkan peubah tahun masehi tersebut adalah *tahun*.
Analisis kasus:

Kasus 1: jika *tahun* mod 4 = 0, maka *tahun* adalah tahun kabisat
Kasus 2: jika *tahun* mod 4 ≠ 0, maka *tahun* bukan tahun kabisat

```
PROGRAM TahunKabisat
{ Menentukan apakah suatu tahun merupakan tahun kabisat atau bukan }

DEKLARASI
  tahun : integer

ALGORITMA:
  read(tahun)
  if tahun mod 4 = 0 then
    write('tahun kabisat')
```

```
  else
    write('bukan tahun kabisat')
  endif
```

Algoritma 6.11 Menentukan apakah sebuah tahun merupakan tahun kabisat (versi 1).

Sebenarnya penentuan tahun kabisat tidak sesederhana Algoritma 6.11 di atas. Suatu tahun disebut tahun kabisat jika memenuhi salah satu syarat berikut: (i) habis dibagi 4 tetapi tidak habis dibagi 100, atau (ii) habis dibagi 400. Misalnya, tahun 1996 adalah tahun kabisat karena habis dibagi 4 dan tidak habis dibagi 100, tetapi tahun 1900 bukan tahun kabisat karena 1900 habis dibagi 4 dan habis dibagi 100, juga tidak memenuhi syarat habis dibagi 400. Tahun 2000 adalah tahun kabisat habis dibagi 400. Algoritma 6.11 harus kita revisi menjadi Algoritma 6.12 berikut:

```
PROGRAM TahunKabisat
{ Menentukan apakah suatu tahun merupakan tahun kabisat atau bukan }

DEKLARASI
  tahun : integer

ALGORITMA:
  read(tahun)
  if (tahun mod 4 = 0 and tahun mod 100 ≠ 0) or (tahun mod 400 = 0) then
    write('tahun kabisat')
  else
    write('bukan tahun kabisat')
  endif
```

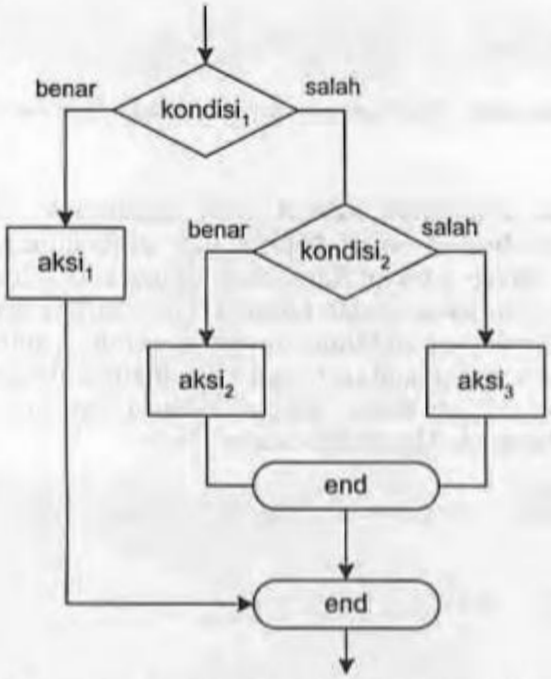
Algoritma 6.12 Menentukan apakah sebuah tahun merupakan tahun kabisat (versi 2).

6.6 Tiga Kasus atau Lebih

Masalah yang mempunyai tiga buah kasus atau lebih dapat dianalisis dengan konstruksi *IF-THEN-ELSE* bertingkat-tingkat. Gambar 6.4 memperlihatkan bagan alir untuk masalah dengan 3 kasus.

Tiga Kasus:

```
if kondisi 1 then
   aksi 1
else
   if kondisi 2 then
      aksi 2
else
      if kondisi 3 then
         aksi 3
      endif
   endif
endif
```



Gambar 6.4 Bagan alir yang memperlihatkan pemilihan dengan tiga kasus.

Empat Kasus:

```
if kondisi1 then
   aksi1
else
   if kondisi2 then
      aksi2
   else
      if kondisi3 then
         aksi3
   else
         if kondisi4 then
            aksi4
         endif
      endif
   endif
endif
```

dan seterusnya untuk lima kasus, enam kasus, ...

6.7 Contoh-contoh Masalah dengan Tiga Kasus atau Lebih

Contoh 6.9. Tulislah algoritma yang membaca sebuah bilangan bulat, lalu menentukan apakah bilangan tersebut positif, negatif, atau nol.

Penyelesaian

Baca bilangan bulat itu adalah x.
Analisis kasus:

Kasus 1: jika $x > 0$, maka x adalah bilangan positif
Kasus 2: jika $x < 0$, maka x adalah bilangan negatif
Kasus 3: jika $x = 0$, maka x adalah bilangan nol.

```
PROGRAM JenisBilanganBulat
{ Menentukan apakah sebuah bilangan bulat merupakan bilangan positif,
negatif, atau nol. }

DEKLARASI
   x : integer

ALGORITMA:
   read(x)
   if x > 0 then
      write('positif')
   else
      if x < 0 then
         write('negatif')
      else
         if x = 0
            write('nol')
         endif
      endif
   endif
```

Algoritma 6.13 Menentukan bilangan positif, negatif, atau nol.

Kita dapat juga menuliskan analisis kasus tanpa memeriksa kasus ketiga, karena jika suatu bilangan bulat bukan positif atau negatif, pastilah bilangan tersebut 0:

```
if x > 0 then
   write('positif')
else
   if x < 0 then
      write('negatif')
   else
      write('nol')
   endif
endif
```

Contoh 6.10. Tulislah algoritma yang membaca temperatur air, T, (dalam satuan derajat celcius) pada tekanan normal, lalu menentukan apakah wujud air tersebut dalam keadaan padat ($T \leq 0°C$), cair ($0<T<100$), atau gas ($T > 100$).

Penyelesaian

Misalkan suhu air adalah T.

Analisis kasus:

Kasus 1: jika $T \leq 0$, maka air berwujud padat
Kasus 2: jika $0 < T < 100$, maka air berwujud cair
Kasus 3: jika $T \geq 100$, maka air berwujud uap

```
PROGRAM WujudAir
{ Menentukan wujud air: padat, cair, atau gas, bergantung pada suhunya }

DEKLARASI
  T : real  { suhu air, dalam derajat celcius }

ALGORITMA:
  read(T)
  if T ≤ 0 then                          { kasus 1 }
    write('padat')
  else
    if (T > 0) and (T < 100) then        { kasus 2 }
      write('cair')
    else
      if T ≥ 100 then                    { kasus 3 }
        write('gas atau uap');
      endif
    endif
  endif
```

Algoritma 6.14 Menentukan wujud air.

Analisis kasus rentang nilai T di dalam Algoritma 6.14 atas dapat juga ditulis sebagai berikut:

```
  if T ≤ 0 then          { kasus 1 }
    write('padat')
  else
    if T < 100 then      { kasus 2 }
      write('cair')
    else { T ≥ 100 }     { kasus 3 }
      write('gas atau uap');
    endif
  endif
```

Contoh 6.11. Buatlah algoritma yang membaca sebuah titik $P(x, y)$ di bidang kartesian, lalu menentukan di kuadran mana letak titik tersebut (lihat contoh ilustrasi pada bagian awal bab ini).

Penyelesaian

Analisis kasus:

Kasus 1 : jika $x > 0$ dan $y > 0$, maka $P(x,y)$ terletak di kuadran I
Kasus 2 : jika $x < 0$ dan $y > 0$, maka $P(x,y)$ terletak di kuadran II
Kasus 3 : jika $x < 0$ dan $y < 0$, maka $P(x,y)$ terletak di kuadran III
Kasus 4 : jika $x > 0$ dan $y < 0$, maka $P(x,y)$ terletak di kuadran IV
Kasus 5 : selain Kasus 1, 2, 3, dan 4, maka $P(x,y)$ tidak terletak di kuadran manapun

```
PROGRAM KuadranTitik
{ Menentukan kuadran titik P(x,y) di bidang kartesian }

DEKLARASI
   type Titik : record < x : real, y : real>
   P : Titik

ALGORITMA:
  read(P.x, P.y)
  if (P.x > 0) and (P.y > 0) then
     write('Kuadran 1')
  else
     if (P.x < 0) and (P.y > 0) then
        write('Kuadran 2')
     else
        if (P.x < 0) and (P.y < 0) then
           write('Kuadran 3')
        else
           if (P.x > 0) and (P.y < 0) then
             write('Kuadran 4')
           else
             write('Tidak terletak di kuadran manapun')
           endif
        endif
     endif
  endif
```

Algoritma 6.15 Menentukan kuadran titik di bidang kartesian.

Contoh 6.12. Misalkan karyawan PT "ABC" dikelompokkan berdasarkan golongannya. Upah per jam tiap karyawan bergantung pada golongannya (lihat tabel di bawah). Jumlah jam kerja yang normal selama seminggu adalah 48 jam. Kelebihan jam kerja dianggap lembur dengan upah lembur adalah Rp3000/jam untuk semua golongan karyawan. Buatlah algoritma yang membaca nama karyawan dan jumlah jam kerjanya selama seminggu, lalu menghitung gaji mingguannya.

Golongan	Upah Per Jam
A	Rp 4000
B	Rp 5000
C	Rp 6000
D	Rp 7500

Penyelesaian

Analisis kasus masalah ini lebih rumit. Mula-mula kita harus menentukan upah per jam berdasarkan golongannya.

Kasus 1 : jika golongan = 'A', maka upah per jam = 4000
Kasus 2 : jika golongan = 'B', maka upah per jam = 4000
Kasus 3 : jika golongan = 'C', maka upah per jam = 4000
Kasus 4 : jika golongan = 'D', maka upah per jam = 4000

Selanjutnya, kita menghitung upah yang dihitung dari jumlah jam kerja. Upah per jam bergantung pada hasil analisis kasus yang pertama. Jika ada jam lembur, maka upah total adalah upah kerja + upah lembur. Penentuan apakah ada jam lembur atau tidak sudah dijelaskan pada Contoh 6.7.

```
PROGRAM UpahKaryawan
{ Menghitung upah mingguan karyawan. Masukan yang dibaca adalah nama
karyawan, golongan, dan jumlah jam kerja. Keluaran program adalah nama
karyawan dan upahnya }

DEKLARASI
   const JamKerjaNormal  = 48 { jumlah jam kerja normal per minggu }
   const UpahLembur = 3000    { upah per jam lembur, Rp 3000 }

   Nama        : string
   gol         : char        {'A', 'B', 'C', 'D'}
   JJK         : integer     { jumlah jam kerja karyawan dalam seminggu }
   JamLembur   : integer
   UpahPerJam  : real        { upah per jam }
   UpahTotal   : real

ALGORITMA:
   read(Nama, gol, JJK)
   if gol = 'A' then
     UpahPerJam ← 4000.0
   else
     if gol = 'B' then
       UpahPerJam ← 5000.0
     else
      if gol = 'C' then
         UpahPerJam ← 6000.0
      else
         if gol = 'D' then
           UpahPerJam ← 7500.0
         endif
      endif
     endif
   endif

   if JJK ≤ JamNormal then
      UpahTotal ← JJK * UpahPerJam
   else
      JamLembur ← JJK - JamKerjaNormal
      UpahTotal ← JamKerjaNormal * UpahPerJam + JamLembur*UpahLembur
   endif
   write(Nama,UpahTotal)
```

Algoritma 6.16 Menentukan upah karyawan, bergantung golongannya.

Contoh 6.13. Indeks nilai mahasiswa ditentukan berdasarkan nilai ujian yang diraihnya. Ketentuan pemberian nilai indeks sebagai berikut:

jika nilai ujian ≥ 80 , indeks nilai = A
jika 70 ≤ nilai ujian < 80, indeks nilai = B
jika 55 ≤ nilai ujian < 70, indeks nilai = C
jika 40 ≤ nilai ujian < 55, indeks nilai = D
jika nilai ujian < 40 , indeks nilai = E

Buatlah algoritma yang membaca nilai ujian seorang mahasiswa, lalu menentukan indeks nilainya, kemudian mencetak nilai dan indeksnya ke piranti keluaran.

Penyelesaian

Analisis permasalahan ini menghasilkan empat buah kasus, yaitu kasus bila nilai ujian di atas 80, antara 70 dan 80, antara 55 dan 70, antara 40 dan 55, dan terakhir di bawah 40. Untuk masing-masing kasus ditentukan indeks nilainya, seperti yang dinyatakan dalam permasalahan.

```
PROGRAM INDEKS_NILAI_UJIAN
{ Menghitung indeks nilai ujian mahasiswa }

DEKLARASI
  nilai : real
 indeks : char

ALGORITMA:
  read(nilai)
  if nilai ≥ 80 then
    indeks ← 'A'
  else
    if (nilai ≥ 70) and (nilai < 80) then
      indeks ← 'B'
    else
      if (nilai ≥ 55) and (nilai < 70) then
        indeks ← 'C'
      else
        if (nilai ≥ 40) and (nilai < 55) then
          indeks ← 'D'
        else
          indeks ← 'E'
        endif
      endif
    endif
  endif
  write(indeks)
```

Algoritma 6.17 Menghitung indeks nilai ujian mahasiswa.

6.8 Struktur *CASE*

Untuk masalah dengan dua kasus atau lebih, konstruksi *CASE* dapat menyederhanakan penulisan *IF-THEN-ELSE* yang bertingkat-tingkat sebagaimana pada contoh-contoh sebelum ini. Konstruksi *CASE* sebagai berikut:

```
case ekspresi
       nilai1 : aksi1
       nilai2 : aksi2
       nilai3 : aksi3
         .
         .
         .
       nilain : aksin
       otherwise : aksix
endcase
```

ekspresi adalah sembarang ekspresi (aritmetika atau *boolean*) yang menghasilkan suatu nilai (konstanta). Konstruksi *CASE* memeriksa apakah nilai dari ekspresi tersebut sama dengan salah satu dari $nilai_1$, $nilai_2$, ..., $nilai_n$ (catatan: semua nilai-nilai ini harus berbeda). Jika nilai ekspresi sama dengan $nilai_k$ benar, maka $aksi_k$ dilaksanakan. Aksi yang bersesuaian dengan $nilai_k$ dapat lebih dari satu, karena itu ia berupa runtunan. Jika tidak ada satu pun nilai ekspresi yang cocok, maka aksi sesudah *otherwise* dikerjakan. *otherwise* bersifat *optional*, artinya ia boleh ditulis atau tidak di dalam konstruksi *CASE*.

Konstruksi CASE di atas menggantikan analisis kasus yang ekivalen dengan konstruksi *IF-THEN-ELSE* berikut ini:

```
if ekspresi = nilai1 then
    aksi1
  else
    if ekspresi = nilai2 then
      aksi2
    else
      if ekspresi = nilai3 then
        aksi3
        .
        .
        .
      if ekspresi = nilain then
        aksin
      else  { otherwise }
        aksix
      endif
    endif
  endif
```

Tidak semua bahasa pemrograman menyediakan struktur *CASE* (misalnya bahasa *Fortran*). Bahasa *Pascal* dan *C* menyediakan struktur ini (lihat

upabab 6.9 mengenai translasi notasi algoritma ke dalam notasi *Pascal* dan *C*). Jika bahasa pemrograman tidak menyediakan struktur *CASE*, maka *CASE* dapat diganti dengan struktur *IF-THEN-ELSE* yang ekivalen.

Contoh 6.14. Buatlah algoritma yang membaca sebuah bilangan bulat yang nilainya terletak antara 1 sampai 4, lalu mencetak tulisan angka tersebut. Misalkan bila dibaca angka 1, maka tercetak tulisan "satu", bila dibaca 2, maka tercetak di layar tulisan "dua", demikian seterusnya. Jika angka yang dimasukkan selain 1 sampai 4, tuliskan pesan bahwa angka yang dimasukkan salah.

Penyelesaian

Dengan struktur *IF-THEN-ELSE*, algoritma mencetak kata untuk angka yang bersesuaian sebagai berikut:

```
PROGRAM KonversiAngkaKeTeks
{ Mencetak kata untuk angka 1 sampai 4. }

DEKLARASI
   angka : integer     { angka yang dibaca }

ALGORITMA:
  read(angka)
  if angka = 1 then
    write('satu')
  else
    if angka = 2 then
      write('dua')
    else
      if angka = 3 then
        write('tiga')
      else
        if angka = 4 then
          write('empat')
        else
          write('angka yang dimasukkan harus 1 s/d 4')
        endif
      endif
    endif
  endif
```

Algoritma 6.18 Mencetak angka menjadi teks (dengan konstruksi *IF-THEN-ELSE*).

Dengan konstruksi *CASE*, algoritma untuk masalah di atas dapat dibuat menjadi lebih singkat sebagai berikut:

```
PROGRAM KonversiAngkaKeTeks
{ Mencetak kata untuk angka 1 sampai 4. }

DEKLARASI
   angka : integer   { angka yang dibaca }

ALGORITMA:
  read(angka)
```

```
case angka
  1 : write('satu')
  2 : write('dua')
  3 : write('tiga')
  4 : write('empat')
  otherwise : write('angka yang dimasukkan salah')
endcase
```

Algoritma 6.19 Mencetak angka menjadi teks (dengan konstruksi *CASE*).

Perhatikan Algoritma 6.19 di atas. Ekspresi hanya berupa sebuah peubah (yaitu angka). Ekspresi dapat saja berupa bentuk yang melibatkan *operand* dan operator sebagaimana sudah dijelaskan di dalam Bab 4. Contohnya seperti algoritma 6.20 berikut ini.

```
PROGRAM GenapGanjil
{ Mencetak pesan "genap" jika sebuah bilangan bulat yang dibaca
merupakan bilangan genap, atau "ganjil" jika bilangan tersebut ganjil }

DEKLARASI
   x : integer

ALGORITMA:
   read(x)
   case x mod 2
     0 : write('genap')
     1 : write('ganjil')
   endcase
```

Algoritma 6.20 Menentukan genap atau ganjil.

Contoh 6.15. Buatlah algoritma yang membaca nomor bulan (*integer*), lalu menuliskan nama bulan sesuai angka bulannya. Misalnya jika dibaca bulan 8, maka tercetak "Agustus".

Penyelesaian

Masalah ini memiliki 13 buah kasus karena nama bulan berbeda-beda bergantung pada nomor bulan yang diberikan (ada 12 bulan dalam kalender masehi). Satu kasus tambahan adalah bila nomor bulan yang dimasukkan di luar rentang 1..12.

```
PROGRAM NamaBulan
{ Mencetak nama bulan berdasarkan nomor bulan (1..12) }

DEKLARASI
   Nomor_bulan : integer

ALGORITMA:
   read(nomor_bulan)
   case nomor_bulan
     1 : write('Januari')
     2 : write('Pebruari')
     3 : write('Maret')
     4 : write('April')
```

```
      5 : write('Mei')
      6 : write('Juni')
      7 : write('Juli')
      8 : write('Agustus')
      9 : write('September')
      10 : write('Oktober')
      11 : write('November')
      12 : write('Desember')
      otherwise : write('Bukan bulan yang benar')
  endcase
```

Algoritma 6.21 Mencetak nama bulan berdasarkan nomornya.

Contoh 6.16. Tulislah kembali Algoritma 6.16 dengan menggunakan konstruksi *CASE*.

Penyelesaian

Konstruksi *CASE* akan kita gunakan untuk menggantikan konstruksi *IF-THEN-ELSE* pada analisis golongan karyawan.

```
PROGRAM UpahKaryawan
{ Menghitung upah mingguan karyawan. Masukan yang dibaca adalah nama
karyawan, golongan, dan jumlah jam kerja. Keluaran program adalah nama
karyawan dan upahnya }

DEKLARASI
   const JamKerjaNormal  = 48 { jumlah jam kerja normal per minggu }
   const UpahLembur = 3000    { upah per jam lembur, Rp 3000 }

   Nama        : string
   gol         : char       {'A', 'B', 'C', 'D'}
   JJK         : integer
   JamLembur   : integer
   UpahPerJam  : real       { upah per jam }
   UpahTotal   : real

ALGORITMA:
   read(Nama, gol, JJK)
   case (gol)
     'A' : UpahPerJam ← 4000.0
     'B' : UpahPerJam ← 5000.0
     'C' : UpahPerJam ← 6000.0
     'D' : UpahPerJam ← 7500.0
   endcase

   if JJK ≤ JamNormal then
      UpahTotal ← JJK * UpahPerJam
   else
      JamLembur ← JJK - JamKerjaNormal
      UpahTotal ← JamKerjaNormal * UpahPerJam + JamLembur*UpahLembur
   endif
   write(Nama,UpahTotal)
```

Algoritma 6.22 Menentukan upah karyawan, bergantung golongannya (*CASE*).

6.9 Contoh-contoh Tambahan

Contoh 6.17. Buatlah algoritma yang membaca sebuah jam (*hh*:*mm*:*ss*), lalu menentukan jam yang baru setelah jam yang lama ditambah (*increment*) satu detik. Misalnya,

Jam Lama	Jam Lama + 1 Detik
14:27:31	14:27:32
15:36:59	15:37:00
10:59:59	11:00:00
23:59:59	00:00:00

Penyelesaian

Cara 1 (komputasi):
(a) konversi jam-menit-detik (*hh*:*mm*:ss) ke jumlah detik
(b) tambahkan jumlah detik dengan 1
(c) konversi kembali hasil dari (b) ke dalam jam-menit-detik (*hh*:*mm*:ss)

```
PROGRAM JamBerikutnya
{ Menentukan jam berikutnya setelah jam sekarang ditambah 1 detik.}

DEKLARASI
   type Jam : record <hh:integer, { 0..59 }
                      mm:integer, { 0..59 }
                      ss:integer  { 0..23 }
                     >
   J : Jam
   TotalDetik, SisaDetik : integer

ALGORITMA:
   read(J.hh,J.mm,J.ss)    {baca jam}

   TotalDetik ← J.hh*3600 + J.mm*60 + J.ss  {konversi J ke jumlah detik}

   TotalDetik ← TotalDetik+1    {tambahkan TotalDetik dengan 1}

   {konversi TotalDetik ke hh:mm:ss }
   J.hh ← TotalDetik div 3600
   SisaDetik ← TotalDetik mod 3600
   J.mm ← SisaDetik div 60
   J.ss ← sisadetik mod 60

   write(J.hh,J.mm,J.ss)    { jam yang baru }
```

Algoritma 6.23 Menentukan jam berikutnya setelah ditambah satu detik (versi 1).

Cara 2 (analisis kasus):
(a) Tambahkan *ss* dengan 1. Jika jumlahnya < 60 → OK, tetapi jika = 60, maka set *ss* kembali menjadi 0 dan tambahkan 1 ke *mm*.

(b) Jika pertambahan *mm* dengan 1 jumlahnya < 60 → OK, tetapi jika = 60, maka set *mm* kembali menjadi nol dan tambahkan 1 ke *hh*.
(c) Jika pertambahan *hh* dengan 1 jumlahnya < 24 → OK, tetapi jika = 24, maka set *hh* kembali menjadi 0.

Contoh:
14:27:31 + 1 detik → 14:27:(31+1) = 14:27:32
15:36:59 + 1 detik → 15:36:(59+1) = 15:(36+1):00 = 15:37:00
10:59:59 + 1 detik → 10:59:(59+1) = 10:(59+1):00 = (10+1):00:00 = 11:00:00
23:59:59 + 1 detik → 00:00:00

```
PROGRAM JamBerikutnya
{ Menentukan jam berikutnya setelah jam sekarang ditambah 1 detik.}

DEKLARASI
 type Jam : record<hh:integer, { 0..59 }
                   mm:integer, { 0..59 }
                   ss:integer  { 0..23 }
                  >
 J : Jam

ALGORITMA:
read(J.hh,J.mm,J.ss)   { baca jam }

   if J.ss + 1 < 60 then  { OK, tidak ada masalah penambahan 1 detik }
      J.ss ← J.ss + 1
   else { berarti J.ss + 1 = 60 }
      J.ss←0    { detik kembali menjadi nol, menit bertambah 1, tapi
                  periksa dulu apakah menit + 1 < 60 }
      if J.mm + 1 < 60 then { OK, tidak ada masalah penambahan 1 menit }
         J.mm ← J.mm + 1
      else  { berarti J.mm + 1 = 60 }
         J.mm ← 0 { menit menjadi nol, jam bertambah 1, tapi periksa dulu
                    apakah jam + 1 < 24 }
         if J.hh + 1 < 24 then  {OK, tidak ada masalah penambahan 1 jam }
            J.hh ← J.hh+1
         else  { berarti J.hh + 1 = 24 }
            J.hh←0
         endif
      endif
   endif

    write(J.hh,J.mm,J.ss)
```

Algoritma 6.24 Menentukan jam berikutnya setelah ditambah satu detik (versi 1)

Contoh 6.18. Simulasikan sebuah kalkulator sederhana untuk melakukan operasi aritmetika sebagai berikut: dibaca *operand* 1, *operator*, dan *operand* 2. Tentukan hasil operasi aritmetika:

*operand*1 *operator* *operand*2

dan cetak hasilnya. Misalnya, bila dibaca 8, '+', 7, maka hasilnya = 15 (yaitu 8+7); bila dibaca 6, '*', 5, maka hasilnya = 30 (yaitu 6 × 5);

Penyelesaian

Analisis kasus terhadap masalah ini menghasilkan algoritma KalkulatorSederhana sebagai berikut:

```
PROGRAM KaklkulatorSederhana
{ Mensimulasikan kalkulator sederhana, yaitu menghitung hasil operasi
aritmetika bila dibaca operand_1, operator, dan operand_2.}

DEKLARASI
   op1, op2 : integer    { operand }
   oprt     : char       {'+', '-', '*', '/'}    { operator }

ALGORITMA:
   read(op1, oprt, op2)
   case (oprt)
     '+' : write(op1 + op2)
     '-' : write(op1 - op2)
     '*' : write(op1 * op2)
     '/' : write(op1/op2)
   endcase
```

Algoritma 6.25 Simulasi kalkulator sederhana (fatal).

Algoritma 6.25 di atas mengandung kesalahan yang sangat fatal, yaitu jika nilai op2 sama dengan nol karena pembagian dengan nol tidak terdefinisi. Karena itu, analisis kasus tambahan harus dilakukan untuk kasus operator '/', yaitu:

```
case (oprt)
     :
     '/' : if op2 ≠ 0 then
              write(op1/op2)
           else
              write('Eror!')
           endif
   endcase
```

Algoritma 6.25 diperbaiki menjadi Algoritma 6.26 sebagai berikut:

```
PROGRAM KaklkulatorSederhana
{ Mensimulasikan kalkulator sederhana, yaitu menghitung hasil operasi
aritmetika bila dibaca operand_1, operator, dan operand_2.}

DEKLARASI
   op1, op2 : integer    { operand }
   oprt     : char       { '+', '-', '*', '/' }    { operator }

ALGORITMA:
   read(op1, oprt, op2)
   case (oprt)
     '+' : write(op1 + op2)
     '-' : write(op1 - op2)
     '*' : write(op1 * op2)
```

```
    '/' : if op2 ≠ 0 then
            write(op1/op2)
          else
            write('Eror!')
          endif
  endcase
```

Algoritma 6.26 Simulasi kalkulator sederhana (dengan penanganan op2 = 0).

Contoh 6.19. Konstruksi *CASE* sering digunakan dalam memilih menu program. Program menawarkan sejumlah menu. Pengguna cukup mengetikkan nomor menu yang diinginkan. Setiap kali nomor menu dipilih, maka prosedur yang berasosiasi dengan nomor menu tersebut dieksekusi. Misalkan sebuah program mempunyai menu sebagai berikut:

```
     MENU

1. Baca data
2. Cetak data
3. Ubah data
4. Hapus data
5. Keluar program
```

Buatlah algoritma yang mencetak menu tersebut dan membaca nomor pilihan menu. Untuk setiap nomor menu yang dipilih, cukup tuliskan pesan seperti contoh berikut:

```
Anda memilih menu nomor <NomorMenu>
```

yang dalam hal ini `<NomorMenu>` adalah nomor menu yang dimasukkan oleh pengguna program.

Penyelesaian

```
PROGRAM SimulasiMenuProgram
{ Menampilkan menu, membaca pilihan menu, dan menampilkan nomor menu
yang dipilih oleh pengguna }

DEKLARASI
  NomorMenu : integer

ALGORITMA:
  { Cetak menu }
  write('     MENU          ')
  write(' 1. Baca data      ')
  write(' 2. Cetak data     ')
  write(' 3. Ubah data      ')
  write(' 4. Hapus data     ')
  write(' 5. Keluar program ')
  write(' Masukkan pilihan anda(1/2/3/4/5)? ')

  read(NomorMenu)    { baca nomor menu yang akan dipilih }

  case NomorMenu
    1 : write('Anda memilih menu nomor 1')
```

```
    2 : write('Anda memilih menu nomor 2')
    3 : write('Anda memilih menu nomor 3')
    4 : write('Anda memilih menu nomor 4')
    5 : write('Anda memilih menu nomor 5')
  endcase
```

Algoritma 6.27 Simulasi menu program.

Sebagai contoh pengunaan menu program, buatlah algoritma yang memungkinkan pengguna dapat memilih untuk menentukan luas empat persegi panjang, keliling empat persegi panjang, dan panjang diagonal. Lengkapi program dengan pilihan menu: 1) Luas, 2) Keliling, 3) Panjang diagonal, 4) Keluar program. Pada setiap pilihan menu, dibaca panjang dan lebar empat persegi panjang, lalu dihitung hasil yang diinginkan pengguna.

Penyelesaian

```
PROGRAM EmpatPersegiPanjang
{ Menampilkan menu perhitungan empat persegi panjang, memilih menu,
dan melakukan proses perhitungan }

DEKLARASI
  NomorMenu : integer
  panjang, lebar : real
  luas, keliling, diagonal : real

ALGORITMA:
  { Cetak menu }
  write('   MENU EMPAT PERSEGI PANJANG       ')
  write('      1. Hitung luas                ')
  write('      2. Hitung keliling            ')
  write('      3. Hitung panjang diagonal    ')
  write('      4. Keluar program             ')
  write('   Masukkan pilihan anda(1/2/3/4)?  ')
  read(NomorMenu)

  case NomorMenu
    1 : read(panjang,lebar)
        luas ← panjang*lebar
        write(luas)

    2 : read(panjang,lebar)
        keliling ← 2*panjang + 2*lebar
        write(keliling)

    3 : read(panjang,lebar)
        diagonal ← sqrt(panjang*panjang + lebar*lebar)
        write(diagonal)

    4 : write('Keluar program... sampai jumpa')
  endcase
```

Algoritma 6.28 Menghitung properti empat persegi panjang dengan pilihan menu.

Contoh 6.20. Buatlah algoritma yang membaca nomor bulan dan tahun, lalu menuliskan jumlah hari dalam bulan tersebut. Misalnya jika dibaca bulan 8 (bulan Agustus), maka jumlah harinya adalah 31.

Penyelesaian

Setiap bulan mempunyai jumlah hari yang berbeda-beda. Kita harus mengidentifikasi bulan-bulan dan jumlah harinya sebagai berikut:

Bulan	Jumlah hari
1, 3, 5, 7, 8, 10, 12	31
4, 6, 9, 11	30
2	29 (jika tahun kabisat), 28 (jika bukan kabisat

```
PROGRAM JumlahHari
{ Menentukan jumlah hari dalam satu bulan}

DEKLARASI
   nomor_bulan: integer  {1..12}
   tahun      : integer  { > 0 }
   jumlah_hari : integer

ALGORITMA:
   read(nomor_bulan, tahun)
   case nomor_bulan
      1,3,5,7,8,10,12 : jumlah_hari ← 31
      4,6,9,11        : jumlah_hari ← 30
      2 : if (tahun mod 4 = 0 and tahun mod 100 ≠ 0) or
             (tahun mod 400 = 0) then {tahun kabisat}
             jumlah_hari ← 29
          else  {bukan tahun kabisat }
             jumlah_hari ← 28
          endif
   endcase
   write(jumlah_hari)
```

Algoritma 6.29 Menghitung jumlah hari dalam satu bulan.

6.10 Tabel Translasi Notasi Algoritmik Struktur Pemilihan ke dalam Notasi Pascal dan C

Kelompok	*Algoritma*	Pascal	C

1. IF-THEN	if *kondisi* then
	  aksi
	endif
		if *kondisi* **then**
		  aksi;
		*(*endif*)*
		
		Catatan: bila aksi lebih dari satu buah:
		  if *kondisi* **then**
		  begin
		  aksi$_1$;
		  aksi$_2$;
		  ...
		  aksi$_n$;
		  end; *(*if*)*
			if (*kondisi*)
			  aksi;
			*/*endif*/*
			
			Catatan: bila aksi lebih dari satu buah:
			  if (*kondisi*)
			  {
			  aksi$_1$;
			  aksi$_2$;
			  ...
			  aksi$_n$;
			  }

2. IF-THEN-ELSE	if *kondisi* then
	  aksi$_1$
	else
	  aksi$_2$;
	endif
		if *kondisi* **then**
		  aksi$_1$
		else
		  aksi$_2$;
		*(*endif*)*
			if (*kondisi*)
			  aksi$_1$;
			else
			  aksi$_2$;
			*/*endif*/*

3. CASE	case *ekspresi*
	  nilai$_1$: *aksi*$_1$;
	  nilai$_2$: *aksi*$_2$;
	  .
	  .
	  .
	  nilai$_n$: *aksi*;
	  otherwise : *aksi*;
	endcase
		case *ekspresi* **of**
		  nilai$_1$: *aksi*$_1$;
		  nilai$_2$: *aksi*$_2$;
		  .
		  .
		  .
		  nilai$_n$: *aksi*$_n$;
		  else *aksi*$_x$;
		end;
		
		Keterangan:
		nilai1, nilai2, ...
		adalah nilai yang bertipe integer, char,atau boolean.
			switch (*ekspresi*)
			{
			case *nilai*$_1$: *aksi*$_1$;
			  break;
			case *nilai*$_2$: *aksi*$_2$;
			  break;
			  .
			  .
			  .
			case *nilai*$_n$: *aksi*$_n$;
			  break;
			default : aksiX;
			}
			
			Keterangan:
			nilai1, nilai2, ...
			adalah nilai yang bertipe int atau char.

Contoh-contoh translasi:

1. Algoritma untuk menentukan apakah sebuah bilangan bulat merupakan bilangan genap

ALGORITMIK:

```
PROGRAM BilanganGenap
{ Mencetak pesan "genap" jika sebuah bilangan bulat yang dibaca
merupakan bilangan genap}

DEKLARASI
   x : integer

ALGORITMA:
   read(x)
   if x mod 2 = 0 then
      write('genap')
   endif
```

PASCAL:

```
PROGRAM BilanganGenap;
{ Mencetak pesan "genap" jika sebuah bilangan bulat yang dibaca
merupakan bilangan genap}

(*DEKLARASI*)
var x : integer;

(* ALGORITMA:*)
begin
   write('Ketikkan sembarang bilangan bulat :'); readln(x);
   if x mod 2 = 0 then
      writeln('genap');
   (*endif*)
end.
```

C:

```
/* PROGRAM BilanganGenap */
/* Mencetak pesan "genap" jika sebuah bilangan bulat yang dibaca
merupakan bilangan genap */

#include <stdio.h>

main()
{  /*DEKLARASI*/
   int bil;

/* ALGORITMA:*/

   printf("Ketikkan sembarang bilangan bulat :"); scanf("%d",&x);
   if (x % 2 == 0)
      printf("genap");
   /*endif*/
}
```

2. Menentukan bilangan terbesar (maksimum) dari dua buah bilangan

ALGORITMIK:

```
PROGRAM Maksimum
{ Menentukan bilangan terbesar dari dua buah bilangan bulat }

DEKLARASI
   A, B : integer

ALGORITMA:
   read(A,B)
   if A > B then
     write('Bilangan terbesar : ', A)
   else  { berarti B ≥ A }
     write('Bilangan terbesar : ', B)
   endif
```

PASCAL:

```
program Maksimum;
{ Menentukan bilangan terbesar dari dua buah bilangan bulat }

(*DEKLARASI*)
var A, B : integer;

(*ALGORITMA:*)
begin
   write('A=?'); readln(A);
   write('B=?'); readln(B);
   if A > B then
     writeln('Bilangan terbesar : ', A)
   else  { B >= A }
     writeln('Bilangan terbesar : ', B);
   (*endif*)
end.
```

C:

```
/* PROGRAM Maksimum */
/* Menentukan bilangan terbesar dari dua buah bilangan bulat */
#include <stdio.h>
main()
{
   /*DEKLARASI*/
   int A, B;

   /*ALGORITMA:*/
   printf("A=?"); scanf("%d",&A);
   printf("B=?"); scanf("%d",&B);
   if (A > B)
     printf("Bilangan terbesar : %d", A);
   else  /* berarti B >= A */
     printf("Bilangan terbesar : %d", B);
   /*endif*/
}
```

3. Menentukan jam yang baru setelah ditambah 1 detik

ALGORITMIK:

```
PROGRAM JamBerikutnya
{ Menentukan jam berikutnya setelah jam sekarang ditambah 1 detik.}

DEKLARASI
 type Jam : record<hh:integer, { 0..59 }
                   mm:integer, { 0..59 }
                   ss:integer  { 0..23 }
                  >
 J : Jam

ALGORITMA:

read(J.hh,J.mm,J.ss)   { baca jam }

   if J.ss + 1 < 60 then  { OK, tidak ada masalah penambahan 1 detik }
      J.ss ← J.ss + 1
   else { berarti J.ss + 1 = 60 }
      J.ss ← 0  { detik kembali menjadi nol, menit bertambah 1, tapi
                  periksa dulu apakah menit + 1 < 60 }
      if J.mm + 1 < 60 then { OK, tidak ada masalah penambahan 1 menit }
         J.mm ← J.mm + 1
      else  { berarti J.mm + 1 = 60 }
         J.mm ← 0 { menit menjadi nol, jam bertambah 1, tapi periksa dulu
                    apakah jam + 1 < 24 }
         if J.hh + 1 < 24 then  {OK, tidak ada masalah penambahan 1 jam }
            J.hh ← J.hh+1
         else  { berarti J.hh + 1 = 24 }
            J.hh ← 0
         endif
      endif
   endif

   write(J.hh,J.mm,J.ss)
```

PASCAL:

```
program JamBerikutnya;
{ Menentukan jam berikutnya setelah jam sekarang ditambah 1 detik.}

(* DEKLARASI *)
type Jam = record
             hh:integer; { 0 .. 59 }
             mm:integer; { 0 .. 59 }
             ss:integer; { 0 .. 23 }
           end;

var  J : Jam;

(* ALGORITMA: *)
begin
   { baca jam }
```

```
   write('jam (hh)   : '); readln(J.hh);
   write('menit (mm): '); readln(J.mm);
   write('detik (ss): '); readln(J.ss);

   if J.ss + 1 < 60 then  { OK, tidak ada masalah penambahan 1 detik }
       J.ss := J.ss + 1
   else { berarti J.ss + 1 = 60 }
     begin
       J.ss := 0; { detik kembali menjadi nol, menit bertambah 1, tapi
                    periksa dulu apakah menit + 1 < 60 }
       if J.mm + 1 < 60 then { OK, tidak ada masalah penambahan 1 menit }
          J.hh := J.hh + 1
       else  { berarti J.mm + 1 = 60 }
         begin
           J.mm := 0;{ OK, menit menjadi nol, jam bertambah 1, tapi
                       periksa dulu apakah jam + 1 < 24 }


           if J.hh + 1 < 24 then {OK,tidak ada masalah penambahan 1 jam }
              J.hh := J.hh + 1
           else  { berarti J.hh + 1 = 24 }
              J.hh := 0;
           {endif}
         end;{if}
      end;{if}

   writeln('Jam yang baru  ', J.hh,':',J.mm,':',J.ss);
end.
```

C:

```
/* PROGRAM JamBerikutnya */
/* Menentukan jam berikutnya setelah jam sekarang ditambah 1 detik. */

#include <stdio.h>

main()
{
   /* DEKLARASI */
   typedef struct {int hh; /* 0 .. 59 */
                   int mm; /* 0 .. 59 */
                   int ss; /* 0 .. 23 */
                  }Jam;

   Jam  J;

/* ALGORITMA: */

   /*  baca jam */
   printf("jam (hh)   : "); scanf("%d",&J.hh);
   printf("menit (mm): "); scanf("%d",&J.mm);
   printf("detik (ss): "); scanf("%d",&J.ss);

   if (J.ss + 1 < 60)  /* OK, tidak ada masalah penambahan 1 detik */
      J.ss = J.ss + 1;
   else /* berarti J.ss + 1 = 60 */
      {
```

```
      J.ss = 0; /* detik kembali menjadi nol, menit bertambah 1, tapi
                   Periksa dulu apakah menit + 1 < 60 */
      if (J.mm + 1 < 60) /* OK, tidak ada masalah penambahan 1 menit */
         J.mm = J.mm + 1;
      else  /* berarti J.mm + 1 = 60 */
        {
          J.mm = 0;/* menit menjadi nol, jam bertambah 1, tapi
                      periksa dulu apakah jam + 1 < 24 */
          if (J.hh + 1 < 24) /*OK,tidak ada masalah penambahan 1 jam */
             J.hh = J.hh + 1;
          else  /* berarti J.hh + 1 = 24 */
             J.hh = 0;
          /*endif*/
        }
    }

  printf("Jam yang baru  %d:%d:%d \n", J.hh,J.mm,J.ss);
}
```

4. Menentukan nama bulan berdasakan nomor bulannya

ALGORITMIK:

```
PROGRAM NamaBulan
{ Mencetak nama bulan berdasarkan nomor bulan (1..12)}

DEKLARASI
   Nomor_bulan : integer

ALGORITMA:
   read(nomor_bulan)
   case nomor_bulan
      1 : write('Januari')
      2 : write('Februari')
      3 : write('Maret')
      4 : write('April')
      5 : write('Mei')
      6 : write('Juni')
      7 : write('Juli')
      8 : write('Agustus')
      9 : write('September')
      10 : write('Oktober')
      11 : write('November')
      12 : write('Desember')
     otherwise : write('Bukan bulan yang benar')
    endcase
```

PASCAL:

```
program NamaBulan;
{ Mencetak nama bulan berdasarkan nomor bulan (1..12)}

(* DEKLARASI *)
var
   nomor_bulan : integer;
```

```
(* ALGORITMA: *)
begin
   write('Ketikkan nomor bulan (1-12):'); readln(nomor_bulan);
   case nomor_bulan of
        1 : writeln('Januari');
        2 : writeln('Februari');
        3 : writeln('Maret');
        4 : writeln('April');
        5 : writeln('Mei');
        6 : writeln('Juni');
        7 : writeln('Juli');
        8 : writeln('Agustus');
        9 : writeln('September');
       10 : writeln('Oktober');
       11 : writeln('November');
       12 : writeln('Desember');
       else writeln('Bukan bulan yang benar');
  end;
end.
```

C:

```
/* PROGRAM NAMA NAMA BULAN */
/* Mencetak nama bulan berdasarkan nomor bulan (1..12) */

#include <stdio.h>

main()
{
   /* DEKLARASI */
   int nomor_bulan;

   /* ALGORITMA: */
   printf("Masukkan angka bulan (1-12):"); scanf("%d", &nomor_bulan);
   switch (AngkaBulan)
   {
    case 1 : printf("Januari \n"); break;
    case 2 : printf("Februari \n"); break;
    case 3 : printf("Maret \n"); break;
    case 4 : printf("April \n"); break;
    case 5 : printf("Mei \n"); break;
    case 6 : printf("Juni \n"); break;
    case 7 : printf("Juli \n"); break;
    case 8 : printf("Agustus \n"); break;
    case 9 : printf("September \n"); break;
    case 10 : printf("Oktober \n"); break;
    case 11 : printf("November \n"); break;
    case 12 : printf("Desember \n"); break;
    default : printf("Bukan bulan yang benar \n");
   }
}
```

5. Algoritma menu empat persegi panjang

ALGORITMIK:

```
PROGRAM EmpatPersegiPanjang
{ Menampilkan menu perhitungan empat persegi panjang, memilih menu,
dan melakukan proses perhitungan }

DEKLARASI
  NomorMenu : integer
  panjang, lebar : real
  luas, keliling, diagonal : real

ALGORITMA:
  { Cetak menu }
  write('   MENU EMPAT PERSEGI PANJANG       ')
  write('      1. Hitung luas                ')
  write('      2. Hitung keliling            ')
  write('      3. Hitung panjang diagonal    ')
  write('      4. Keluar program             ')
  write('   Masukkan pilihan anda(1/2/3/4)?  ')
  read(NomorMenu)

  case NomorMenu
     1 : read(panjang,lebar)
         luas ← panjang * lebar
         write(luas)


     2 : read(panjang,lebar)
         keliling ← 2*panjang + 2*lebar
         write(keliling)

     3 : read(panjang,lebar)
         diagonal ← sqrt(panjang*panjang + lebar*lebar)
         write(diagonal)

     4 : write('Keluar program')
  endcase
```

PASCAL:

```
program EmpatPersegiPanjang;
{ Menampilkan menu perhitungan empat persegi panjang, memilih menu,
dan melakukan proses perhitungan }

(* DEKLARASI *)
var
  NomorMenu : integer;
  panjang, lebar : real;
  luas, keliling, diagonal : real;

(* ALGORITMA: *)
begin
  { Cetak menu }
  writeln('   MENU EMPAT PERSEGI PANJANG       ');
  writeln('      1. Hitung luas                ');
```

```
  writeln('      2. Hitung keliling              ');
  writeln('      3. Hitung panjang diagonal      ');
  writeln('      4. Keluar program               ');
  write('   Masukkan pilihan anda(1/2/3/4)? '); readln(NomorMenu);

  case NomorMenu of
    1 : begin
          write('Panjang? '); readln(panjang);
          write('Lebar? '); readln(lebar);
          luas := panjang * lebar;
          writeln('Luas = ',luas);
        end;
    2 : begin
          write('Panjang? '); readln(panjang);
          write('Lebar? '); readln(lebar);
          keliling := 2*panjang + 2*lebar;
          writeln('Keliling = ',keliling);
        end;
    3 : begin
          write('Panjang? '); readln(panjang);
          write('Lebar? '); readln(lebar);
          diagonal := sqrt(panjang*panjang + lebar*lebar);
          writeln('Diagonal = ', diagonal);
        end;
    4 : writeln('Keluar program');
  end;
end.
```

C:

```
/* PROGRAM EmpatPersegiPanjang */
/* Menampilkan menu perhitungan empat persegi panjang, memilih menu,
dan melakukan proses perhitungan */

#include <stdio.h>

main()
{
   /* DEKLARASI */
  int NomorMenu;
  float panjang, lebar;
  float luas, keliling, diagonal;

/* ALGORITMA: */

  /* Cetak menu */
  printf("   MENU EMPAT PERSEGI PANJANG     \n");
  printf("      1. Hitung luas              \n");
  printf("      2. Hitung keliling          \n");
  printf("      3. Hitung panjang diagonal  \n");
  printf("      4. Keluar program           \n");
  printf(" Masukkan pilihan anda(1/2/3/4)? ");
  scanf("%d",&NomorMenu);

  switch (NomorMenu)
    case 1 : {
               printf("Panjang? "); scanf("%f",&panjang);
```

```
            printf("Lebar? "); scanf("%f",&lebar);
            luas = panjang * lebar;
            printf("Luas = %f \n",luas);
            break;
          }

    case 2 : {
            printf("Panjang? "); scanf("%f"&panjang);
            printf("Lebar? "); scanf("%f"&lebar);
            keliling = 2*panjang + 2*lebar;
            printf("Keliling = %f \n",keliling);
            break;
          }

    case 3 : {
            printf("Panjang? "); scanf("%f"&panjang);
            printf("Lebar? "); scanf("%f"&lebar);
            diagonal = sqrt((double)panjang*panjang +
            (double)lebar * lebar);
            printf("Diagonal = %f \n",diagonal);
            break;
          }

    case 4 : printf("Keluar program \n");
            break;
  }
}
```

6. Algoritma untuk mencetak nama-nama provinsi bila diberikan nama ibu kota di Pulau Jawa

ALGORITMIK:

```
PROGRAM NamaPropinsi
{ Mencetak nama propinsi jika diberikan nama ibukota provinsi di pulau
Jawa. }

DEKLARASI
   ibk : string { nama ibukota provinsi di Jawa }

ALGORITMA:
   read(ibk)
   case ibk
      'serang'     : write('Banten')
      'jakarta'    : write('DKI Jakarta')
      'bandung'    : write('Jawa Barat')
      'semarang'   : write('Jawa Tengah')
      'yogyakarta': write('DI Yogyakarta')
      'surabaya'   : write('Jawa Timur')
   endcase
```

PASCAL:

Konstruksi *CASE* di dalam algoritma ini tidak dapat diterjemahkan menjadi *case*, karena nilai konstanta dari `ibk` bertipe *string*. Karena itu, konstruksi *CASE* hanya dapat diterjemahkan menjadi konstruksi *IF-THEN-ELSE*.

```
program IBUKOTA;
{ Diberikan nama ibukota provinsi di pulau Jawa. Harus ditentukan
nama
  provinsinya }

(* DEKLARASI *)
var
   ibk : string[10]; { nama ibukota provinsi di Jawa }

(* ALGORITMA: *)
begin
   write('Ketikkan nama ibukota di Jawa:'); readln(ibk);
   if ibk ='serang' then
      writeln('banten')
   else
      if ibk = 'jakarta' then
        writeln('DKI Jakarta')
      else
         if ibk = 'bandung' then
           writeln('Jawa Barat')
         else
           if ibk = 'semarang' then
              writeln('Jawa Tengah')
           else
              if ibk = 'yogyakarta' then
                 writeln('DI Yogyakarta')


              else
                 if ibk = 'surabaya' then
                   writeln('Jawa Timur');
                 {endif}
              {endif}
           {endif}
         {endif}
      {endif}
   {endif}
end.
```

C:

```
/* PROGRAM IBUKOTA */
/* Diberikan nama ibukota provinsi di pulau Jawa. Harus ditentukan
nama provinsinya */

#include <stdio.h>
#include <string.h>   /* syarat menggunakan fungsi strcmp */

main()
{  /* DEKLARASI */
   char ibk[10]; /* nama ibukota provinsi di Jawa */
```

```
  /* ALGORITMA: */
  printf("Ketikkan nama ibukota di Jawa:"); scanf("%s",ibk);
  if (strcmp(ibk,"serang")== 0)
     printf("Banten \n");
  else
     if (strcmp(ibk,"jakarta") == 0)
        printf("DKI Jakarta \n");
     else
        if (strcmp(ibk,"bandung") == 0)
           printf("Jawa Barat \n");
        else
           if (strcmp(ibk,"semarang") == 0)
              printf("Jawa Tengah \n");
           else
              if (strcmp(ibk,"yogyakarta") == 0)
                 printf("DI Yogyakarta \n");
              else
                 if (strcmp(ibk,"surabaya" == 0)
                    printf("Jawa Timur \n");
                 /*endif*/
              /*endif*/
           /*endif*/
        /*endif*/
     /*endif*/
  /*endif*/
}
```

Catatan:

Fungsi `strcmp` *membandingkan kesamaan dua buah string. Nilai* `strcmp(s1,s2)` *sama dengan 0 jika* `s1 = s2`*, dan tidak sama dengan 0 jika* `s1 ≠ s2`*.*

7. Algoritma untuk menentukan jumlah hari dalam satu bulan

ALGORITMIK:

```
PROGRAM JumlahHari
{ Menentukan jumlah hari dalam satu bulan}

DEKLARASI
   nomor_bulan: integer  {1..12}
   tahun      : integer  { > 0 }
   jumlah_hari : integer

ALGORITMA:
   read(nomor_bulan, tahun)
   case nomor_bulan
      1,3,5,7,8,10,12 : jumlah_hari ← 31
      4,6,9,11        : jumlah_hari ← 30
      2 : if (tahun mod 4 = 0 and tahun mod 100 ≠ 0) or
            (tahun mod 400 = 0)) then  { tahun kabisat }
          jumlah_hari ← 29
        else
```

```
            {bukan tahun kabisat }
            jumlah_hari ← 28
          endif
   endcase
   write(jumlah_hari)
```

PASCAL:

```
program JumlahHari;
{ Menentukan jumlah hari dalam satu bulan}

(*DEKLARASI:*)
var
   nomor_bulan: integer;  {1..12}
   tahun      : integer;  { > 0 }
   jumlah_hari : integer;

(*ALGORITMA:*)
begin
   write('Bulan (1-12)?'); readln(nomor_bulan);
   write('Tahun?'); readln(tahun);
   case nomor_bulan of
      1,3,5,7,8,10,12 : jumlah_hari := 31;
      4,6,9,11        : jumlah_hari := 30;
      2               : if ((tahun mod 4 = 0) and (tahun mod 100 <> 0))
                            or (tahun mod 400 = 0)
                        then  { tahun kabisat }
                          jumlah_hari := 29
                        else  {bukan tahun kabisat }
                          jumlah_hari := 28;
                        {endif}
   end; {case}
   writeln('Jumlah hari dalam bulan ',nomor_bulan,
           ' adalah ',jumlah_hari);
end.
```

C:

```
/* PROGRAM JumlahHari */
/* Menentukan jumlah hari dalam satu bulan */

#include <stdio.h>

main()
{
   /* DEKLARASI */

   int nomor_bulan;  /* 1..12 */
   int tahun;        /* > 0 */
   int jumlah_hari;

   /* ALGORITMA: */
   printf("Bulan (1-12)?"); scanf("%d",&nomor_bulan);
   printf("Tahun?"); scanf("%d", &tahun);
   switch (nomor_bulan)
```

```
  {
     case 1:
     case 3:
     case 5:
     case 7:
     case 8:
     case 10:
     case 12 :
               jumlah_hari = 31;
               break;
     case 4:
     case 6:
     case 9:
     case 11:
              jumlah_hari = 30;
              break;
     case 2:  if ((tahun % 4 == 0 && tahun % 100 != 0) ||
                  tahun % 400 == 0)
                /* tahun kabisat */
                jumlah_hari = 29;
              else
                /* bukan tahun kabisat */
                jumlah_hari = 28;
             /*endif*/
  }

  printf("Jumlah hari dalam bulan %d adalah %d",
         nomor_bulan,jumlah_hari);
}
```

Soal Latihan Bab 6

1. Buatlah algoritma yang membaca sebuah bilangan bulat positif lalu menentukan apakah bilangan tersebut merupakan kelipatan 4.

2. Pasar swalayan *X* memberikan diskon harga bagi pembeli yang nilai total belanjanya lebih dari Rp100.000. Tulislah algoritma untuk menentukan harga belanja setelah dikurangi diskon. Data masukan adalah nilai total belanja pembeli, sedangkan keluarannya adalah diskon harga dan nilai belanja setelah dikurangi diskon.

3. Tulislah algoritma yang membaca tiga buah bilangan bulat, lalu mengurutkan tiga buah bilangan tersebut dari nilai yang kecil ke nilai yang besar. Keluaran adalah tiga buah bilangan yang terurut.

4. Tulislah algoritma yang membaca panjang (*integer*) tiga buah sisi sebuah segitiga, *a*, *b*, dan *c*, yang dalam hal ini $a \leq b \leq c$, lalu menentukan apakah ketiga sisi tersebut membentuk segitiga siku-siku, segitiga lancip, atau segitiga tumpul (Petunjuk: gunakan hukum *Phytagoras*).
5. Tulislah algoritma yang membaca sebuah karakter digit ('0'..'9') lalu mengonversinya menjadi nilai *integer* (0..9). Misalnya, jika dibaca

karakter '5', maka nilai konversinya ke *integer* adalah 5. Buatlah masing-masing algoritma untuk dua keadaan berikut:

(a) karakter digit yang dibaca diasumsikan sudah benar terletak dalam rentang '0'..'9'

(b) karakter yang dibaca mungkin bukan digit '0'..'9'. Jika karakter yang dibaca bukan karakter digit, maka hasil konversinya diasumsikan bernilai –99.

6. Jika kita berbelanja di pasar swalayan/supermarket, nila total belanja kita seringkali bukan kelipatan pecahan rupiah yang berlaku. Misalnya, nilai total belanja adalah Rp19.212,-. Andaikan saat ini pecahan rupiah yang berlaku paling kecil Rp25,-. Selain itu, juga ada pecahan Rp50,- dan Rp100,-. Umumnya kasir pasar swalayan membulatkan nilai belanja ke pecahan yang terbesar. Jadi Rp19.212,- dibulatkan menjadi Rp19.225,-. Hal ini jelas merugikan konsumen. Misalkan Anda memiliki pasar swalayan yang jujur dan tidak merugikan pembeli, sehingga jika ada nilai belanja yang bukan kelipatan pecahan yang ada, maka nilai belanja itu dibulatkan ke pecahan terendah. Jadi, Rp19.212,- dibulatkan menjadi Rp19.200. Tulislah algoritma yang membaca nilai belanja (*integer*) lalu membulatkannya ke nilai uang dengan pecahan terendah.

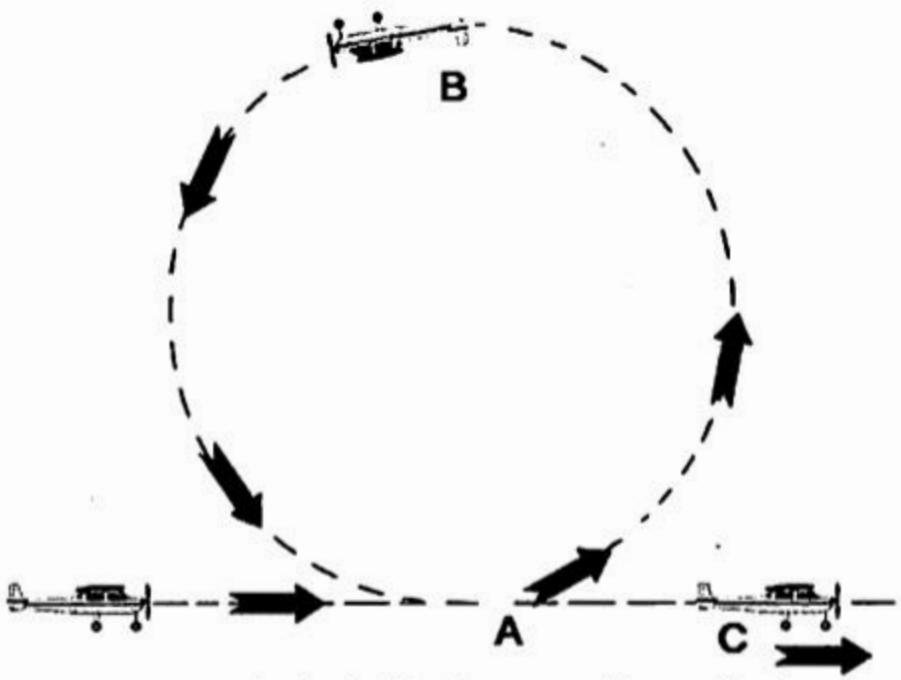
7. (a) Tuliskan algoritma yang membaca bilangan bulat positif dalam rentang 1 sampai 10, lalu mengonversinya ke dalam angka romawi.
   (b) Kembangkan algoritma (a) di atas sehingga dapat mengonversi bilangan bulat positif sembarang ke dalam angka romawinya.

8. Dalam bidang pengolahan citra (*image processing*), elemen gambar terkecil disebut *pixel* (*picture element*). Nilai *pixel* untuk gambar 256 warna adalah dari 0 sampai 255. Operasi-operasi terhadap *pixel* seringkali berada di luar rentang nilai ini. Jika ini kasusnya, maka nilai hasil operasi harus dipotong (*clipping*) sehingga tetap berada di dalam interval [0..25]. Jika nilai hasil operasi lebih besar dari 255, maka nilai tersebut dipotong menjadi 255, dan bila negatif maka dipotong menjadi 0. Dibaca sebuah nilai hasil operasi pengolahan citra, buatlah algoritma untuk melakukan *clipping* tersebut.

9. Tinjau kembali soal latihan Bab 5 nomor 10 untuk menentukan berat badan ideal. Dibaca berat badan dan tinggi badan seseorang. Tulislah pesan "ideal" jika berat badan orang tersebut hanya berselisih ± 2 kg dari berat badan ideal, atau pesan "tidak ideal" jika tidak berselisih ± 2 kg dari berat badan ideal. Tulislah algoritmanya.

10. Tuliskan ke dalam bahasa *Pascal* dan *C* untuk algoritma soal nomor 1 sampai nomor 9 di atas.

7
Pengulangan



Berputar terus berulang kali.(Sumber: www.rc-airplane-world.com)

Salah satu kelebihan komputer dibandingkan dengan manusia adalah kemampuannya untuk melaksanakan suatu perintah berulangkali tanpa mengenal lelah dan bosan. Di dalam algoritma, pengulangan atau kalang (*repetition* atau *loop*) dapat dilakukan sejumlah kali, atau sampai kondisi berhenti pengulangan tercapai. Bab 7 ini membahas struktur pengulangan.

7.1 Struktur Pengulangan

Struktur pengulangan secara umum terdiri atas dua bagian:
1. kondisi pengulangan, yaitu ekspresi *boolean* yang harus dipenuhi untuk melaksanakan pengulangan. Kondisi ini ada yang dinyatakan secara eksplisit oleh pemrogram atau dikelola sendiri oleh komputer (implisit);
2. badan (*body*) pengulangan, yaitu bagian algoritma yang diulang.

Di samping itu, struktur pengulangan biasanya disertai dengan bagian:
1. inisialisasi, yaitu, aksi yang dilakukan sebelum pengulangan dilakukan pertama kali;
2. terminasi, yaitu, aksi yang dilakukan setelah pengulangan selesai dilaksanakan.

Inisialisasi dan terminasi tidak selalu harus ada, namun pada berbagai kasus inisialisasi umumnya diperlukan.

Struktur pengulangan secara umum:

<inisialisasi>
awal pengulangan
  badan pengulangan
akhir pengulangan
<terminasi>

yang dalam hal ini awal dan akhir pengulangan dinyatakan sebagai kata kunci yang bergantung pada konstruksi pengulangan yang digunakan. Selain itu, *<inisialisasi>* dan *<terminasi>* adalah bagian yang opsional (tidak selalu harus dituliskan).

Di dalam algoritma terdapat beberapa macam konstruksi pengulangan yang berbeda. Beberapa konstruksi dapat dipakai untuk masalah yang sama, namun ada konstruksi pengulangan yang hanya cocok dipakai untuk masalah tertentu. Pemilihan konstruksi pengulangan untuk masalah tertentu dapat mempengaruhi kebenaran algoritma. Pemilihan konstruksi pengulangan yang tepat bergantung pada masalah yang akan diprogram. Buku ini hanya memberikan 3 macam notasi konstruksi pengulangan dalam bentuk pernyataan (*repeat N times* tidak dibicarakan), yaitu:
1. Pernyataan *FOR*;
2. Pernyataan *WHILE*;
3. Pernyataan *REPEAT*.

Pernyataan *FOR* adalah konstruksi **pengulangan tanpa kondisi** (*unconditional looping*), artinya instruksi-instruksi di dalam badan pengulangan diulangi sejumlah kali yang dispesifikasikan oleh pemrogram. Dalam hal ini, jumlah pengulangan sudah diketahui sebelum konstruksi pengulangan eksekusi.

Pada sebagian besar masalah, jumlah pengulangan tidak diketahui sebelum eksekusi program. Yang dapat ditentukan hanyalah kondisi berhenti pengulangan, artinya instruksi-instruksi di dalam badan pengulangan diulangi sampai kondisi berhenti terpenuhi. Jenis pengulangan ini termasuk ke dalam **pengulangan dengan kondisi** (*conditional looping*). Pernyataan *WHILE* dan *REPEAT* termasuk ke dalam jenis pengulangan ini. Masing-masing konstruksi pengulangan dijelaskan pada upa-bab berikut.

7.2 Pernyataan *FOR*

Konstruksi *FOR* digunakan untuk menghasilkan pengulangan sejumlah kali yang telah dispesifikasikan. Jumlah pengulangan diketahui atau dapat ditentukan sebelum eksekusi. Untuk mencacah berapa kali pengulangan dilakukan, kita memerlukan sebuah peubah (*variable*) pencacah (*counter*). Peubah ini nilainya selalu bertambah satu setiap kali pengulangan dilakukan. Jika cacah pengulangan sudah mencapai jumlah yang dispesifikasikan, maka proses pengulangan berhenti.

Bentuk umum pernyataan *FOR* ada dua macam: menaik (*ascending*) atau menurun (*descending*):

***FOR* menaik:**

```
for pencacah ← nilai_awal to nilai_akhir do
  aksi
endfor
```

Keterangan:

(i) *pencacah* haruslah dari tipe data yang memiliki *predecessor* dan *successor*, yaitu *integer* atau karakter. Tipe riil tidak dapat digunakan sebagai pencacah.
(ii) *aksi* dapat berupa satu atau lebih instruksi yang diulang.
(iii) *nilai_awal* harus lebih kecil atau sama dengan *nilai_akhir*. Jika *nilai_awal* lebih besar dari nilai_akhir, maka badan pengulangan tidak dimasuki.
(iv) Pada awalnya, *pencacah* diinisialisasi dengan *nilai_awal*. Nilai *pencacah* secara otomatis bertambah satu setiap kali badan pengulangan dimasuki, sampai akhirnya nilai *pencacah* sama dengan *nilai_akhir*.
(v) Jumlah pengulangan yang terjadi adalah *nilai_akhir* – *nilai_awal* + 1.

Contoh 7.1. Sebagai contoh pertama, misalkan kita ingin mencetak pesan *"Hello, world"* sebanyak 10 kali, maka algoritmanya adalah:

```
PROGRAM CetakBanyak_HelloWorld
{ Mencetak 'Hello world' sebanyak 10 kali }

DEKLARASI
  i : integer  { pencacah pengulangan }
```

```
ALGORITMA:
  for i ← 1 to 10 do    { ulangi sebanyak 10 kali }
    write('Hello, world')
  endfor
```

Algoritma 7.1 Mencetak pesan "Hello, world" sebanyak 10 kali (Versi *FOR*).

Perhatikan Algoritma 7.1 di atas:

- jumlah pengulangan sudah ditentukan sebelumnya, yaitu 10 kali, sehingga kita dapat menggunakan pernyataan *FOR*.
- badan pengulangan hanya berisi satu buah pernyataan, yaitu `write('Hello, world')`
- `i` adalah peubah pencacah yang bertipe *integer*
- pada mulanya `i` bernilai 1; nilai `i` selalu bertambah 1 setiap kali pengulangan dilakukan, sampai akhirnya i mencapai 10, yang berarti proses pengulangan selesai.
- jumlah pengulangan yang dilakukan: 10 – 1 + 1 = 10 kali

Keluaran dari Algoritma 7.1 adalah pesan "*Hello, world*" sebanyak 10 kali seperti di bawah ini (asumsikan setiap pesan ditulis pada setiap baris):

```
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
Hello, world
```

Contoh 7.2. Misalkan kita ingin mencetak angka 1, 2, ..., 8 di layar, dengan satu angka pada setiap baris. Algoritmanya sebagai berikut:

```
PROGRAM Cetak1Sampai10
{ Mencetak 1, 2, ..., 10 }

DEKLARASI
  i : integer

ALGORITMA:
  for i ← 1 to 10 do    { ulangi sebanyak 10 kali }
    write(i)
  endfor
```

Algoritma 7.2 Mencetak angka 1 sampai 10 (dengan *FOR*).

Perhatikan Algoritma 7.2 di atas. Angka-angka yang dicetak bersesuaian dengan nilai pencacah pengulangan. Ketika cacah i bernilai 1, angka 1

dicetak, ketika pencacah i bernilai 2, angka 2 dicetak. Jadi, instruksi yang diulang cukup kita tulis write(k).

Keluaran dari algoritma di atas adalah angka 1 sampai 10 seperti di bawah ini (asumsikan setiap angka ditulis pada setiap baris):

```
1
2
3
4
5
6
7
8
9
10
```

Contoh 7.3. Contoh lebih umum dari Contoh 7.2 di atas, misalkan kita ingin mencetak angka 1, 2, ..., *N*, yang dalam hal ini nilai *N* dibaca terlebih dahulu, maka algoritmanya menjadi:

```
PROGRAM CetakISampaiN
{ Mencetak 1, 2, ..., N }

DEKLARASI
  N : integer
  i : integer

ALGORITMA:
  read(N)
  for i ← 1 to N do      { ulangi sebanyak N kali }
    write(i)
  endfor
```

Algoritma 7.3 Mencetak angka 1 sampai N (versi *FOR*)

Pertanyaan:

Apa yang terjadi bila $N = 0$? $N = -1$? $N = 1$?

Jawab:

Jika $N = 0$ atau $N = -1$, proses pengulangan tidak terjadi, karena nilai akhir pencacah pengulangan lebih besar dari nilai awalnya (1).
Jika $N = 1$, pengulangan yang terjadi hanya 1 kali, karena $1 - 1 + 1 = 1$.

Contoh 7.4. Misalkan kita ingin menghitung jumlah angka-angka dari deret 1 sampai *N*, yaitu:

```
1 + 2 + 3 + ... + N
```

Nilai *N* dibaca terlebih dahulu. Misalnya, bila *N* = 5, maka jumlah angka dari 1 sampai 5 adalah deret

```
1 + 2 + 3 + 4 + 5 = 15.
```

Algoritma penjumlahan deret sebagai berikut:
Inti dari masalah ini adalah penjumlahan deret bilangan. Untuk menghitung jumlah deret, kita memerlukan sebuah peubah yang mencatat hasil penjumlahan, misalkan namanya jumlah. Sebelum penjumlahan dimulai, jumlah diinisialisasi dengan 0:

```
jumlah ← 0
```

Pada setiap kali pengulangan, kita menjumlahkan angka sekarang dengan jumlah deret. Jika angka sekarang adalah i, maka pernyataan penjumlahan adalah

```
jumlah ← jumlah + i
```

Pernyataan penjumlahan ini diulang sebanyak *N* kali, yaitu sebanyak angka yang dijumlahkan. Proses penjumlahan ini kita analogikan sebagai berikut: misalkan kita mempunyai sebuah kotak (analogi dari peubah `jumlah`). Mula-mula kotak dalam keadaan kosong (analogi dari pernyataan `jumlah←0`). Kita akan memasukkan jeruk (analogi dari `i`) ke dalam kotak secara bertahap. Dimulai dengan memasukkan 1 buah jeruk, 2 buah jeruk, dan seterusnya sampai terakhir memasukkan *N* buah jeruk. Setiap kali memasukkan `i` buah jeruk ke dalam kotak berarti menambah isi kotak sebanyak i (analog dari pernyataan `jumlah ← jumlah + i`). Karena jeruk dimasukkan berulang-ulang secara bertahap, maka ini berarti pernyaatan `jumlah ← jumlah + i` dieksekusi berulang-ulang sebanyak `N` kali. Dengan menggunakan pernyataan *FOR*, maka pengulangan tersebut ditulis menjadi:

```
for i ← 1 to N
  jumlah ← jumlah + i
endfor
```

Algoritma penjumlahan deret selengkapnya adalah:

```
PROGRAM PenjumlahanDeret
{ Menjumlahkan deret
                1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

DEKLARASI
   N : integer
   i : integer
   jumlah : integer

ALGORITMA:
    read(N)
    jumlah ← 0        { inisialisasi jumlah deret dengan 0 }
    for i← 1  to N    { ulangi penjumlahan deret sebanyak N kali }
```

```
    jumlah ← jumlah + i
  endfor
  write(jumlah)
```

Algoritma 7.4 Menjumlahkan 1 + 2 + ... + N (versi *FOR*)

Contoh 7.5. Misalkan kita ingin menghitung nilai rata-rata dari *N* buah data bilangan bulat yang dibaca dari papan ketik. Nilai rata-rata adalah jumlah seluruh data dibagi dengan banyaknya data. Misalkan jika $N = 5$ dan data bilangan yang dibaca berturut-turut adalah 12, 10, 6, 2, 4, maka nilai rata-ratanya adalah $(12 + 10 + 6 + 2 + 4)/5 = 34/5 = 6.8$.

Inti dari masalah ini adalah membaca data dan menjumlahkannya. Proses membaca dan menjumlahkan dilakukan sebanyak *N* kali. Cara penjumlahannya sama seperti pada Contoh 7.4. Jika pencatat hasil penjulahan adalah peubah `jumlah` dan data yang dibaca adalah `x`, maka tambahkan `x` ke `jumlah` dengan pernyataan:

```
jumlah ← jumlah + x
```

Algoritma selengkapnya sebagai berikut:

```
PROGRAM HitungRataRata
{ Menghitung rata-rata N buah bilangan bulat yang dibaca dari papan
ketik. N > 0}

DEKLARASI
  N : integer        { jumlah data, > 0 }
  x : integer        { bilangan bulat yang dibaca dari papan ketik }
  i : integer        { pencacah banyaknya pengulangan }
  jumlah : integer   { pencatat jumlah nilai }
  rerata : real      { rata-rata nilai }

ALGORITMA:
  read(N)
  jumlah ← 0         { inisialisasi penjumlah seluruh nilai }
  for i ← 1 to N do
    read(x)
    jumlah ← jumlah + x
  endfor
  rerata ← jumlah/N
  write(rerata)
```

Algoritma 7.5 Menjumlahkan N buah bilangan bulat yang dibaca dari papan kunci

Pencacah pengulangan tidak harus bertipe *integer*. Tipe lain pun dapat dijadikan sebagai pencacah asal tipe tersebut memiliki keterurutan (ada *predecessor* dan *successor* - lihat Bab 3 pada bagian penjelasan tipe *integer* dan *char*). Bilangan bulat dan karakter adalah tipe data yang memiliki *predecessor* dan *successor*. Bilangan riil tidak mempunyai *predecessor* dan *successor* karena itu ia tidak dapat dipakai sebagai pencacah.

Contoh pengulangan dengan pencacah bukan bilangan bulat misalnya:

```
for i ← 'a' to 'm' do
   write(i)
endfor
```

yang dalam hal ini, sebagai pencacah pengulangan adalah i yang bertipe karakter.

Adakalanya kita melakukan pengulangan dengan cacah yang menurun, yaitu dari nilai cacah yang besar ke nilai cacah yang kecil. Untuk memfasilitasi hal ini, maka kita mendefinisikan struktur *FOR* menurun.

***FOR* menurun:**

```
for pencacah ← nilai_akhir downto nilai_awal do
  aksi
endfor
```

Keterangan:

(i) *pencacah* haruslah dari tipe data yang memiliki *predecessor* dan *successor*, yaitu *integer* atau karakter. Tipe riil tidak dapat digunakan sebagai pencacah.
(ii) *aksi* dapat berupa satu atau lebih instruksi yang diulang
(iii) *nilai_akhir* harus lebih besar atau sama dengan nilai_awal. Jika *nilai_akhir* lebih kecil dari *nilai_awal*, maka badan pengulangan tidak dimasuki.
(iv) Pada awalnya, pencacah diinisialisasi dengan *nilai_akhir*. Nilai *pencacah* secara otomatis berkurang satu setiap kali badan pengulangan dimasuki, sampai akhirnya nilai *pencacah* sama dengan *nilai_awal*.
(v) Jumlah pengulangan yang terjadi adalah *nilai_awal* – *nilai_akhir* + 1.

Contoh 7.6. Sebuah roket diluncurkan dengan hitungan mundur, mulai dari 100, 99, 98, ..., 0. Algoritma perhitungan mundur adalah:

```
PROGRAM PeluncuranRoket
{ Hitung mundur peluncuran roket }

DEKLARASI
  i : integer

ALGORITMA:
  for i ← 100 downto 0 do
    write(i)
  endfor
  write('Go!')   { roket meluncur pada saat hitungan 0}
```

Algoritma 7.6 Perhitungan mundur untuk peluncuran roket (versi *FOR*).

Kita dapat membuat kelambatan waktu (*delay*) antara satu hitungan dan hitungan selanjutnya dengan penggunaan *FOR* bersarang (*nested FOR*) seperti pada algoritma di bawah ini:

```
PROGRAM PeluncuranRoket
{ Hitung mundur peluncuran roket }
```

```
DEKLARASI
  i, p : integer

ALGORITMA:
  for i ← 100 downto 0 do
    { delay }
    for p ← 1 to 1000 do
      { kosong, tidak melakukan apa-apa }
    endfor

    write(i)
  endfor
  write('Go!')   { roket meluncur pada hitungan 0 }
```

Algoritma 7.7 Perhitungan mundur untuk peluncuran roket dengan *delay*.

FOR terluar "diputar" sebanyak 101 kali, sedangkan *FOR* terdalam "diputar" sebanyak 1000 kali. Untuk setiap "putaran" *FOR* terluar, *FOR* terdalam dieksekusi sebanyak 1000 kali. Jika setiap putaran *FOR* terdalam membutuhkan waktu *a* detik, maka 1000 putaran membutuhkan waktu 1000*a* detik. Inilah yang disebut dengan *delay*.

7.3 Pernyataan *WHILE*

Bentuk umum pernyataan *WHILE* adalah:

```
while kondisi do
  aksi
endwhile
```

Keterangan:
aksi akan dilaksanakan berulangkali selama *kondisi* bernilai *true*. Jika *kondisi* bernilai *false*, badan pengulangan tidak akan dimasuki, yang berarti pengulangan selesai. Kondisi di akhir pengulangan (setelah *endwhile*) disebut *loop invariant*, yaitu variabel kondisi yang nilainya sudah tidak berubah lagi.

Yang harus diperhatikan adalah pengulangan harus berhenti. Pengulangan yang tidak pernah berhenti menandakan bahwa logika algoritma tersebut salah. Pengulangan berhenti apabila *kondisi* bernilai *false*. Agar *kondisi* suatu saat bernilai *false*, maka di dalam badan pengulangan harus ada instruksi yang mengubah nilai *kondisi*. Kita memakai kembali enam contoh pengulangan yang sudah diberikan pada pembahasan pernyataan *FOR* untuk menggambarkan penggunaan pernyataan *WHILE*.

1) Mencetak pesan "*Hello, world*" sebanyak 10 kali

```
PROGRAM CetakBanyak_HelloWorld
{ Mencetak 'Hello world' sebanyak 10 kali }

DEKLARASI
```

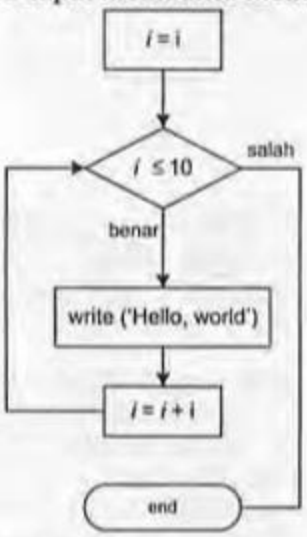
```
  i : integer  { pencacah pengulangan }
ALGORITMA:
  i ← 1
  while i ≤ 10 do   { ulangi sebanyak 10 kali }
    write('Hello, world')
    i ← i + 1
  endwhile
  { i > 10  → loop invariant }
```

Algoritma 7.8 Mencetak pesan "Hello, world" sebanyak 10 kali (versi *WHILE*).

Perhatikan Algoritma 7.8 di atas:

- badan pengulangan berisi dua instruksi: `write('Hello world')` dan `i←i+1`
- kondisi pengulangan adalah `i ≤ 10`, artinya selama `i` masih ≤ 10, maka badan pengulangan boleh dimasuki.
- `i` harus terdefinisi nilainya sebelum pengulangan dilaksanakan pertama kali. Karena itu, pada bagian inisialisasi, `i` diisi dengan nilai 1.
- instruksi yang mengubah nilai peubah kondisi adalah pernyataan `i←i+1`. Pernyataan ini mencacah jumlah pengulangan, sehingga jika `i > 10`, pengulangan dihentikan.
- pada akhir pengulangan, nilai `i > 10`. Ini adalah kondisi *loop invariant*.

Bagan alir pada Gambar 7.1 dapat membantu memahami Algoritma 7.8.



Gambar 7.1 Bagan alir yang memperlihatkan penggunaan struktur *WHILE*. Perhatikan bahwa kondisi selalu dipeiksa di awal pengulangan.

Kesalahan yang sering muncul adalah melupakan inisialisasi. Misalkan pada Algoritma 7.8 di atas pemrogram mungkin lupa menuliskan pernyataan `i←1` seperti pada algoritma bawah ini:

```
PROGRAM CetakBanyak_HelloWorld
{ Mencetak 'Hello, world' sebanyak 10 kali }

DEKLARASI
  i : integer  { pencacah pengulangan }

ALGORITMA:
  while i ≤ 10 do
    write('Hello, world')
    i ←i + 1
  endwhile
 { i > 10 }
```

Karena nilai i tidak terdefinisi sebelumnya, maka proses pengulangan mungkin tidak pernah dilaksanakan (dikatakan "mungkin" karena pada bahasa pemrograman tertentu, bila sebuah peubah tidak diinisialisasi, maka peubah tersebut sudah otomatis berisi sebuah nilai tertentu. Sebagai contoh, di dalam bahasa *C*, peubah numerik (*int* atau *float*) yang tidak diisi nilai apa pun, maka nilai *default* peubah tersebut adalah 0, sedangkan di dalam bahasa *Pascal*, peubah *integer* yang tidak diisi nilai apa pun, akan diisi oleh komputer dengan nilai sembarang. Jadi, jika nilai awal dari sebuah peubah tidak memenuhi kondisi pengulangan, maka pengulangan tidak pernah dimasuki).

Kesalahan yang lain adalah tidak menuliskan instruksi yang mengubah peubah kondisi, misalnya tidak menuliskan pernyataan i←i+1 sebagaimana pada algoritma berikut ini:

```
PROGRAM CetakBanyak_HelloWorld
{ Mencetak 'Hello world' sebanyak 10 kali }

DEKLARASI
  i : integer  { pencacah pengulangan }

ALGORITMA:
  i←1          { inisialisasi }
  while i ≤ 10 do
    write('Hello world')
  endwhile
 { i > 10 }
```

Pada algoritma di atas, nilai i selalu 1. Karena nilai i tidak pernah berubah di dalam badan pengulangan, maka pada setiap kali pengulangan kondisi i ≤ 10 selalu tetap benar. Akibatnya, pengulangan *WHILE* di atas tidak pernah berhenti. Pengulangan yang tidak pernah berhenti disebut *looping*.

2) Mencetak angka-angka 1, 2, ..., 10

```
PROGRAM CetakISampai10
{ Mencetak 1, 2, ..., 10. }

DEKLARASI
  k : integer
```

```
ALGORITMA:
  i ← 1          { inisialisasi }
  while i ≤ 10 do
    write(i)
    i ← i + 1
  endwhile
  { i > 10 }
```

Algoritma 7.9 Mencetak angka 1 sampai 10 (versi *WHILE*).

3) Mencetak angka 1 sampai *N*; nilai *N* dibaca terlebih dahulu (*N* > 0)

```
PROGRAM CetakISampaiN
{ Mencetak 1, 2, ..., N }

DEKLARASI
  N : integer
  i : integer

ALGORITMA:
  read(N)
  i←1
  while i ≤ N do      { ulangi sebanyak N kali }
    write(i)
    i ← i + 1
  endwhile
  { i > N }
```

Algoritma 7.10 Mencetak angka 1 sampai N (versi *WHILE*)

4) Menghitung 1 + 2 + 3 + ... + *N* (*N* > 0)

```
PROGRAM PenjumlahanDeret
{ Menjumlahkan deret
          1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

DEKLARASI
   N : integer        { banyaknya suku deret, > 0 }
   i : integer        { suku deret }
   jumlah : integer   { jumlah deret }

ALGORITMA:
    read(N)       { banyaknya suku deret }
    jumlah ← 0    { inisialisasi jumlah deret }
    i ← 1         { suku deret yang pertama }
    while i ≤ N do
      jumlah ← jumlah + i     { jumlah deret sekarang }
      i ← i + 1               { suku deret berikutnya }
    endwhile
    { i > N }   { kondisi setelah pengulangan berhenti }

    write(jumlah)
```

Algoritma 7.11 Menjumlahkan 1 + 2 + ... + N (versi *WHILE*).

5) Menghitung nilai rata-rata *N* buah bilangan bulat yang dibaca dari papan ketik

```
PROGRAM HitungRataRata
{ Menghitung rata-rata N buah bilangan bulat yang dibaca dari papan
ketik, N > 0. }

DEKLARASI
  N : integer        { banyaknya data, > 0 }
  x : integer        { data yang dibaca dari papan ketik }
  i : integer        { pencacah banyak data }
  jumlah : integer   { pencatat jumlah data }
  rerata : real      { nilai rata-rata seluruh data }

ALGORITMA:
  read(N)
  jumlah ← 0    { inisialisasi }
  i ← 1         { inisialisasi pencacah }
  while i ≤ N do
    read(x)
    jumlah ← jumlah + x
    i ← i + 1
  endwhile
  { i > N }
  rerata ← jumlah/N
  write(rerata)
```

Algoritma 7.12 Menghitung rata-rata N buah data *integer* (versi *WHILE*).

6) Algoritma peluncuran roket dengan hitung mundur

```
PROGRAM PeluncuranRoket
{ Hitung mundur peluncuran roket }

DEKLARASI
  i : integer

ALGORITMA:
  i←100
  while i ≥ 0 do
    write(i)
    i ← i - 1
  endwhile
  { i < 0 }
  write('Go!')   { roket meluncur pada hitungan 0}
```

Algoritma 7.13 Perhitungan mundur untuk peluncuran roket (versi *WHILE*).

Pada kasus-kasus di mana jumlah pengulangan diketahui di awal program, *WHILE* dapat digunakan sebaik penggunaan *FOR*, seperti pada enam contoh di atas. Namun, untuk proses yang jumlah pengulangannya tidak dapat ditentukan di awal, hanya struktur *WHILE* yang dapat kita gunakan, sebab kondisi pengulangan diperiksa di awal pengulangan. Jadi, meskipun kita

tidak mengetahui kapan persisnya *WHILE* ini berhenti, tetapi kita menjamin bahwa jika kondisi bernilai salah, maka pengulangan pasti berhenti.

Untuk lebih memperjelas, misalkan kita akan menghitung nilai rata-rata sekumpulan data bilangan bulat yang dibaca dari papan ketik. Banyaknya data tidak diketahui di awal, namun proses pembacaan data dianggap berakhir jika nilai yang dibaca adalah 0. Untuk masalah ini jelas kita tidak dapat menggunakan *FOR* karena jumlah pengulangan tidak diketahui. *WHILE* dapat digunakan untuk pengulangan semacam ini karena proses pembacaan data dilakukan selagi data yang dibaca tidak sama dengan 0. Algoritmanya seperti di bawah ini:

```
PROGRAM HitungRataRata2
{ Menghitung rata-rata dari sejumlah data bilangan bulat yang dibaca
dari papan ketik selama data yang dibaca tidak sama dengan 0. }

DEKLARASI
  x : integer        { data bilangan yang dibaca dari papan ketik }
  i : integer        { pencacah banyak data }
  jumlah : integer   { pencatat jumlah data }
  rerata : real      { nilai rata-rata seluruh data }

ALGORITMA:
  i ← 0              { inisialisasi pencatat banyak data dengan 0}
  jumlah ← 0         { inisialisasi jumlah data dengan 0}
  read(x)
  while x ≠ 0 do   { lakukan penjumlahan selama x tidak nol }
    i ← i + 1          { naikkan pencatat banyaknya data }
    jumlah ← jumlah + x
    read(x)
  endwhile
  { x = 0 }
  if  i ≠ 0 then  { data yang dibaca minimal 1 buah }
    rerata ← jumlah/i
    write(rerata)
  else
    write('tidak ada data yang dimasukkan')
  endif
```

Algoritma 7.14. Menghitung rata-rata dari sejumlah data (penanda akhir data = 0).

7.4 Pernyataan *REPEAT*

Bentuk umum pernyataan *REPEAT* adalah:

```
repeat
   aksi
until kondisi
```

Penjelasan:

Notasi ini mendasarkan pengulangan pada kondisi *boolean*. *Aksi* di dalam badan kalang diulang-ulang sampai *kondisi* bernilai *true*. Dengan kata lain, jika *kondisi* masih *false*, proses pengulangan masih terus dilakukan. Karena

proses pengulangan suatu saat harus berhenti, maka di dalam badan pengulangan harus ada pernyataan yang mengubah nilai peubah kondisi.

Pernyataan *REPEAT* memiliki makna yang sama dengan *WHILE*, dan dalam beberapa masalah kedua pernyataan tersebut komplemen satu sama lain.

Kita memakai kembali enam contoh pengulangan yang sudah diberikan pada pembahasan pernyataan *FOR* untuk menggambarkan penggunaan pernyataan *REPEAT*.

1) Mencetak pesan "*Hello, world*" sebanyak 10 kali

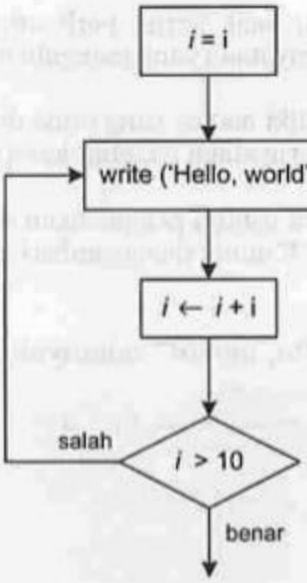
```
PROGRAM CetakBanyak_HelloWorld
{ Mencetak 'Hello world' sebanyak 10 kali }

DEKLARASI
  i : integer  { pencacah pengulangan }

ALGORITMA:
  i ← 1
  repeat   { ulangi sebanyak 10 kali }
    write('Hello, world')
    i ← i + 1
  until i > 10
```

Algoritma 7.15 Mencetak pesan "Hello, world" sebanyak 10 kali (versi *REPEAT*).

Perhatikan Algoritma 7.15 di atas:

- badan pengulangan berisi dua instruksi: `write('Hello world')` dan `i ← i + 1`
- kondisi berhenti pengulangan adalah bila `i > 10`, artinya selama `i` masih ≤ 10, maka badan pengulangan boleh dimasuki.
- `i` harus terdefinisi nilainya sebelum pengulangan dilaksanakan pertama kali. Karena itu, pada bagian inisialisasi, `i` diisi dengan nilai 1.
- instruksi yang mengubah nilai peubah kondisi adalah pernyataan `i←i+1`. Pernyataan ini mencacah jumlah pengulangan, sehingga jika `i > 10`, pengulangan dihentikan.
- pada akhir pengulangan, nilai `i > 10`.

Bagan alir pada Gambar 7.2 dapat membantu memahami penggunaan struktur *REPEAT* pada Algoritma 7.15. Perhatikan di mana perbedaannya dengan Gambar 7.1?



Gambar 7.2 Bagan alir yang memperlihatkan penggunaan struktur *REPEAT*. Perhatikan bahwa kondisi selalu dipeiksa di akhir pengulangan.

2) Mencetak angka-angka 1, 2, ..., 10

```
PROGRAM Cetak1Sampai10
{ Mencetak 1, 2, ..., 10. }

DEKLARASI
  i : integer

ALGORITMA:
  i ← 1          { inisialisasi }
  repeat
    write(i)
    i ← i + 1
  until i > 10
```

Algoritma 7.16 Mencetak angka 1 sampai 10 (versi *REPEAT*).

3) Mencetak angka 1 sampai N ($N > 0$)

```
PROGRAM Cetak1SampaiN
{ Mencetak 1, 2, ..., N }

DEKLARASI
  N, i : integer

ALGORITMA:
  read(N)
  i ← 1
  repeat      { ulangi sebanyak N kali }
```

```
    write(i)
    i ← i + 1
  until i > N
```

Algoritma 7.17 Mencetak angka 1 sampai N (versi *REPEAT*)

4) Menghitung 1 + 2 + 3 + ... + *N* (*N* > 0)

```
PROGRAM PenjumlahanDeret
{ Menjumlahkan deret
                1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih
dahulu.}

DEKLARASI
   N : integer        { banyaknya suku deret, > 0 }
   i : integer        { suku deret }
   jumlah : integer   { jumlah deret }

ALGORITMA:
    read(N)         { banyaknya suku deret }
    jumlah ← 0   { inisialisasi jumlah deret }
    i ← 1        { suku deret yang pertama }
    repeat
      jumlah ← jumlah + i      { jumlah deret sekarang }
      i ← i + 1               { suku deret berikutnya }
    until i > N

    write(jumlah)
```

Algoritma 7.18 Menjumlahkan 1 + 2 + ... + N (versi *REPEAT*).

5) Menghitung nilai rata-rata *N* buah bilangan bulat yang dibaca dari papan ketik

```
PROGRAM HitungRataRata
{ Menghitung rata-rata N buah bilangan bulat yang dibaca dari papan
ketik. N > 0. }

DEKLARASI
  N : integer        { banyaknya data, > 0 }
  x : integer        { data bilangan bulat yang dibaca dari papan ketik
}
  i : integer        { pencacah banyak data }
  jumlah : integer   { pencatat jumlah data }
  rerata : real      { nilai rata-rata seluruh data }

ALGORITMA:
  read(N)
  jumlah ← 0     { inisialisasi }
  i ← 1         { inisialisasi pencacah }
  repeat
    read(x)
    jumlah ← jumlah + x
```

```
    i ← i + 1
  until i > N
  rerata ← jumlah/N
  write(rerata)
```

Algoritma 7.19 Menghitung rata-rata N buah data (versi *WHILE*).

6) Algoritma peluncuran roket dengan hitung mundur

```
PROGRAM PeluncuranRoket
{ Hitung mundur peluncuran roket }

DEKLARASI
  i : integer

ALGORITMA:
  i←100
  repeat
    write(i)
    i ← i - 1
  until i < 0
  write('Go!')   { roket meluncur pada hitungan 0}
```

Algoritma 7.20 Perhitungan mundur untuk peluncuran roket (versi *REPEAT*).

7.5 *WHILE* atau *REPEAT*?

Contoh-contoh pada upabab 7.4 dan 7.5 memperlihatkan bahwa konstruksi *WHILE* dan *REPEAT* ekivalen. Kita dapat menggunakan *WHILE* maupun *REPEAT*. Keduanya benar untuk masalah-masalah tertentu. Tetapi, pada beberapa masalah, pemilihan *WHILE* atau *REPEAT* bergantung pada natural dari masalah itu sendiri. Ini artinya, ada masalah yang hanya benar bila menggunakan struktur *WHILE*, tetapi bisa fatal bila menggunakan *REPEAT*.

Untuk mengetahui struktur mana yang lebih tepat, kita harus mengetahui perbedaan mendasar di antara keduanya. Perbedaannya adalah sebagai berikut: pada konstruksi *REPEAT*, kondisi pengulangan diperiksa pada akhir pengulangan. Jadi, instruksi di dalam badan pengulangan dilaksanakan dulu, barulah pengetesan kondisi dilakukan. Konsekuensinya, badan pengulangan dilaksanakan paling sedikit satu kali. Sebaliknya, pada konstruksi *WHILE*, kondisi pengulangan diperiksa di awal pengulangan. Jadi, instruksi di dalam badan pengulangan hanya dapat dilaksanakan bila pengetesan kondisi menghasilkan nilai *true*. Konsekuensinya, badan pengulangan mungkin tidak akan pernah dilaksanakan bila kondisi pengulangan pertama kali bernilai *false*.

Berdasarkan perbedaan di atas, maka kita dapat menarik kesimpulan kapan menggunakan *WHILE* dan kapan menggunakan *REPEAT*:

⇒ Gunakan konstruksi *WHILE* pada kasus yang mengharuskan terlebih dahulu pemeriksaan kondisi objek sebelum objek tersebut dimanipulasi.

⇒ Gunakan konstruksi *REPEAT* pada kasus yang terlebih dahulu memanipulasi objek, baru kemudian memeriksa kondisi objek tersebut.

Contoh-contoh berikut memperlihatkan perbedaan pemakaian *WHILE* dan *REPEAT*.

Contoh (a): Penggunaan *WHILE* benar, tetapi penggunaan *REPEAT* salah

Dibaca sejumlah nilai bilangan bulat (x) dari proses masukan. Kita diminta menghitung jumlah seluruh nilai pecahannya ($1/x$). Tanda akhir pembacaan data adalah bila nilai x yang dibaca = 0. Sebagai contoh, jika nilai x yang dibaca berturut-turut adalah 3, 5, 8, dan 0, maka jumlah seluruh nilai pecahannya adalah

$$1/3 + 1/5 + 1/8 = 0.65833$$

Misalkan jumlah deret tersebut adalah S. Sebelum penjumlahan, S kita inisialisasi dengan 0. Setiap kali membaca x, harus diperiksa apakah $x = 0$. Jika $x = 0$, maka proses pengulangan berhenti, tetapi jika tidak, maka jumlahkan S dengan $1/x$:

$$S \leftarrow S + 1/x$$

***WHILE* (benar)**

```
PROGRAM PenjumlahanDeretPecahan
{ Menghitung jumlah seluruh nilai pecahan dari sejumlah data bilanagn
bula (x) yang dibaca dari papan ketik. Akhir pembacaan adalah bila
nilai x yang dibaca = 0. }

DEKLARASI
   x : integer    { data bilangan bulat yang dibaca }
   S : real       { jumlah deret }

ALGORITMA:
   S←0    { inisialisasi jumlah deret }
   read(x)

   while x ≠ 0 do
     S ← S + 1/x
     read(x)
   endwhile
   { x = 0 }      { kondisi setelah pengulangan berhenti }
   write(S)
```

Algoritma 7.21 Menjumlahkan deret pecahan (benar).

Komentar:
Penggunaan *WHILE* pada Algoritma 7.21 benar, karena sebelum dijumlahkan, x diperiksa di awal pengulangan apakah sudah 0 atau belum 0.

REPEAT (salah)

```
PROGRAM PenjumlahanDeretPecahan
{ Menghitung jumlah seluruh nilai pecahan dari sejumlah data bilanagn
bula (x) yang dibaca dari papan ketik. Akhir pembacaan adalah bila
nilai x yang dibaca = 0. }

DEKLARASI
   x : integer    { harga bilangan bulat yang dibaca }
   S : real       { jumlah deret }

ALGORITMA:
   S ← 0     { inisialisasi jumlah deret }
   repeat
     read(x)
     S ← S + 1/x
   until x = 0

   write(S)
```

Algoritma 7.22 Menjumlahkan deret pecahan (salah).

Komentar:
Penggunaan struktur *REPEAT* pada Algoritma 7.22 berakibat sangat fatal bila x yang pertama kali dibaca berharga nol sebab pembagian dengan nol (1/0) tidak terdefinisi (*error*).

Contoh (b): *REPEAT* benar, *WHILE* salah.

Tinjau kembali algoritma menampilkan menu pada pembahasan konstruksi *CASE* di dalam Bab 6.

```
PROGRAM SimulasiMenuProgram
{ Menampilkan menu, membaca pilihan menu, dan menampilkan nomor menu
yang dipilih oleh pengguna }

DEKLARASI
  NomorMenu : integer

ALGORITMA:
  { Cetak menu  }
  write('      MENU          ')
  write(' 1. Baca data      ')
  write(' 2. Cetak data     ')
  write(' 3. Ubah data      ')
  write(' 4. Hapus data     ')
  write(' 5. Keluar program ')
  write(' Masukkan pilihan anda(1/2/3/4/5)? ')

  read(NomorMenu)    { baca nomor menu yang akan dipilih }

  case NomorMenu
```

```
    1 : write('Anda memilih menu nomor 1')
    2 : write('Anda memilih menu nomor 2')
    3 : write('Anda memilih menu nomor 3')
    4 : write('Anda memilih menu nomor 4')
    5 : write('Keluar program...')
  endcase
```

Algoritma `SimulasiMenuProgram` di atas tidak dapat digunakan untuk memilih menu secara berulang-ulang. Kita menginginkan dapat memilih menu manapun berkali-kali sampai menu yang kita pilih adalah menu nomor 5. Bagaimana hal ini dapat dilakukan? Menggunakan *REPEAT* atau *WHILE*? Perhatikan masing-masing algoritma dengan *REPEAT* dan *WHILE* di bawah ini.

REPEAT (benar)

```
PROGRAM SimulasiMenuProgram
{ Menampilkan menu, membaca pilihan menu, dan menampilkan nomor menu
yang dipilih oleh pengguna }

DEKLARASI
  NomorMenu : integer

ALGORITMA:
  repeat
    { Cetak menu }
    write('      MENU          ')
    write(' 1. Baca data       ')
    write(' 2. Cetak data      ')
    write(' 3. Ubah data       ')
    write(' 4. Hapus data      ')
    write(' 5. Keluar program  ')
    write(' Masukkan pilihan anda(1/2/3/4/5)? ')

    read(NomorMenu)    { baca nomor menu yang akan dipilih }
    case NomorMenu
      1 : write('Anda memilih menu nomor 1')
      2 : write('Anda memilih menu nomor 2')
      3 : write('Anda memilih menu nomor 3')
      4 : write('Anda memilih menu nomor 4')
      5 : write('Keluar program...')
    endcase

  until NomorMenu = 5
```

Algoritma 7.23 Simulasi menu program dengan pengulangan (benar)

Keterangan:
Konstruksi *REPEAT* benar untuk masalah ini, sebab nomor pilihan menu dibaca terlebih dahulu, barulah kemudian diperiksa di akhir pengulangan apakah nomor menu yang dipilih = 5.

WHILE (salah)

```
PROGRAM SimulasiMenuProgram
{ Menampilkan menu, membaca pilihan menu, dan menampilkan nomor menu
yang dipilih oleh pengguna }

DEKLARASI
  NomorMenu : integer

ALGORITMA:
  while NomorMenu ≠ 5 do
    { Cetak menu }
    write('      MENU          ')
    write(' 1. Baca data      ')
    write(' 2. Cetak data     ')
    write(' 3. Ubah data      ')
    write(' 4. Hapus data     ')
    write(' 5. Keluar program ')
    write(' Masukkan pilihan anda(1/2/3/4/5)? ')

    read(NomorMenu)    { baca nomor menu yang akan dipilih }

    case NomorMenu
      1 : write('Anda memilih menu nomor 1')
      2 : write('Anda memilih menu nomor 2')
      3 : write('Anda memilih menu nomor 3')
      4 : write('Anda memilih menu nomor 4')
      5 : write('Keluar program...')
    endcase

  endwhile
  { NomorMenu = 5 }
```

Algoritma 7.24 Simulasi menu program dengan pengulangan (salah)

Keterangan:
Penggunaan *WHILE* pada masalah ini salah karena `NomorMenu` belum terdefinisi pada pengulangan pertama kali, padahal nilai `NomorMenu` baru didefinisikan di dalam kalang.

Agar penggunaan *WHILE* pada Algoritma 7.24 benar, maka di dalam algoritma tersebut harus ditambahkan sebuah peubah *boolean*, misalkan `stop`. Peubah `stop` ini mula-mula diinisialisasi dengan `false`

```
stop ← false
```

Jika menu yang dipilih nomor 5, maka pada kasus `NomorMenu = 5` ditambahkan pernyataan berikut:

```
stop ← true
```

Pengetesan kondisi pada awal pengulangan menjadi

```
    while not stop do
      ...
```

Algoritma perbaikannya menjadi sebagai berikut:

```
PROGRAM SimulasiMenuProgram
{ Menampilkan menu, membaca pilihan menu, dan menampilkan nomor menu
yang dipilih oleh pengguna }

DEKLARASI
  NomorMenu : integer
  stop : boolean

ALGORITMA:
  stop ← false
  while not stop do

    { Cetak menu }
    write('      MENU          ')
    write(' 1. Baca data      ')
    write(' 2. Cetak data     ')
    write(' 3. Ubah data      ')
    write(' 4. Hapus data     ')
    write(' 5. Keluar program ')
    write(' Masukkan pilihan anda(1/2/3/4/5)? ')
    read(NomorMenu)    { baca nomor menu yang akan dipilih }
    case NomorMenu
      1 : write('Anda memilih menu nomor 1')
      2 : write('Anda memilih menu nomor 2')
      3 : write('Anda memilih menu nomor 3')
      4 : write('Anda memilih menu nomor 4')
      5 : write('Anda memilih menu nomor 5')
          stop←true
    endcase

  endwhile
  { NomorMenu = 5 }
```

Algoritma 7.25 Simulasi menu program dengan pengulangan (benar)

Contoh (c): menghitung upah karyawan

Algoritma menghitung upah karyawan pada Contoh 6.11 hanya menghitung upah seorang karyawan saja. Agar dapat digunakan untuk menghitung upah beberapa orang karyawan, maka kita harus menambahkan struktur pengulangan di dalam algoritmanya. Andaikan jumlah karyawan tidak diketahui sebelum eksekusi program, maka jumlah pengulangan tidak dapat ditentukan. Namun, pengulangan perhitungan upah untuk karyawan yang lain dilanjutkan jika pemakai program menjawab pertanyaan:

```
write('Ulangi untuk karyawan yang lain?(y/t) ')
read(jawab)
```

dan memberikan jawaban 'y' (artinya: ya). Jika jawabnya 't' (artinya; tidak) maka proses pengulangan dihentikan. Jadi, proses perhitungan upah karyawan diulang sampai jawaban yang diberikan pengguna = 't'. Karena konfirmasi pengulangan dilakukan di akhir, maka konstruksi pengulangan yang tepat adalah dengan mengguanken pernyataan *REPEAT*.

```
PROGRAM Upah_Karyawan
{ Menentukan upah mingguan seorang karyawan. Masukan yang dibaca dari
papan kunci adalah nama karyawan, golongan, dan jumlah jam kerja.
Keluaran program adalah nama karyawan dan upahnya. }

DEKLARASI
   const JamNormal  = 48      { jumlah jam kerja normal per minggu }
   const UpahPerJam = 2000    { upah per jam, Rp 2000 }
   const UpahLembur = 3000    { upah per jam lembur, Rp 3000 }

   nama   : string   { nama karyawan }
   JJK    : integer  { Jumlah Jam Kerja }
   lembur : real     { jumlah jam lembur }
   upah   : real     { upah karyawan }
   jawab  : char

ALGORITMA:
   repeat
     read(nama,JJK)
     if JJK ≤ JamNormal then
        upah ← JJK * UpahPerJam
     else
        lembur ← JJK - JamNormal
        upah ← JamNormal * UpahPerJam + lembur * UpahLembur
     endif
     write(nama,upah)
     write('Ulangi untuk karyawan yang lain?(y/t)')
     read(jawab)
   until jawab = 't'
```

Algoritma 7.26 Menentukan honor karyawan dengan kemungkinan lembur

7.6 Tabel Translasi Notasi Algoritmik Struktur Pengulangan ke Notasi Pascal dan C

Struktur	Algoritma	Pascal	C

FOR	FOR menaik:
	
	for *pencacah*←*a* to *b* do
	  aksi
	endfor
		FOR menaik:
		
		for *pencacah*:=*a* **to** *b* **do**
		  aksi;
		*(*endfor*)*
		
		Keterangan:
		Bila aksi lebih dari satu buah, maka runtunan aksi diawali dengan **begin** dan **end**.
		
		for *pencacah*:=*a* **to** *b*
			FOR menaik:
			
			for(*pencacah* = *a*;*a*<=*b*;*a*++)
			  aksi;
			*/*endfor*/*
			
			Keterangan:
			Bila aksi lebih dari satu buah, maka runtunan aksi diawali dengan "{" dan "}".
			
			for(*pencacah* =

		do
		begin
		aksi 1;
		aksi 2;
		.
		.
		.
		end; *(*for*)*
			a;a<=b;a++)
			{
			aksi 1;
			aksi 2;
			.
			.
			.
			}

	FOR menurun:
	
	for *pencacah←a* downto *b* do
	aksi
	endfor
		FOR menurun:
		
		for *pencacah:=a*
		downto *b* **do**
		aksi;
		*(*endfor*)*
		
		Keterangan:
		Bila aksi lebih dari satu buah, maka runtunan aksi diawali dengan **begin** dan **end**.
		
		for *pencacah:=a* **to** *b*
		do
		begin
		aksi 1;
		aksi 2;
		.
		.
		.
		end; *(*for*)*
			FOR menurun:
			
			for *(pencacah=*
			a;a<=b;a++)
			aksi;
			*/*endfor*/*
			
			Keterangan:
			Bila aksi lebih dari satu buah, maka runtunan aksi diawali dengan "{" dan "}".
			
			for *(pencacah=a;a<=b;a++)*
			{
			aksi 1;
			aksi 2;
			.
			.
			.
			}

WHILE	while *kondisi* do
	aksi
	endwhile
		while *kondisi* **do**
		aksi;
		*(*endwhile*)*
		
		Keterangan:
		Bila aksi lebih dari satu buah, maka runtunan aksi diawali dengan **begin** dan **end**.
		
		while *kondisi* **do**
		begin
		aksi 1;
		aksi 2;
		.
		.
		.
		end; *(*while*)*
			while *(kondisi)*
			aksi
			*/*endwhile*/*
			
			Keterangan:
			Bila aksi lebih dari satu buah, maka runtunan aksi diawali dengan "{"dan "}".
			
			while(*kondisi*)
			{
			aksi 1;
			aksi 2;
			.
			.
			.
			}

REPEAT	repeat *aksi* until *kondisi*	**repeat** *aksi;* **until** *kondisi;* Keterangan:	**do** { *aksi* }**while** *(!kondisi);* Keterangan:

		Bila aksi lebih dari satu buah, maka runtunan aksi *tidak perlu* diawali dengan **begin** dan **end**.	Bila aksi lebih dari satu buah, maka runtunan aksi diawali dengan **begin** dan **end**. **do** { *aksi 1*; *aksi 2*; . . . }**while** (*kondisi*)

Contoh-contoh Translasi:

1. Algoritma menghitung jumlah deret $1 + 2 + 3 + ... + N$.

ALGORITMIK:

a. *FOR*

```
PROGRAM PenjumlahanDeret
{ Menjumlahkan deret
                1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

DEKLARASI
   N : integer
   i : integer
   jumlah : integer

ALGORITMA:
   read(N)
   jumlah ← 0        { inisialisasi jumlah deret dengan 0 }
   for i ← 1 to N    { ulangi penjumlahan deret sebanyak N kali }
     jumlah ← jumlah + i
   endfor
   write(jumlah)
```

b. *WHILE*

```
PROGRAM PenjumlahanDeret
{ Menjumlahkan deret
                1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

DEKLARASI
   N : integer
   i : integer
   jumlah : integer

ALGORITMA:
   read(N)
```

```
    jumlah ← 0        { inisialisasi jumlah deret dengan 0 }
    i ← 1
    while i ≤ N do    { ulangi penjumlahan deret sebanyak N kali }
      jumlah ← jumlah + i
      i ← i + 1
    endwhile
    { i > N }
    write(jumlah)
```

c. *REPEAT*

```
PROGRAM PenjumlahanDeret
{ Menjumlahkan deret
                1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

DEKLARASI
   N : integer
   i : integer
   jumlah : integer

ALGORITMA:
    read(N)
    jumlah ← 0     { inisialisasi jumlah deret dengan 0}
    i ← 1
    repeat       { ulangi penjumlahan deret sebanyak N kali }
      jumlah ← jumlah + i
      i ← i + 1
    until i > N

    write(jumlah)
```

PASCAL:

a. *FOR*

```
program PenjumlahanDeret;
{ Menjumlahkan deret
                1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

(*DEKLARASI*)
var
   N : integer;
   i : integer;
   jumlah : integer;

(*ALGORITMA:*)
begin
    write('Berapa N?'); readln(N);     { banyaknya suku deret }
    jumlah := 0;          { inisialisasi jumlah deret dengan 0}
    for i:=1 to N
     begin
       jumlah := jumlah + i;
```

```
    end; {for}
    writeln('Jumlah deret = ', jumlah);
end.
```

b. *WHILE*

```
program PenjumlahanDeret;
{ Menjumlahkan deret
              1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

(*DEKLARASI*)
var
   N : integer;
   i : integer;
   jumlah : integer;

(*ALGORITMA:*)
begin
    write('Berapa N?'); readln(N);     { banyaknya suku deret }
    jumlah:=0;         { inisialisasi jumlah deret dengan 0 }
    i:=1;
    while i <= N do
     begin
       jumlah := jumlah + i;
       i := i + 1;
     end;{while}
    { i > N  }          { kondisi setelah pengulangan berhenti }
    writeln('Jumlah deret = ', jumlah);
end.
```

c. *REPEAT*

```
program PENJUMLAHAN_DERET;
{ Menjumlahkan deret
              1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu.}

(*DEKLARASI*)
var
   N : integer;
   i : integer;         { suku deret }
   jumlah : integer;    { jumlah deret }

(*ALGORITMA:*)
begin
    write('Berapa N?'); readln(N);     { banyaknya suku deret }
    jumlah:=0;         { inisialisasi jumlah deret dengan 0 }
    i := 1;
    repeat
      jumlah := jumlah + i;
      i := i + 1;
    until i > N;
    writeln('Jumlah deret = ', jumlah);
end.
```

C:

a. *FOR*

```
/* PROGRAM PenjumlahanDeret */
/* Menjumlahkan deret
              1 + 2 + 3 + … + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu. */

#include <stdio.h>

main()
{   /*DEKLARASI*/
   int N;
   int i;
   int jumlah;

  /*ALGORITMA:*/
   printf("Berapa N?"); scanf("%d",&N);   /* banyaknya suku deret */
   jumlah = 0;                  /* inisialisasi jumlah deret dengan 0 */
   for (i = 1; i <= N; i++)
   {
      jumlah = jumlah + i;
   }
   printf("Jumlah deret = %d \n", jumlah);
}
```

b. *WHILE*

```
/* PROGRAM PenjumlahanDeret */
/* Menjumlahkan deret
              1 + 2 + 3 + … + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu. */

#include <stdio.h>

main()
{   /*DEKLARASI*/
   int N;
   int i;
   int jumlah;

  /*ALGORITMA:*/
   printf("Berapa N?"); scanf("%d",&N);  /* banyaknya suku deret */
   jumlah = 0;        /* inisialisasi jumlah deret dengan 0*/
   i = 1;
   while (i <= N)
    {
      jumlah = jumlah + i;
      i++;      /* sama dengan i = i + 1 */
    }
   /* i > N  */      /* kondisi setelah pengulangan berhenti */
   printf("Jumlah deret = %d", jumlah);
}
```

c. *REPEAT*

```
/* PROGRAM PenjumlahanDeret */
/* Menjumlahkan deret
              1 + 2 + 3 + ... + N
dengan N adalah bilangan bulat positif. Nilai N dibaca terlebih dahulu. */

#include <stdio.h>

main()
{   /*DEKLARASI*/
   int N;
   int i;
   int jumlah;

   /*ALGORITMA:*/

   printf("Berapa N?"); scanf("%d",&N);
   jumlah = 0;     /* inisialisasi jumlah deret dengan 0 */
   i = 1;
   do {
      jumlah = jumlah + i;
      i++;      /* sama dengan i = i + 1 */
   } while (i <= N);
   printf("Jumlah deret = %d", jumlah);
}
```

2. Algoritma menghitung rata-rata dari dari sejumlah data bilangan bulat

ALGORITMIK

```
PROGRAM HitungRataRata2
{ Menghitung rata-rata dari sejumlah data bilangan bulat yang dibaca
dari papan ketik selama data yang dibaca tidak sama dengan 0. }

DEKLARASI
  x : integer        { data bilangan yang dibaca dari papan ketik }
  i : integer        { pencacah banyak data }
  jumlah : integer   { pencatat jumlah data }
  rerata : real      { nilai rata-rata seluruh data }

ALGORITMA:
  i ← 0              { inisialisasi pencatat banyak data dengan 0}
  jumlah ← 0         { inisialisasi jumlah data dengan 0}
  read(x)
  while x ≠ 0 do   { lakukan penjumlahan selama x tidak nol }
    i ← i + 1         { naikkan pencatat banyaknya data }
    jumlah ← jumlah + x
    read(x) /
  endwhile
  { x = 0 }
  if  i ≠ 0 then  { data yang dibaca minimal 1 buah }
    rerata ← jumlah/i
    write(rerata)
```

```
  else
    write('tidak ada data yang dimasukkan')
  endif
```

PASCAL:

```
program HitungRataRata2;
{ Menghitung rata-rata dari sejumlah data bilangan bulat yang dibaca
dari papan ketik selama data yang dibaca tidak sama dengan 0. }

(*DEKLARASI*)
var
  x : integer;          { data bilangan yang dibaca dari papan ketik }
  i : integer;          { pencacah banyak data }
  jumlah : integer;     { pencatat jumlah data }
  rerata : real;        { nilai rata-rata seluruh data }

(*ALGORITMA:*)
begin
  i:=0;                 { inisialisasi pencatat banyak data dengan 0}
  jumlah:=0;            { inisialisasi jumlah data dengan 0}
  write('Ketikkan nilai x : '); readln(x);
  while x <> 0 do    { lakukan penjumlahan selama x tidak nol }
   begin
     i := i + 1;           { naikkan pencatat banyaknya data }
     jumlah := jumlah + x;
     write('Ketikkan nilai x : '); readln(x);
  end;{while}
  { x = 0 }

  if  i <> 0 then  { data yang dibaca minimal 1 buah }
   begin
     rerata := jumlah/i;
     writeln('Rata-rata = ', rerata);
   end
  else
     writeln('tidak ada data yang dimasukkan');
  {endif}
```

C:

```
/* PROGRAM HitungRataRata2 */
/* Menghitung rata-rata dari sejumlah data bilangan bulat yang dibaca
dari papan ketik selama data yang dibaca tidak sama dengan 0. */

#include <stdio.h>

main()
{
 /*DEKLARASI*/
  int x;        /* data bilangan yang dibaca dari papan ketik */
  int i;        /* pencacah banyak data */
  int jumlah;   /* pencatat jumlah data */
  float rerata;     /* nilai rata-rata seluruh data */
```

```
/*ALGORITMA:*/
i = 0;                  /* inisialisasi pencatat banyak data dengan 0 */
jumlah = 0;             /* inisialisasi jumlah data dengan 0 */
printf"Ketikkan nilai x : "); scanf("%f", &x);
while (x != 0)   /* lakukan penjumlahan selama x tidak nol */
 {
   i = i + 1;           /* naikkan pencatat banyaknya data */
   jumlah = jumlah + x;
   printf("Ketikkan nilai x : "); scanf("%f", &x);
 }
/* x = 0 */

if  (i != 0)  /* data yang dibaca minimal 1 buah */
 {
   rerata = jumlah/(float)i;
   printf("Rata-rata = %f \n", rerata);
 }
else
   printf("tidak ada data yang dimasukkan");
/*endif*/
```

3. Algoritma peluncuran roket

ALGORITMIK

```
PROGRAM PeluncuranRoket
{ Hitung mundur peluncuran roket }

DEKLARASI
  i : integer

ALGORITMA:
  for i ← 100 downto 0 do
    write(i)
  endfor
  write('Go!')  { roket meluncur pada hitungan 0 }
```

PASCAL:

```
program PeluncuranRoket;
{ Hitung mundur peluncuran roket }

(*DEKLARASI*)
var
  i : integer;

(*DESKRIPSI:*)
begin
  for i:=100 downto 0 do
   begin
     writeln(i);
   end;{for}
  writeln('Go!');  { roket meluncur pada hitungan 0 }
end.
```

C:

```
/* PROGRAM PeluncuranRoket */
/* Hitung mundur peluncuran roket */

#include <stdio.h>

main()
{
  /*DEKLARASI*/
  int i;

  /*DESKRIPSI:*/
  for (i = 100; k >= 1; i--)
   {
     printf("%d",i);
   }
  printf("Go!");  /* roket meluncur pada hitungan 0*/
}
```

7.7 Membaca/Menulis dari/ke Arsip

Jika data masukan yang akan dibaca dari arsip banyak jumlahnya, maka kita perlu menggunakan struktur pengulangan. Pembacaan data dari arsip bergantung pada susunan data tersebut di dalam arsip. Susunan data di dalam arsip harus diketahui sebelum program ditulis agar dapat ditentukan cara membaca datanya datanya. Pembaca harus mengacau kembali ke upa-Bab 5.3 perihal cara membaca/menulis data dari/ke arsip dalam bahasa Pascal dan C.

Di bawah ini diberikan beberapa contoh program pembacaan dan penulisan dari/ke arsip teks. Algoritma dari program-program tersebut diambil dari contoh-contoh yang sudah dikemukakan di dalam Bab 7 ini.

(a) Menghitung nilai rata-rata (1)
Tinjau kembali Algoritma 7.11 untuk menghitung nilai rata-rata dari *N* buah data *integer*. Misalkan data banyaknya data (*N*) dan seluruh data *integer* sudah disimpan di dalam arsip `data.txt`. Susunan data di dalam `data.txt` sebagai berikut: baris pertama berisi nilai *N*, sedangkan baris-baris berikutnya berisi data *integer* sebanyak *N* buah, antara setiap data dipisahkan oleh spasi. Sebagai contoh, misalkan arsip `data.txt` sudah berisi data yang telah disusun sesuai dengan format yang ditetapkan di atas:

```
10
6  5  7  2  3
8  1  4  8  12
```

Baris pertama, 10, menyatakan banyaknya data, sedangkan baris-baris berikutnya berisi 10 buah data *integer*. Tulislah program untuk membaca data

integer dari arsip dan menghitung nilai rata-rata seluruh data. Keluaran program adalah nilai rata-rata tersebut. Keluaran program dicetak ke layar. Program *Pascal* dan *C* untuk spesifikasi masalah ini adalah seperti di bawah ini:

PASCAL:

```
PROGRAM HitungRataRata;
{ Menghitung rata-rata N buah bilangan bulat yang dibaca dari arsip
"data.txt". N > 0. }

(* DEKLARASI *)
var
  N : integer;        { banyaknya data, > 0 }
  x : integer;        { data bilangan bulat yang dibaca dari arsip }
  i : integer;        { pencacah banyak data }
  jumlah : integer;   { pencatat jumlah data }
  rerata : real;      { nilai rata-rata seluruh data }
  Fin    : text;      { pointer ke arsip masukan }

(* ALGORITMA: *)
begin
  { buka arsip masukan }
  assign(Fin, 'data.txt');
  reset(Fin);

  read(Fin, N);    {baca banyaknya data }
  jumlah := 0;     { inisialisasi }
  i := 1;            { inisialisasi pencacah }
  while i <= N do
   begin
    read(Fin, x);
    jumlah := jumlah + x;
    i := i + 1;
   end;{ while }
  { i > n }
  rerata:=jumlah/N;
  writeln('Rata-rata = ', rerata);
end;
```

C:

```
/* PROGRAM HitungRataRata */
/* Menghitung rata-rata N buah bilangan bulat yang dibaca dari arsip
"data.txt". N > 0. */

main()
{
  /* DEKLARASI */
  int N;        /* banyaknya data, > 0 */
  int x;        /* data bilangan bulat yang dibaca dari arsip */
  int i;        /* pencacah banyak data */
  int jumlah;   /* pencatat jumlah data */
  float rerata;     /* nilai rata-rata seluruh data */
  FILE *Fin;      /* pointer ke arsip masukan */
```

```
/* ALGORITMA: */

  /* buka arsip masukan */
  Fin = fopen("data.txt", "r");

  fscanf(Fin, "%d", &N);  /* baca banyaknya data */
  jumlah = 0;     /* inisialisasi */
  i = 1;          /* inisialisasi pencacah */
  while (i <= N)
   {
    fscanf(Fin, "%d", &x);
    jumlah = jumlah+x;
    i = i + 1;
   }/* while */
  /* i > n */

  rerata = (float)jumlah/N;
  printf("Rata-rata = %f", rerata);
}
```

(b) Menghitung nilai rata-rata (2)

Tinjau kembali Algoritma 7.13 untuk menghitung nilai rata-rata dari sekumpulan data *integer*, tetapi penanda akhir data adalah 9999. Misalkan seluruh data *integer* sudah disimpan di dalam arsip `data.txt`. Susunan data di dalam `data.txt` adalah nilai-nilai *integer* yang masing-masing dipisahkan oleh spasi. Nilai terakhir adalah 9999 yang menandakan akhir data, tetapi 9999 tidak termasuk data yang akan dihitung dalam perhitungan nilai rata-rata.

Sebagai contoh, misalkan arsip `data.txt` sudah berisi data yang telah disusun sesuai dengan format yang ditetapkan di atas:

```
6  5  7  2  3
8  1  4  8  12
9999
```

Tulislah program untuk membaca data *integer* dari arsip dan menghitung nilai rata-rata seluruh data. Keluaran program adalah nilai rata-rata tersebut. Keluaran program dicetak ke layar. Program *Pascal* dan *C* untuk spesifikasi masalah ini adalah seperti di bawah ini:

PASCAL:

```
PROGRAM HitungRataRata2;
{ Menghitung rata-rata dari sejumlah data bilangan bulat yang dibaca
dari arsip "data.txt". penanda akhir data di dalam arsip adalah 9999.
}

(* DEKLARASI *)
var
  x : integer;        { data bilangan bulat yang dibaca dari arsip }
  i : integer;        { pencacah banyak data }
```

```
  jumlah : integer;  { pencatat jumlah data }
  rerata : real;     { nilai rata-rata seluruh data }
  Fin    : text;     { pointer ke arsip masukan }

(* ALGORITMA: *)
begin
  { buka arsip masukan }
  assign(Fin, 'data.txt');
  reset(Fin);

  i := 0;              { inisialisasi pencatat banyak data dengan 0}
  jumlah := 0;         { inisialisasi jumlah data dengan 0}
  read(Fin,x);
  while x <> 9999 do   { lakukan penjumlahan selama x <> 9999 }
   begin
    i := i + 1;        { naikkan pencatat banyaknya data }
    jumlah := jumlah + x;
    read(Fin,x);
   end;{while}
  { x = 9999 }

  if  i <> 0 then  { data yang dibaca minimal 1 buah }
   begin
    rerata := jumlah/i;
    writeln('Rata-rata = ', rerata);
   end
  else
    writeln('Arsip kosong (hanya berisi 9999)');
  {endif}
end.
```

C:

```
/* PROGRAM HitungRataRata */
{ Menghitung rata-rata dari sejumlah data bilangan bulat yang dibaca dari
arsip "data.txt". penanda akhir data di dalam arsip adalah 9999. }

main()
{
  /* DEKLARASI */
  int x;       /* data bilangan bulat yang dibaca dari arsip */
  int i;       /* pencacah banyak data */
  int jumlah;  /* pencatat jumlah data */
  float rerata;     /* nilai rata-rata seluruh data */
  FILE *Fin;     /* pointer ke arsip masukan */

/* ALGORITMA: */

  /* buka arsip masukan */
  Fin = fopen("data.txt", "r");

  jumlah = 0;    /* inisialisasi */
  i = 1;         /* inisialisasi pencacah */
  fscanf(Fin, "%d", &x);
  while (x != 9999)       /* lakukan penjumlahan selama x <> 9999 */
   {
    i = i + 1;            /* naikkan pencatat banyaknya data */
```

```
    jumlah = jumlah + x;
    fscanf(Fin, "%d", &x);
   }/* while */
  /* x = 9999 n */

  if (i != 0)    /* data yang dibaca minimal 1 buah */
   {
    rerata = (float)jumlah/i;
    printf("Rata-rata = %f", rerata);
   }
  else
    printf("Arsip kosong (hanya berisi 9999) \n");
  /*endif*/
}
```

(c) Menghitung nilai rata-rata (3)

Tinjau kembali Algoritma 7.12 untuk menghitung nilai rata-rata dari sekumpulan data *integer*. Misalkan seluruh data *integer* sudah disimpan di dalam arsip `data.txt`. Banyaknya data tidak diketahui sebelum pembacaan. Susunan data di dalam `data.txt` adalah nilai-nilai *integer* yang masing-masing dipisahkan oleh spasi.

Sebagai contoh, misalkan arsip `data.txt` sudah berisi data yang telah disusun sesuai dengan format yang ditetapkan di atas:

```
6  5  7  2  3
8  1  4  6  12
```

Tulislah program untuk membaca data *integer* dari arsip dan menghitung nilai rata-rata seluruh data. Keluaran program adalah nilai rata-rata tersebut. Keluaran program dicetak ke layar. Program *Pascal* dan *C* untuk spesifikasi masalah ini adalah seperti berikut ini:

PASCAL:

Tidak seperti persoalan (b) yang akhir datanya adakah nilai 9999, maka pada persoalan (c) ini akhir data diidentifikasi dengan akhir arsip. Akhir arsip dapat ditentukan dengan fungsi *EOF(f)*. Fungsi *EOF(f)* mengembalikan nilai *true* jika posisi arsip melampaui karakter terakhir di dalam arsip atau jika arsip kosong (tidak berisi karakter apa pun); sebaliknya *EOF(f)* mengembalikan *false*.

```
PROGRAM HitungRataRata;
{ Menghitung rata-rata N buah bilangan bulat yang dibaca dari arsip
"data.txt". Akhir data diidentifikasi dengan fungsi EOF. }

(* DEKLARASI *)
var
  x : integer;        { data bilangan bulat yang dibaca dari arsip }
  i : integer;        { pencacah banyak data }
  jumlah : integer;   { pencatat jumlah data }
  rerata : real;      { nilai rata-rata seluruh data }
```

```
  Fin    : text;      { pointer ke arsip masukan }

(* ALGORITMA: *)
begin
  { buka arsip masukan }
  assign(Fin, 'data.txt');
  reset(Fin);

  jumlah := 0;     { inisialisasi }
  i := 0;          { inisialisasi pencacah }
  while (not EOF(Fin) do
   begin
    read(Fin, x);
    i := i + 1;
    jumlah := jumlah + x;
   end;{while}
  { EOF(Fin) }

  if  i <> 0 then  { data yang dibaca minimal 1 buah }
   begin
     rerata := jumlah/i;
     writeln('Rata-rata = ', rerata);
   end
  else
     writeln('Arsip kosong');
  {endif}

end;
```

C:

Di dalam bahasa *C*, fungsi yang serupa dengan fungsi *EOF(f)* adalah *feof(f)*. Fungsi ini mengembalikan nilai tidak-nol jika akhir arsip dicapai.

```
/* PROGRAM HitungRataRata */
/* Menghitung rata-rata N buah bilangan bulat yang dibaca dari arsip
"data.txt". Akhir data diidentifikasi dengan fungsi EOF.*/

main()
{

  /* DEKLARASI */
  int x;        /* data bilangan bulat yang dibaca dari arsip */
  int i;        /* pencacah banyak data */
  int jumlah;   /* pencatat jumlah data */
  float rerata;     /* nilai rata-rata seluruh data */
  FILE *Fin;     /* pointer ke arsip masukan */

/* ALGORITMA: */

  /* buka arsip masukan */
  Fin = fopen("data.txt", "r");

  jumlah = 0;     /* inisialisasi */
  i = 0;          /* inisialisasi pencacah */
  while (!feof(Fin))
```

```
  {
   fscanf(Fin, "%d", &x);
   i = i + 1;
   jumlah = jumlah+x;
  }/* while */
  /* feof(Fin) */

 if (i != 0)    /* data yang dibaca minimal 1 buah */
  {
   rerata = (float)jumlah/i;
   printf("Rata-rata = %f", rerata);
  }
 else
   printf("Arsip kosong (hanya berisi 9999) \n");
 /*endif*/
}
```

Contoh-contoh program *Pascal/C* di atas dapat dimodifikasi sehingga nama arsip di-*input*-kan oleh pengguna program pada waktu eksekusi (jadi, tidak harus `data.txt`). Selain itu, keluaran program juga dapat ditulis ke dalam arsip (jadi, tidak ditampilkan ke layar). Lihat contoh hal ini di dalam upa-Bab 5.3.

Soal Latihan Bab 7

1. Buatlah algoritma yang membaca sembarang karakter dan mencetaknya ke layar. Buatlah algoritma untuk dua kasus:
   (a) jumlah karakter yang dibaca diketahui, yaitu *N* buah (baca *N* terlebih dahulu)
   (b) jumlah karakter yang dibaca tidak diketahui, tetapi proses pembacaan berakhir jika karakter yang dimasukkan adalah karakter titik (karakter titik tidak ikut dicetak)

2. Buatlah algoritma untuk menghitung jumlah *N* buah bilangan ganjil pertama (yaitu, 1 + 3 + 5 + ...). Catatan: *N* adalah bilangan bulat tidak negatif.

3. Buatlah algoritma untuk menghitung jumlah bilangan ganjil dari 1 sampai *N* (yaitu, 1 + 3 + 5 + ... + *N*). (perhatikan perbedaan soal latihan ini dengan soal nomor 1).

4. Tuliskan program *Pascal* dan *C* untuk mencetak gambar segitiga bintang sebanyak *N* baris, setiap baris ke-*i* berisi *i* buah bintang. Nilai *N* dibaca dari papan kunci.

   Contoh: masukan: *N* = *5*

keluaran:

```
*
**
***
****
*****
```

5. Tuliskan algoritma untuk menampilkan semua solusi bilangan bulat tidak negatif dari persamaan berikut:

   $x + y + z = 25$

   yang dalam hal ini, $x \geq 0$, $y \geq 0$, dan $z \geq 0$.

6. Buatlah algoritma yang akan mengonversi bilangan bulat positif ke angka romawinya. Rancanglah algoritma tersebut sehingga pengonversian tersebut dapat dilakukan berulangkali sampai nilai nol dibaca dari piranti masukan.

7. Seseorang mempunyai tabungan di sebuah bank. Ia dapat menyetor dan mengambil uangnya di bank tersebut, namun jumlah saldo minimum yang harus disisakan di dalam adalah Rp10.000. Ini artinya, jika saldonya Rp10.000, ia tidak dapat mengambil uang lagi. Kode transaksi untuk menyetor adalah 0 dan kode transaksi untuk mengambil adalah 1. Buatlah algoritma yang menyimulasikan transaksi yang dilakukan orang tersebut. Algoritma menerima masukan berupa kode transaksi dan jumlah uang yang disetor/diambil. Rancanglah algoritma tersebut sehingga memungkinkan penabung dapat melakukan transaksi berulangkali sampai saldo yang tersisa Rp10.000 atau jumlah uang yang diambil lebih besar dari saldonya.

   Catatan: nilai uang yang diambil selalu merupakan bilangan bulat.

8

Contoh-contoh Pemecahan Masalah I

Bab 8 ini menyajikan contoh-contoh masalah dan algoritmanya, mulai dari contoh yang sederhana sampai contoh yang lebih sulit. Contoh masalah yang disajikan mencakup struktur runtunan, struktur pemilihan, dan struktur pengulangan. Belajar melalui contoh adalah salah satu cara terbaik belajar algoritma dan pemrograman. Melalui contoh kasus, kita menemukan pola-pola yang dapat digunakan untuk memecahkan masalah yang sejenis.

Contoh Masalah 8.1: Pengubahan jam

Didefinisikan tipe `Jam` dan peubah `J` dan `p` sebagai berikut:

```
type Jam : record
             <hh : integer, {0..23}
              mm : integer, {0..59}
              ss : integer, {0..59}
             >

J : Jam
p : integer
```

Anda diminta membuat algoritma yang:
- mengisi (*assignment*) peubah `J` dengan jam `16:10:34`;
- membaca `p` menit dari piranti masukan;
- mengubah nilai `J` setelah ditambah `p` menit. Tampilkan jam `J` yang baru.

Penyelesaian

Penyelesaian masalah ini menggunakan prinsip konversi jam (*hh*:*mm*:*ss*) ke jumlah detik, tambahkan jumlah detik dengan *p*, lalu konversi kembali jumlah detik ke jam (*hh*:*mm*:*ss*).

```
PROGRAM PengubahanJam
{ Mengisi nilai jam ke sebuah peubah, menambah dengan p menit }

DEKLARASI
  type Jam : record <hh:integer ,  {0..23}
                     mm:integer ,  {0..59}
                     ss:integer    {0..59}
                    >
  J : Jam
  p, TotalDetik : integer

ALGORITMA:
   { isikan jam 16:10:34 ke peubah J }
   J.hh ← 16
   J.mm ← 10
   J.ss ← 34

   read(p)     { baca p menit }

   { konversi J ke total detik, tambahkan dengan p * 60 detik }
   TotalDetik ← (J.hh*3600) + (J.mm*60) + J.ss + (p*60)

   { tentukan jam yang baru setelah penambahan p menit }
   J.hh ← TotalDetik div 3600
   sisa ← TotalDetik mod 3600
   J.mm ← sisa div 60
   J.ss ← sisa mod 60
   write(J.hh,J.mm,J.ss)
```

Algoritma 8.1 Mengubah jam setelah ditambah p menit.

Contoh Masalah 8.2: Aritmetika pecahan

Nilai pecahan disajikan dalam bentuk a/b, dengan syarat $b \neq 0$. Dalam hal ini, a disebut pembilang dan b penyebut. Keduanya bertipe *integer*. Contohnya, 4/5, 2/3, 6/10. Bilangan bulat pun dapat dinyatakan dalam bentuk pecahan, misalnya 5 = 5/1. Tulislah algoritma yang membaca dua buah nilai pecahan, $p1$ dan $p2$, lalu menghitung hasil penjumlahan keduanya ($p1 + p2$), selisih ($p1 - p2$), perkalian ($p1 \times p2$), dan pembagian ($p1/p2$). Hasil perhitungan tetap dalam bentuk pecahan.

Penyelesaian

Misalkan dua buah nilai pecahan yang akan dioperasikan adalah $p_1 = \frac{a}{b}$ dan $p_2 = \frac{c}{d}$.

Operasi aritmetika untuk pecahan:

(i) $p_1 + p_2 = \frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$ (iii) $p_1 \times p_2 = \frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$

(ii) $p_1 - p_2 = \frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$ (iv) $p_1 : p_2 = \frac{a}{b} : \frac{c}{d} = \frac{d}{c} \times \frac{a}{b} = \frac{ad}{bc}$

Kita definisikan tipe pecahan dan peubah $p1$, $p2$, dan $p3$ bertipe pecahan sebagai berikut:

```
type pecahan : record <a:integer, b:integer>
p1, p2 : pecahan
p3     : pecahan  { hasil operasi p1 dan p2 }
```

Algoritma operasi aritmetika dua buah pecahan sebagai berikut:

```
PROGRAM AritmetikaPecahan
{ Membaca dua buah pecahan, menghitung jumlah, selisih, perkalian, dan
pembagian kedua pecahan tersebut }

DEKLARASI
   type pecahan : record
                  <a:integer,
                   b:integer  {≠0}
                  >

   p1, p2 : pecahan
   p3     : pecahan  { hasil operasi p1 dan p2 }

ALGORITMA:
   { baca pecahan p1 dan p2 }
   read(p1.a, p1.b)
   read(p2.a, p2.b)

   { hitung p3 = p1 + p2 }
   p3.a ← p1.a*p2.b + p1.b*p2.a
   p3.b ← p1.b*p2.b
   write(p3.a,p3.b)

   { hitung p3 = p1 - p2 }
   p3.a ← p1.a*p2.b - p1.b*p2.a
   p3.b ← p1.b*p2.b
   write(p3.a,p3.b)

   { hitung p3 = p1 * p2 }
   p3.a ← p1.a*p2.a
   p3.b ← p1.b*p2.b
   write(p3.a,p3.b)

   { hitung p3 = p1 / p2 }
   p3.a ← p1.a*p2.b
   p3.b ← p1.b*p2.a
   write(p3.a,p3.b)
```

Algoritma 8.2 Aritmetika pecahan

Contoh Masalah 8.3: Kode Pos

Seorang pengirim surat menuliskan nama kota pada amplop surat tetapi tidak mencantumkan kode pos-nya. Buatlah algoritma yang menerima masukan nama kota dan menuliskan kode pos kota tersebut ke piranti keluaran. Kota-kota yang tersedia di dalam daftar hanya 5, yaitu:

```
Padang     : 25000
Bandung    : 40100
Solo       : 51000
Denpasar   : 72000
Palu       : 92300
```

Penyelesaian

```
PROGRAM KodePos
{ Menerima masukan nama kota dan mencetak kode pos kota ybs }

DEKLARASI
  kota : string

ALGORITMA:
  read(kota)
  case kota
    'padang'   : write('25000')
    'bandung'  : write('40100')
    'solo'     : write('51000')
    'denpasar' : write('72000')
    'palu'     : write('92300')
     otherwise :  write ('tidak ada di dalam daftar')
  endcase
```

Algoritma 8.3 Mencetak kode pos salah satu dari 5 kota

Contoh Masalah 8.4: Durasi antara dua buah jam

Seseorang yang menelepon dari warung telekomunikasi (wartel) selalu dicatat jam awal percakapan dan jam akhir percakapan bicara (dalam format *hh:mm:ss*). Tulislah algoritma yang membaca jam awal dan akhir percakapan, lalu menghitung lama percakapan dalam format waktu *hh:mm:ss* dengan cara analisis kasus.
Contoh:

Awal	Akhir	Durasi (Akhir – Awal)
8:40:12	8.45:36	0:5:24
8:40:54	8:42:10	0:1:14
8:40:40	10:20:36	1:39:56

Algoritma perhitungan selisih waktu harus menggunakan cara analisis kasus.

Penyelesaian

Masalah ini sama dengan Contoh 5.11, yang mana menghitung durasi antara dua buah jam dengan cara komputasi. Kali ini kita akan menghitung durasi waktu tetapi dengan cara analisis kasus.

Algoritma menghitung selisih dua buah waktu dengan cara analisis kasus relatif lebih sulit dibandingkan dengan cara kalkulasi (lihat Contoh 5.7). Kita harus menganalisis kasus-kasus seperti $ss2 \geq ss1$, $ss2 < ss1$, $mm2 \geq mm1$, dan $mm2 < mm1$.

Jika $ss2 \geq ss1$ atau $mm2 \geq mm1$, tidak ada masalah dalam menghitung selisih karena dapat langsung dikurangkan. Tetapi, jika $ss2 < ss1$ atau $mm2 < mm1$, maka kita harus "meminjam" satu menit atau satu jam dari besaran waktu yang lebih tinggi, sebab selisih dua buah waktu tidak boleh negatif dan harus terletak dalam rentang nilai yang sah (0..59 untuk menit/detik, dan 0..23 untuk jam).

Misalnya pada kasus $ss2 < ss1$ seperti berikut:

(8:42:10) – (8:40:54)

Karena 10 < 54, maka pinjam satu menit dari 42. Satu menit itu sama dengan 60 detik, sehingga pengurangan komponen detik menghasilkan

(10+60) – (54) = 14.

Karena 42 menit sudah dipinjam satu menit, maka sisanya 41 menit, sehingga pengurangan komponen menit menghasilkan

(41) – (40) = 1

Cara yang sama juga berlaku jika $mm2 < mm1$, yaitu pinjam satu jam (=60) menit dari waktu $hh2$. Karena peminjaman ini, $hh2$ berkurang 1.

```
PROGRAM SelisihWaktu
{ Membaca waktu awal dan waktu akhir percakapan, W1(hh:mm:ss) dan
W2(hh:mm:ss). Asumsikan W2.hh ≥ W1.hh. Kemudian, menghitung lama
percakapan, yaitu W2 - W1 = W3 .}

DEKLARASI
   type Jam : record < hh:integer , {0..23}
                       mm:integer , {0..59}
                       ss:integer   {0..59}
                     >

   W1 : Jam       { jam awal percakapan }
   W2 : Jam       { jam akhir percakapan }
   W3 : Jam       { lama percakapan }

ALGORITMA:
   { baca jam awal dan jam akhir percakapan }
   read(W1.hh, W1.mm, W1.ss)
```

```
read(W2.hh, W2.mm, W2.ss)

{ pengurangan detik }
if W2.ss ≥ W1.ss then     { tidak ada masalah pengurangan detik, OK }
   W3.ss ← W2.ss - W1.ss  { selisih detik }
else   { W2.ss < W1.ss }
   W3.ss ← (W2.ss + 60) - W1.ss   { pinjam satu menit dari menit2,
                                    lalu kurangi dengan detik1 }
   W2.mm ← W2.mm - 1      { menit2 berkurang satu karena dipinjam }
endif

{ pengurangan menit }
if W2.mm ≥ W1.mm then      { tidak ada masalah pengurangan menit, OK }
   W3.mm ← W2.mm - W1.mm   { selisih menit }

else  { W2.mm < W1.mm }
   W3.mm ← (W2.mm + 60) - W1.mm   { pinjam satu jam dari jam2,
                                    lalu kurangi dengan menit 1 }
   W2.J ← W2.J - 1        { jam2 berkurang satu karena dipinjam }
endif

{ pengurangan jam. Asumsi: W2.hh ≥ W1.hh }
W3.hh ← W2.hh - W1.hh

write(W3.hh,W3.mm,W3.ss)
```

Algoritma 8.4 Menghitung durasi (selsiih waktu) antara dua buah jam

Contoh Masalah 8.5: Menghitung biaya percakapan

Kembangkan algoritma pada Contoh Masalah 8.4 di atas sehingga dapat menghitung biaya percakapan yang tarif tiap pulsanya bergantung pada kode wilayah tujuan percakapan. Misalnya:

Kode	Wilayah Kota	Tarif Tiap Pulsa	Lama Pulsa
02	Jakarta	Rp 150	1 menit
0751	Padang	Rp 250	30 detik
0737	Medan	Rp 375	25 detik
091	Balikpapan	Rp 415	20 detik
098	Ternate	Rp 510	17 detik

Data masukan tambahan yang diperlukan adalah kode wilayah. Untuk membatasi masalah, maka kode wilayah yang diberikan hanya 5 seperti tabel di atas.

Penyelesaian

Perhitungan pulsa dan biaya percakapan adalah:

```
pulsa = lama percakapan /lama pulsa (bergantung wilayah)
biaya = pulsa * tarif pulsa (bergantung wilayah)
```

```
PROGRAM HitungBiayaPercakapan
{ Membaca waktu awal dan waktu akhir percakapan, W1(hh:mm:ss) dan
W2(hh:mm:ss). Asumsikan W2.hh ≥ W1.hh. Kemudian, menghitung lama
percakapan, yaitu W2 - W1 = W3, dan akhirnya menghitung biaya
percakapan. Tarif tiap pulsa bergantung pada wilayah tujuan
percakapan. Biaya percakapan dihitung dari lama percakapan }

DEKLARASI
   type Jam : record < hh:integer , {0..23}
                       mm:integer , {0..59}
                       ss:integer   {0..59}
                     >
   W1   : Jam       { jam mulai percakapan }
   W2   : Jam       { jam akhir percakapan }
   W3   : Jam       { lama percakapan }


   KodeWil  : string  { kode wilayah tujuan percakapan }
   durasi   : integer { lama percakapan dalam detik }
   PulsWil  : real    { lama pulsa, bergantung kode wilayah }
   TarifWil : real    { tarif per pulsa, bergantung pada kode wilayah}
   biaya    : real    { biaya percakapan }

ALGORITMA:
   read(W1.hh, W1.mm, W1.ss)     { jam awal percakapan }
   read(W2.hh, W2.mm, W2.ss)     { jam akhir percakapan }
   read(KodeWil)                 { kode wilayah tujuan percakapan }

   { pengurangan detik }
   if W2.ss ≥ W1.ss then     { tidak ada masalah pengurangan detik, OK
}
      W3.ss ← W2.ss - W1.ss  { selisih detik }
   else  { W2.ss < W1.ss }
      W3.ss ← (W2.ss + 60) - W1.ss   { pinjam satu menit dari menit2,
                                       lalu kurangi dengan detik1 }
      W2.mm ← W2.mm - 1       { menit2 berkurang satu karena dipinjam }
   endif

   { pengurangan menit }
   if W2.mm ≥ W1.mm then     { tidak ada masalah pengurangan menit, OK
}
      W3.mm ← W2.mm - W1.mm     { selisih menit }
   else   { W2.mm < W1.mm }
      W3.mm ← (W2.mm + 60) - W1.mm   { pinjam satu jam dari jam2,
                                       lalu kurangi dengan menit1 }
      W2.J ← W2.J - 1          { jam2 berkurang satu karena dipinjam }
   endif

   { pengurangan jam }
   W3.hh ← W2.hh - W1.hh

   { hitung lama percakapan dalam detik }
   durasi←(W3.hh*3600) + (W3.mm*60) + W3.ss

   { tentukan lama pulsa dan tarif tiap pulsa, bergantung
     pada kode wilayah tujuan percakapan }
```

```
case KodeWil
   '021'  : PulsWil ← 60
            TarifWil ← 150
   '0751' : PulsWil ← 30
            TarifWil ← 250
   '0737' : PulsWil ← 25
            TarifWil ← 375
   '0912' : PulsWil ← 20
            TarifWil ← 415
   '0981' : PulsWil ← 17
            TarifWil ← 510
endcase

{ hitung jumlah pulsa dan biaya untuk seluruh pulsa }
pulsa ← durasi/PulsWil
biaya ← pulsa * TarifWil
write(W3.hh, W3.mm, W3.ss, biaya)
```

Contoh Masalah 8.6: Hari berikutnya pada bulan Februari

Bulan Februari mempunyai jumlah hari yang unik. Jumlah harinya ada yang 28 hari dan ada yang 29 hari. Bulan Februari mempunyai jumlah hari 29 bila berada pada tahun kabisat. Pada tahun yang bukan tahun kabisat, jumlah harinya 28. Misalkan dibaca sebuah penanggalan pada bulan Februari. Tulislah algoritma untuk menentukan tanggal pada hari berikutnya.

Contoh:

```
Sekarang: 17-2-1999    → besok: 18-2-1999
Sekarang: 28-2-1999    (bukan tahun kabisat) → besok: 1-3-1999
Sekarang: 28-2-1996 (tahun kabisat) → besok: 29-2-1999
Sekarang: 29-2-1996    (tahun kabisat) → besok: 1-3-1999
```

Penyelesaian

Sebuah tanggal terdiri atas komponen: tanggal (*dd*), bulan (*mm*), dan tahun (*yy*). Untuk menentukan tanggal keesokan hari, kita harus menganalisis kasus berdasarkan tanggal sekarang:

- jika *dd* sekarang < 28, maka *dd* keesokan harinya adalah *dd* sekarang ditambah 1
- jika *dd* sekarang = 28, periksa apakah *yy* tahun kabisat. Jika *yy* tahun kabisat, maka *dd* keesokan harinya adalah *dd* sekarang ditambah 1 (tanggal 29). Jika *yy* bukan tahun kabisat, maka *dd* keesokan hari adalah tanggal 1 (bulan Maret)
- jika *dd* sekarang = 29, maka *dd* keesokan hari adalah tanggal 1 (bulan Maret)

```
PROGRAM TanggalBerikutnyaDiBulanFebuari
{ Membaca sebuah tanggal pada bulan Februari, kemudian menentukan
tanggal keesokan harinya }

DEKLARASI
```

```
type tanggal : record <dd : integer, {1..31}
                      mm : integer, {1..12}
                      yy : integer  { > 0 }
                     >
T : tanggal

ALGORITMA:
  T.mm ← 2                { bulan Februari }
  read(T.dd, T.yy)        { baca komponen tanggal dan tahun }
  if T.dd < 28 then       { tidak ada masalah dengan tahun }
    T.dd ← T.dd + 1       { tanggal besok }
  else
    if T.dd ≥ 28 then {tanggal besok bergantung tahun kabisat/tidak}
      if (T.yy mod 4 = 0 and T.yy mod 100 ≠ 0) or (T.yy mod 400 = 0))
      then  { tahun kabisat }
          if T.dd = 28 then
            T.dd ← T.dd + 1   { 29 }
          else  { T.dd = 29 }
            T.dd ← 1
            T.mm ← T.mm + 1   { bulan Maret }
          endif
      else { bukan tahun kabisat, jadi sesudah 28 langsung tanggal 1 }
          T.dd ← 1            { tanggal 1 bulan Maret }
          T.mm ← T.mm + 1     { bulan Maret }
      endif
    endif
  endif
  write(T.dd, T.mm, T.yy)  { cetak tanggal besok }
```

Algoritma 8.5 Menentukan tanggal berikutnya pada bulan Februari.

Contoh Masalah 8.7: Hari berikutnya (bebas bulan)

[LIE96] Perluaslah Contoh Masalah 8.6 di atas sehingga dapat menghitung tanggal berikutnya dari tanggal sekarang untuk sembarang bulan. Misalnya,

Tanggal Sekarang	Tanggal Besok
12-6-1996	13-6-1996
30-4-1986	1-5-1986
31-1-1991	1-2-1991
31-12-1992	1-1-1993

Tanggal sekarang dibaca terlebih dahulu dari piranti masukan.

Penyelesaian

Mula-mula kita harus memilah bulan-bulan berdasarkan jumlah harinya (lihat Contoh 6.20) karena tiap-tiap bulan tidak sama jumlah harinya:

Bulan	Jumlah hari
1, 3, 5, 7, 8, 10,12	31 hari
4, 6, 9, 11	30 hari
2	28 atau 29 hari, bergantung tahun kabisat.

Pada tahun kabisat, jumlah hari dalam bulan Februari adalah 28, sedangkan pada tahun bukan kabisat jumlah hari dalam bulan Februari adalah 29 (lihat Contoh Masalah 8.6). Di samping itu kita juga harus menangani kasus pergantian tahun, misalnya pada contoh pergantian tanggal 31-12-1992 menjadi 1-1-1993.

```
PROGRAM TanggalBerikutnya
{ Menentukan tanggal berikutnya setelah tanggal sekarang. Tanggal
  sekarang dibaca dari piranti masukan }

DEKLARASI
    type tanggal : record < dd : integer, {1..31}
                            mm : integer, {1..12}
                            yy : integer  { >0 }
                          >
    T : tanggal

ALGORITMA
    read(T.dd, T.mm, T.yy)
    case T.mm
       4, 6, 9, 11 : { sebulan 30 hari }
                     if T.dd < 30 then
                        dd ← T.dd + 1
                     else  { T.dd = 30 }
                        T.dd ← 1
                        T.mm ← T.mm + 1  { bulan baru }
                     endif


       1,3,5,7,8,10,12 : { sebulan 31 hari }
                         if T.dd < 31 then
                           T.dd ← T.dd + 1
                         else  { T.dd = 31 }
                           T.dd ← 1
                           T.mm ← T.mm + 1  { bulan baru }
                         endif


       2 : { bulan Februari}
           if T.dd < 28 then
              T.dd ← T.dd + 1
           else
              if T.dd ≥ 28 then {tanggal besok bergantung pada tahun
                                 kabisat}
                 if (T.yy mod 4 = 0 and T.yy mod 100 ≠ 0) or
                    (T.yy mod 400 = 0)) then  { tahun kabisat }
                    if T.dd = 28 then
                       T.dd ← T.dd + 1    { 29 }
                    else  { T.dd = 29 }
                       T.dd ← 1
                       T.mm ← T.mm + 1    { bulan Maret }
                    endif
                 else { bukan tahun kabisat, jadi sesudah 28 langsung
                        tanggal 1 }
                    T.dd ← 1            { tanggal 1 bulan Maret }
```

```
                T.mm ← T.mm + 1    { bulan Maret }
            endif
        endif

    12 : { hati-hati dengan pergantian tahun }
        if T.dd < 31 then
            T.dd ← T.dd + 1
        else { T.dd = 31 }
            T.dd ← 1
            T.mm ← 1    {Januari }
            T.yy ← T.yy + 1  { tahun baru }
        endif

endcase
{ cetak tanggal berikutnya itu }
write(T.dd, '-', T.mm, '-', T.yy)
```

Algoritma 8.6 Menentukan tanggal berikutnya (bebas bulan).

Contoh Masalah 8.8: Akar persamaan kuadrat

Di SMA Anda tentu masih ingat bahwa akar-akar persaman kuadrat

$$ax^2 + bx + c = 0 \qquad ; \text{dengan syarat } a \neq 0$$

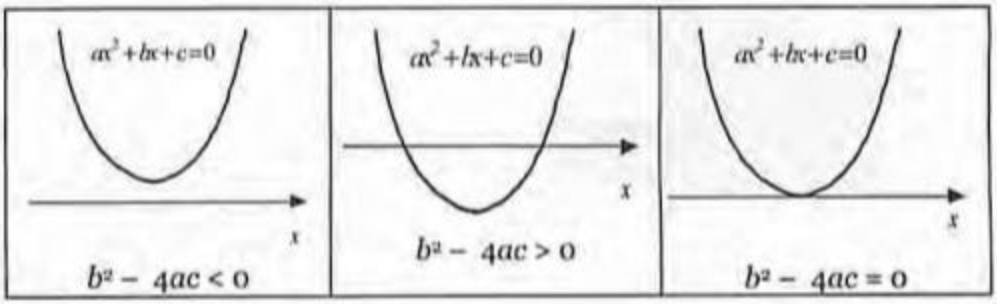
adalah nilai-nilai x yang menyebabkan persamaan menjadi nol. Ditinjau dari grafiknya, akar persamaan kuadrat menyatakan titik potong kurva dengan sumbu-x. Akar-akar persamaan kuadrat dapat dihitung dengan rumus *abc* sebagai berikut:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Jenis akar bergantung pada nilai $b^2 - 4ac$ (yang disebut diskriminan atau D):

- jika $b^2 - 4ac < 0$, maka akar-akarnya imaginer
- jika $b^2 - 4ac > 0$, maka akar-akarnya riil dan berbeda, $x_1 \neq x_2$
- jika $b^2 - 4ac = 0$, maka akar-akarnya riil dan kembar, $x_1 = x_2$



Sebagai contoh, persamaan kuadrat $x^2 - 4x - 5 = 0$, yang dalam hal ini:

$$a = 1, b = -4, c = -5$$

Nilai diskriminan persamaan kuadrat tersebut adalah

$$D = b^2 - 4ac = (-4)^2 - 4(1)(-5) = 16 + 20 = 36$$

Karena $b^2 - 4ac = 36 > 0$, maka akar-akar persamaan kuadrat adalah riil dan berbeda, yaitu

$$x_1 = \frac{-(-4)+\sqrt{36}}{2(1)} = 5 \quad \text{dan} \quad x_2 = \frac{-(-4)-\sqrt{36}}{2(1)} = -1$$

Tulislah algoritma untuk mencari akar persamaan kuadrat. Sebagai data masukannya adalah koefisien persamaan, a, b, dan c, dengan syarat $a \neq 0$, sedangkan keluarannya adalah akar-akar persamaannya.

Penyelesaian

Analisis kasus terhadap permasalahan ini menghasilkan tiga macam kasus:
Kasus 1: $D < 0$, akar-akarnya imaginer
Kasus 2: $D > 0$, akarnya riil dan berbeda ($x_1 \neq x_2$)
Kasus 3: $D = 0$, akar-akarnya kembar ($x_1 = x_2$)

```
PROGRAM AkarPersamaanKuadrat
{ Menghitung akar-akar persamaan kuadrat ax² + bx + c = 0. Nilai a, b,
dan c, dibaca sebagai masukan (a ≠ 0), sedangkan akar-akarnya dicetak
sebagai keluaran. }

DEKLARASI
   a, b, c : real      { koefisien persamaan kuadrat}
   D       : real      { diskriminan }
   x1, x2  : real      { akar-akar persamaan kuadrat}

ALGORITMA:

   read(a,b,c)          { baca koefisien persaman kuadrat}
   D ← b*b - 4*a*c      { hitung diskriminan}
   if D < 0 then        { Kasus 1 }
      write('kar-akar persamaan kuadrat imaginer!'
   else
      if D > 0 then
         { dua akar riil berbeda}
         x1 ← (-b + SQRT(D))/(2*a)
         x2 ← (-b - SQRT(D))/(2*a)
      else { D = 0 }
         { dua akar riil kembar }
         x1 ← (-b + SQRT(D))/(2*a)
         x2 ← x1
      endif
      write(x1,x2)      { cetak akar ke piranti keluaran }
   endif
```

Algoritma 8.7 Menghitung akar-akar persamaan kuadrat.

Keterangan:
SQRT adalah fungsi baku untuk menghitung akar pangkat dua. Kita mengasumsikan fungsi tersebut sudah terdefinisi.

Contoh Masalah 8.9: Menghitung jumlah karakter

Tulislah algoritma yang menghitung jumlah karakter yang dibaca secara berulang-ulang dari papan ketik. Pembacaan karakter selesai jika karakter yang dibaca adalah karakter titik (tetapi titik tidak termasuk dalam hitungan jumlah karakter). Misalnya, jika karakter yang dibaca berturut-turut: 'a', 'm', 't', '.', maka jumlah karakter = 3 (tidak termasuk titik).

Penyelesaian

Kita memerlukan sebuah peubah yang mencacah jumlah karakter yang dibaca. Peubah tersebut diinisialisasi dengan 0. Konstruksi pengulangan yang paling tepat digunakan adalah *WHILE*, karena karakter yang dibaca diperiksa di awal. Jika karakter tersebut titik, maka pembacaan selesai.

```
PROGRAM CacahKarakter
{ Menghitung jumlah karakter yang dibaca dari papan ketik. Akhir
pembacaan adalah karakter titik. }

DEKLARASI
   cc : char
   ncc : integer

ALGORITMA:
   ncc ← 0
   read(cc)    { baca karakter pertama, mungkin '.' }
   while (cc '.') do
     ncc ← ncc + 1
     read(cc)
   endwhile
   { cc = '.' }

   write(ncc)
```

Algoritma 8.8 Menghitung jumlah karakter yang dibaca

Periksa Algoritma 8.8; jika karakter pertama yang dibaca adalah titik, maka jumlah karakter (ncc) akan tetap nol, pertanda algoritma di atas tetap benar.

Contoh Masalah 8.10: Menghitung jumlah kemunculan angka, spasi, dan lainnya

Tulislah algoritma yang menghitung jumlah kemunculan angka (digit), jumlah spasi, dan jumlah karakter lainnya bila karakter dibaca secara berulang-ulang dari papan ketik. Pembacaan karakter selesai jika karakter yang dibaca adalah karakter titik (tetapi titik tidak termasuk dalam hitungan jumlah karakter).

Penyelesaian

Contoh masalah ini merupakan pengembangan dari Contoh Masalah 8.9. Dari karakter-karakter yang dibaca, kita klasifikasikan mana yang karakter angka, mana yang karakter spasi, dan karakter lainnya. Untuk setiap jenis karakter kita hitung jumlah kemunculannya. Misalnya, jika rangkaian karakter yang dibaca adalah

```
'5'   'u'   'v'   ' '   '?'   '8'   '.'
```

maka jumlah karakter angka = 2, jumlah spasi = 1, dan jumlah karakter lainnya = 3. Karakter titik tidak dihitung karena ia merupakan penanda akhir pembacaan.

Kita memerlukan tiga buah peubah yang mencacah jumlah kemunculan masing-masing jenis karakter tesrebut. Ketiga peubah tersebut diinisialisasi semunya dengan 0.

```
PROGRAM CacahJenisKarakter
{ Menghitung jumlah kemunculan angka, spasi, dan karakter lainnya. Karakter
dibaca dari papan ketik. Akhir pembacaan adalah karakter titik. }

DEKLARASI
   cc : char
   nangka : integer
   nspasi : integer
   nlainnya : integer

ALGORITMA:

Nangka ← 0
   Nspasi ← 0
   Nlainnya ← 0
   read(cc)   { baca karakter pertama, mungkin '.' }

   while (cc '.') do

      case cc
        '0', '1', '2', '3', '4',
        '5', '6', '7', '8', '9'  : nangka ← nangka + 1

        ' ' : nspasi ← nspasi + 1

        otherwise : nlainnya ← nlainnya + 1
      endcase

      read(cc)

   endwhile
   { cc = '.' }

   write(ndigit, nangka, nlainnya)
```

Algoritma 8.9 Menghitung jumlah karakter angka, spasi, dan lainnya

Contoh Masalah 8.11: Menentukan nilai terkecil (1)

Tulislah algoritma untuk menentukan nilai terkecil dari *N* buah data *integer* yang dibaca dari papan ketik. Nilai *N* dibaca terlebih dahulu (*N* > 0).

Penyelesaian

Baca data pertama terlebih dahulu. Asumsikan data pertama ini sebagai nilai terkecil (*min*) sementara. Selanjutnya, baca data berikutnya. Jika data tersebut lebih kecil dari *min*, maka ia menjadi *min* yang baru. Ulangi untuk data masukan yang lain. Pada akhir pengulangan, *min* menjadi nilai terkecil dari seluruh data.

Karena banyaknya data diketahui di awal, maka jumlah pengulangan dapat ditentukan. Pada masalah ini, struktur *FOR* dan *WHILE* dapat kita gunakan, sama baiknya. Di sini kita pilih struktur *FOR*. Penggunaan *REPEAT* dapat menghasilkan algoritma yang salah. Penjelasannya ada di bawah.

```
PROGRAM CariMinimum1
{ Menentukan bilangan terkecil dari N data integer yang dibaca dari
papan ketik. Nilai N ditentukan terlebih dahulu (N > 0). }

DEKLARASI
   N   : integer   { banyaknya data masukan, > 0 }
   x   : integer   { data yang dibaca }
   min : integer   { data terkecil/minimum }
   i   : integer   { pencacah pengulangan }

ALGORITMA:
  read(N)
  read(x) { baca data pertama }
  min ← x  { asumsikan min adalah data pertama }
  for i ← 2 to N do  { lanjutkan dengan data ke-2 sampai ke-N }
    read(x)
    if x < min then
      min ← x
    endif
  endfor
  write(min)
```

Algoritma 8.10 Mencari data terkecil (minimum) dari N buah data *integer* yang dibaca dari papan ketik (1).

Jika menggunakan konstruksi *REPEAT* seperti di bawah ini, SALAH:

```
PROGRAM CariMinimum1
{ Menentukan bilangan terkecil dari N data integer yang dibaca dari
piranti masukan. Nilai N ditentukan terlebih dahulu (N > 0). }

DEKLARASI
   N   : integer   { banyaknya data masukan, > 0 }
   x   : integer   { data yang dibaca }
   min : integer   { data terkecil/minimum }
   i   : integer   { pencacah pengulangan }
```

```
ALGORITMA:
  read(N)
  read(x) { baca data pertama }
  min ← x  { asumsikan min adalah data pertama }
  i ← 2    { tinjau untuk data kedua }
  repeat
    read(x)
    if x < min then
      min ← x
    endif
    i ← i + 1
  until i > N
  write(min)
```

Mengapa penggunaan konstruksi *REPEAT* pada algoritma di atas salah? Penyebabnya, bila banyaknya data hanya satu buah ($N = 1$), maka pembacaan data kedua tetap dilakukan (yang sebenarnya tidak ada). Hal ini karena pemeriksaan kondisi pengulangan dilakukan di akhir. Sebagai latihan, cobalah Anda gunakan struktur *WHILE*, lalu periksa apakah algoritma tetap benar.

Contoh Masalah 8.12: Menentukan data terkecil (2)

Ulangi Contoh Masalah 8.11, tetapi banyaknya data tidak diketahui. Pembacaan data berakhir bila data yang dibaca adalah 9999.

Penyelesaian

Dalam masalah ini, struktur *FOR* jelas tidak dapat digunakan, karena jumlah pengulangan tidak dapat ditentukan di awal. Struktur pengulangan yang dapat kita gunakan adalah *WHILE*. Contoh Masalah 8.9 ada kemiripan dengan masalah ini.

Jika data pertama bukan 9999, asumsikan ia sebagai *min* sementara. Selanjutnya, baca data kedua dan seterusnya. Selama data yang dibaca bukan 9999, bandingkan ia dengan *min*. Jika data tersebut lebih kecil dari *min*, maka ia menjadi *min* yang baru. Pada akhir pembacaan data (ditandai 9999), *min* menjadi nilai terkecil dari seluruh data *integer* yang dimasukkan.

```
PROGRAM CariMinimum2
{ Menentukan bilangan terkecil dari sejumlah data integer yang dibaca
dari papan ketik. Akhir pembacaan data = 9999. }

DEKLARASI
   x   : integer   { data yang dibaca }
   min : integer   { data terkecil/minimum }

ALGORITMA:
  read(x) { baca data pertama }
  if x ≠ 9999 then  { periksa, data pertama mungkin 9999 }
    min ← x      {asumsikan min adalah data pertama}
    read(x)    {baca data kedua }
```

```
    while x ≠ 9999 do
      if x < min then
        min ← x
      endif
      read(x)
    endwhile

    write(min)

  endif
```

Algoritma 8.11 Mencari data terkecil (minimum) dari N buah data *integer* yang dibaca dari papan ketik (2)

Contoh Masalah 8.13: Menghitung jumlah kemunculan bilangan genap

Dibaca sejumlah bilangan bulat yang dibaca secara berulang-ulang, akhir pembacaan adalah 9999. Tulislah algoritma untuk menghitung banyaknya bilangan genap.
Contohnya, bilangan yang kita baca misalkan

5 10 47 2 8 20 23 9999

maka banyaknya data bilangan adalah 4.

Penyelesaian

Algoritma untuk menentukan bilangan genap sudah pernah kita kemukakan di dalam Bab 6. Misalkan data yang dibaca adalah x, maka x adalah bilangan bulat jika:

```
x mod 2 = 0
```

Setaip kali x yang dibaca adalah bilangan genap, maka naikkan pencacah jumlahnya dengan 1.

```
PROGRAM JumlahBilanganGenap
{ Menghitung jumlah kemunculan data bilangan genap. Data dibaca dari
papan ketik. }

DEKLARASI
   x  : integer        { bilangan bulat yang dibaca }
   nx : integer        { jumlah kemunculan data bilangan genap }

ALGORITMA:
   nx ← 0
   read(x)
   while (x ≠ 9999) do

      if x mod 2 = 0 then     { x adalah bilangan genap }
         nx ← nx + 1
      endif

      read(x)
```

```
endwhile
{ x = 9999 }

write(nx)
```

Algoritma 8.12 Menghitung jumlah kemunculan data bilangan genap.

Contoh Masalah 8.14: Menulis syair lagu "Anak Ayam"

Tulislah algoritma untuk menuliskan teks lagu *Anak Ayam Turun N* dengan *N* adalah jumlah anak ayam semula (nilai *N* positif dan dibaca terlebih dahulu).

Contoh: *N* = 10, maka sair lagu *Anak Ayam Turun 10* tercetak seperti di bawah ini (perhatikan baris terakhir sedikit berbeda dengan baris sair sebelumnya):

```
Anak Ayam Turun 10
Anak ayam turun 10, mati satu tinggal 9
Anak ayam turun 9, mati satu tinggal 8
Anak ayam turun 8, mati satu tinggal 7
Anak ayam turun 7, mati satu tinggal 6
Anak ayam turun 6, mati satu tinggal 5
Anak ayam turun 5, mati satu tinggal 4
Anak ayam turun 4, mati satu tinggal 3
Anak ayam turun 3, mati satu tinggal 2
Anak ayam turun 2, mati satu tinggal 1
Anak ayam turun 1, mati satu tinggal induknya.
```

Penyelesaian

Masalah ini adalah mencetak *string* "Anak ayam turun *k*, mati satu tinggal *i* – 1" di dalam badan pengulangan. Pada awalnya, *i* = *N*. Setiap kali pengulangan, nilai *i* selalu dikurangi satu. Ketika *i* = 1, pencetakan *string* ditangani secara khusus, karena baris terakhir dari lagu tersebut berbeda dengan baris sebelumnya.

```
PROGRAM LaguAnakAyam
{ Mencetak lagu "Anak Ayam" }

DEKLARASI
   N : integer  { > 0 }

ALGORITMA:
  read(N)
  write('  Anak Ayam Turun ', N)    { cetak judul lagu }
  for i ← N downto 2 do
    write('Anak ayam turun ',i,', mati satu tinggal ',i-1)
  endwhile

  { bait baris terakhir }
  write('Anak ayam turun 1, mati satu tinggal induknya.')
```

Algoritma 8.13 Mencetak lagu "Anak Ayam".

Contoh Masalah 8.15: Menghitung deret pecahan

Tulislah algoritma untuk menghitung jumlah deret pecahan

$$1 - 1/3 + 1/5 - 1/7 + 1/9 + ... \pm 1/N$$

N adalah bilangan bulat positif (dibaca terlebih dahulu).

Penyelesaian

Setiap pecahan a/b terdiri atas pembilang, a, dan penyebut, b. Perhatikan keteraturan deret tersebut: penyebut pada tiap suku (*term*) bertambah 2 dari penyebut suku sebelumnya. Tanda suku berselang-seling, positif, negatif, positif, negatif, dan seterusnya. Tanda positif adalah suku ke-i ganjil (i = 1, 3, 5, ...) dan tanda negatif untuk suku ke-i genap (i = 2, 4, 6, ...). Bilangan ganjil dan genap dapat ditentukan dengan menggunakan operator mod.

Karena setiap suku berbentuk $1/p$, maka setiap penyebut suku ke-k dapat ditulis dalam rumus umum:

$$\text{suku ke-}i = 1/p \quad ; p = 1, 3, ..., N$$

Pengulangan dilakukan selama $p \leq N$. Perhatikan bahwa jika N ganjil, maka penyebut suku terakhir = N, tetapi jika N genap, maka penyebut suku terakhir adalah $N - 1$. Hasil penjumlahan bertipe bilangan riil dan disimpan di dalam peubah jumlah yang bertipe riil.

```
PROGRAM PenjumlahanDeretPecahan
{ Menjumlahkan deret yang berbentuk pecahan:
        1 + 1/3 - 1/5 + 1/7 - 1/9 + ... ± 1/N
Nilai N dibaca ditentukan terleih dahulu. }

DEKLARASI
   jumlah : real              { jumlah deret }
   N      : integer           { penyebut pada suku terakhir, N > 0 }
   i      : integer           { suku ke-i }
   p      : integer           { penyebut suku ke-i }

ALGORITMA:
   read(N)
   jumlah ← 0
   i ← 1          { suku pertama }
   p ← 1          { penyebut suku pertama }
   while p ≤ N do
      if i mod 2 = 1 then          { suku ke-i ganjil }
         jumlah ← jumlah + 1/p
      else                         { suku ke-i genap }
         jumlah ← jumlah - 1/p
      endif
      i ← i + 1        { tinjau suku berikutnya }
      p ← p + 2        { tinjau penyebut suku berikutnya }
   endwhile
   { p > N }

   write(jumlah)
```

Algoritma 8.14 Menghitung jumlah deret pecahan (versi 1).

Algoritma 8.14 dapat disederhanakan lagi dengan mengingat bahwa tanda suku berselang-seling (analog dengan *on/off/on/off* ...). Kita definisikan sebuah nama peubah bernama `tanda`. Untuk suku pertama, nilainya:

```
tanda ← +1      {positif}
```

maka suku-suku berikutnya bertanda

```
tanda ← - tanda        { -1 × harga tanda suku sebelumnya }
```

Dengan demikian, pemeriksaan suku ke-*k* ganjil atau genap dengan notasi if-then dapat dihilangkan.

```
PROGRAM PenjulahanDeretPecahan
{ Menjumlahkan deret yang berbentuk pecahan:
        1 + 1/3 - 1/5 + 1/7 - 1/9 + ... ± 1/N
Nilai N dibaca ditentukan terleih dahulu. }

DEKLARASI
   jumlah : real                { jumlah deret }
   N      : integer             { penyebut pada suku terakhir, N > 0 }
   p      : integer             { penyebut suku ke-i }

ALGORITMA:
   read(N)
   jumlah ← 0
   tanda ← +1          { tanda suku pertama }
   p ← 1
   while p ≤ N do
      jumlah ← jumlah + (tanda * 1/x)
      p ← p + 2
      tanda ← - tanda          { tanda suku berikutnya }
   endwhile
   { p  > N }

   write(jumlah)
```

Algoritma 8.15 Menghitung jumlah deret pecahan.

Contoh Masalah 8.16: Menghitung perpangkatan, a^n (1)

Buatlah algoritma yang membaca a dan n bilangan bulat ($n \geq 0$) lalu untuk menghitung nilai perpangkatan a^n . Perpangkatan a^n berarti mengalikan a sebanyak n kali:

$$a^n = a \times a \times a \times a \times ... \times a \quad \{\text{sebanyak } n \text{ kali}\}$$

Contohnya,

$5^4 = 5 \times 5 \times 5 \times 5 = 625$

$2^8 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$

$3^0 = 1$

Penyelesaian

Misalkan nilai perkalian disimpan di dalam peubah bernama p. Peubah p diinisialisasi dengan 1 (bukan 0), karena ia akan selalu dikalikan dengan a (jika p diinisialisasi dengan 0 maka hasil perkalian akan selalu 0):

```
p ← 1
```

Lakukan perkalian p dengan a sebanyak *n* kali:

```
p ← p * a
```

```
PROGRAM Perpangkatan
{ Menghitung perpangkatan a^n, a dan bilangan bulat, dan n ≥ 0. }

DEKLARASI
  a       : real      { nilai yang dipangkatkan }
  n       : integer   { pemangkat }
  p       : real      { hasil perpangkatan }
  i       : integer   { pencacah pengulangan }

ALGORITMA:
  read(a,n)
  p ← 1
  for i ← 1 to n do
    p ← p * a
  endfor
  write(p)
```

Algoritma 8.16 Menghitung perpangkatan a^n, n bilangan bulat ≥ 0.

Contoh Masalah 8.17: Menghitung perpangkatan, a^n (2)

Kembangkan algoritma pada Contoh Masalah 8.16 di atas sehingga juga dapat digunakan untuk menghitung a^n dengan *n* bilangan bulat sembarang (boleh positif, negatif, atau nol):

$a^n = a \times a \times a \times a \times ... \times a$ { sebanyak *n* kali }

$a^{-n} = 1/a^n = 1/(a \times a \times a \times a \times ... \times a)$ { sebanyak *n* kali }

Contohnya,

$3^4 = 3 \times 3 \times 3 \times 3 = 81$

$5^{-2} = 1/5^2 = 1/(5 \times 5) = 1/25 = 0.25$

$7^0 = 1$

Penyelesaian

Penyelesaiannya sama seperti Contoh Masalah 8.16, hanya saja jika *n* negatif, maka *n* diubah terlebih dahulu menjadi positif (disimpan ke dalam peubah lain). Setelah perpangkatan selesai dihitung, maka jika *n* negatif, hasil perpangkatan dinyatakan dalam bentuk 1/hasil.

```
PROGRAM Perpangkatan
{ Menghitung perpangkatan a^n, a dan n bilangan bulat sembarang. }

DEKLARASI
  a       : real      { nilai yang dipangkatkan }
  n, m    : integer   { pemangkat }
  p       : real      { hasil perpangkatan }
  i       : integer   { pencacah pengulangan }

ALGORITMA:
  read(a,n)
  { hitung perpangkatan }
  p ← 1
  for i ← 1 to ABS(n) do
     p ← p * a
  else

  { tuliskan hasil perpangkatan, bergantung pada n }
  if n < 0 then
    write(1/p)
  else
    write(p)
  endif
```

Algoritma 8.17 Menghitung perpangkatan a^n, n bilangan bulat sembarang.

Keterangan:
ABS adalah fungsi baku untuk mengambil nilai mutlak dari sebuah bilangan. Kita asumsikan fungsi ini sudah tersedia.

Contoh Masalah 8.18: Menghitung faktorial, n!

Faktorial sebuah bilangan bulat tidak negatif n didefinisikan sebagai

$$n! = 1 \times 2 \times 3 \times 4 \times \ldots \times n \quad , n > 0$$

dan khusus untuk $n = 0$, maka faktorial dari 0 didefinisikan sebagai

$$0! = 1$$

Tulislah algoritma yang membaca nilai n dan menghitung faktorial n.

Penyelesaian

Misalkan nilai faktorial disimpan di dalam peubah `fak`.
Inisialisasi `fak` dengan 1.

```
fak ← 1
```

Kalikan `fak` dengan `i`, mulai dari 1 sampai n:

```
fak ← fak * i
```

```
PROGRAM Faktorial
{ Menghitung n! untuk n bilangan bulat tidak negatif. }
DEKLARASI
   n   : integer      { n ≥ 0 }
   fak : integer      { nilai faktorial bilangan n }
   i   : integer      { pencatat pengulangan }

ALGORITMA:
   read(n)
   fak ← 1
   for i ← 1 to n do
      fak ← fak * i
   endfor
   write(fak)
```

Algoritma 8.18 Menghitung n! .

Perhatikan Algoritma 8.18 di atas; jika *n* sama dengan 0, maka kalang *FOR* tidak dimasuki sehingga `fak` tetap sama dengan 1 (sesuai dengan definisi bahwa 0! = 1).

Contoh Masalah 8.19: Menghitung nilai rata-rata untuk setiap mahasiswa

Jurusan tertentu di sebuah Universitas mempunyai *N* orang mahasiswa. Tiap mahasiswa mengambil *m* buah mata kuliah yang sama. Tulislah algoritma yang membaca nama-nama mahasiswa beserta daftar nilai mata kuliah yang dia ambil (nilai bertipe rill), lalu menghitung rata-rata nilai setiap mahasiswa.

Contoh: $N = 3$, $m = 4$

Nama	MK 1	MK 2	MK 3	MK 4	Rata-rata
Ahmad	40.0	80.0	20.0	60.0	50.0
Santi	45.0	75.0	100.0	60.0	70.0
Kuncoro	90.0	35.0	65.0	60.0	60.0

Keterangan:
MK = Mata Kuliah

Penyelesaian

```
PROGRAM NilaiRataRataSetiapMahasiswa
{ Menentukan nilai rata-rata setiap mahasiswa. }

DEKLARASI
  N      : integer   { jumlah mahasiswa > 0 }
  m      : integer   { jumlah mata kuliah > 0 }
  nama   : string    { nama mahasiswa }
  nilai  : real      { nilai ujian }
  jumlah : real      { jumlah nilai ujian }
  rerata : real      { rata-rata nilai ujian }
  i, j   : integer   { pencacah pengulangan }
```

```
ALGORITMA:
  read(N, m)
  i ← 1
  while i ≤ N do
     read(nama)
     j ← 1      { mulai dari mata kuliah pertama }
     jumlah ← 0;
     while j ≤ m do
       read(nilai)
       jumlah ← jumlah + nilai
       j ← j + 1    { tinjau untuk mata kuliah berikutnya }
     endwhile
     { j > m }

     { hitung rata-rata seluruh nilai }
     rerata ← jumlah/m
     write(rerata)

     i ← i + 1  { tinjau untuk mahasiswa berikutnya }
  endwhile
  { i > N }
```

Algoritma 8.19 Menghitung nilai rata-rata ujian setiap mahasiswa.

Contoh Masalah 8.20: Validasi *Password*

Buatlah algoritma yang meniru mekanisme pembacaan sandi-lewat (*password*) untuk masuk ke sebuah sistem (ATM, *server*, reaktor nuklir, lemari besi, dan sebagainya). Apabila sandi-lewat yang dibaca salah, maka pembacaan sandi-lewat hanya boleh diulang maksimum 3 kali. Sandi-lewat yang benar disimpan di dalam algoritma sebagai konstanta (dalam praktek, sandi-lewat disimpan di dalam basis data yang terenkripsi untuk menjaga keamanannya).

Penyelesaian

Kita menggunakan peubah `count` untuk mencacah jumlah pembacaan sandi-lewat. Setiap kali pembacaan sandi-lewat, `count` dinaikkan 1. Jika `count` = 3, maka pembacaan berikutnya tidak diperbolehkan.

```
PROGRAM ValidasiPassword
{ Mensimulasikan mekanisme pembacaan sandi-lewat (password). Maksimum
pembacaan sandi-lewat adalah tiga kali }

DEKLARASI
  const password = 'abc123'   {sandi-lewat}
  SandiLewat : string     {sandi-lewat yang dimasukkan oleh pengguna}
  sah : boolean           {true jika password benar, false jika salah}
  count : integer         { pencatat jumlah pembacaan sandilewat }

ALGORITMA:
  count ← 1
```

```
sah ← false
while (not sah) and (count ≤ 3) do
   read(SandiLewat)
   if SandiLewat = password then
      sah ← true
   else { SandiLewat ≠ password }
      count ← count + 1
   endif
endwhile
{ count > 3 }
```

Algoritma 8.20 Validasi pembacaan *password*.

Contoh Masalah 8.21

Buatlah algoritma yang mengonversi bilangan bulat tidak-negatif dalam sistem desimal menjadi bilangan dalam sistem biner. Misalnya, $18_{10} = 10010_2$.

Penyelesaian

Untuk mendapatkan bentuk biner dari sebuah bilangan desimal, bagi bilangan desimal tersebut dengan 2 terus menerus sampai hasil pembagian = 0. Deretan sisa hasil pembagian menyatakan bilangan biner dari bilangan desimal tersebut.

Contoh:
18 / 2 = 9 sisa 0
9 / 2 = 4 sisa 1
4 / 2 = 2 sisa 0
2 / 2 = 1 sisa 0
1 /2 = 0 sisa 1

Urutan bilangan biner sesuai arah panah: 10010

Misalkan bilangan yang akan dikonversikan ke biner adalah x. Algoritma pengonversiannya adalah membagi x dengan 2 secara berulang-ulang sampai hasil pembagiannya 0. Urutan pembagiannya adalah menghitung sisa pembagian terlebih dahulu:

```
sisa ← x mod 2
```

baru kemudian menghitung hasil pembagian

```
x ← x div 2
```

Bit biner – yang merupakan sisa pembagian ditampung dalam sebuah peubah *string* yang bernama biner. Inisialisasi biner pertama kali dengan karakter kosong (''). Setiap kali diperoleh sisa pembagian, sambungkan (dengan operator '+') sisa pembagian tersebut dengan biner:

```
biner ← '0' + biner     { jika sisa = 0 }
biner ← '1' + biner     { jika sisa = 1 }
```

Algoritma selengkapnya adalah seperti di bawah ini:

```
PROGRAM DesimalKeBiner
{ Mengkonversi bilangan bulat tidak-negatif dalam sistem desimal
menjadi bilangan dalam sistem biner }

DEKLARASI
  X, sisa  : integer
  biner : string     { string yang menampung bit biner }

ALGORITMA:
  read(x)
  biner ← ''    { string kosong, bukan spasi }
  repeat
     sisa ← x mod 2
     x ← x div 2
     case sisa
       0 : biner ← '0' + biner    { operasi penyambungan string }
       1 : biner ← '1' + biner    { operasi penyambungan string
     endcase

  until x = 0

  write(biner)
```

Algoritma 8.21 Konversi bilangan desimal ke bilangan biner

Contoh Masalah 8.22

Tulislah program dalam bahasa *Pascal* dan *C* untuk mencetak segitiga bintang berikut ini jika diberikan tinggi segitiga adalah N (asumsikan $N > 0$) Contohnya, jika $N = 5$, maka segitiga yang dihasilkan adalah

```
*
**
***
****
*****
****
***
**
*
```

Penyelesaian

Perhatikan gambar segitiga untuk contoh $N = 5$ pada soal. Baris ke-1 terdiri dari 1 bintang, baris ke-2 dua bintang, baris ke-3 tiga bintang, ..., dan seterusnya baris ke-5 lima bintang. Secara umum, baris ke-i terdiri dari i bintang. Setelah baris ke-N, baris-baris berikutnya merupakan pencerminan dari baris $N - 1$ sampai baris 1.

```
PROGRAM CetakSegitigaBintang;
{ Mencetak segitiga bintang dengan tinggi segitiga = N }

(* DEKLARASI *)
var
  N, i, j  : integer;

 (* ALGORITMA: *)
begin
  write(' Tinggi segitigam N = ? '); readln(N)

  { cetak bagian segitiga dari baris 1 sampai N }
  for i:=1 to N do
   begin
     for j:=1 to i
       write('*');
     {endfor}
     writeln;
   end;

  { cetak bagian yang merupakan pencerminan dari setengah
    segitiga pertama }
  for i:=N-1 downto 1 do
   begin
     for j:=1 to i
       write('*');
     {endfor}
     writeln;
   end;
end.
```

Algoritma 8.22 Mencetak segitiga bintang (dalam bahasa *Pascal*)

```
/* PROGRAM CetakSegitigaBintang */
/* Mencetak segitiga bintang dengan tinggi segitiga = N  */

#include <stdio.h>

main()
{

  /* DEKLARASI */
  int N, i, j;

  /* ALGORITMA: */
  printf(" Tinggi segitigam N = ? "); scanf("%d", &N)

  /* cetak bagian segitiga dari baris 1 sampai N */

  for (i=1; i<=N; i++)
   {
     for (j=1; j<=i; j++)
       printf("*");
     /*endfor*/
     printf("/n");
   }
```

```
  /* cetak bagian yang merupakan pencerminan dari setengah
     segitiga pertama */
  for (i=N-1; i<=1; i--)
    {
      for (j=1; j<=i; j++)
        printf("*");
      /*endfor*/
      printf("/n");
    }
}
```

Algoritma 8.23 Mencetak segitiga bintang (dalam bahasa *C*)

9

Pengantar Pemrograman Modular

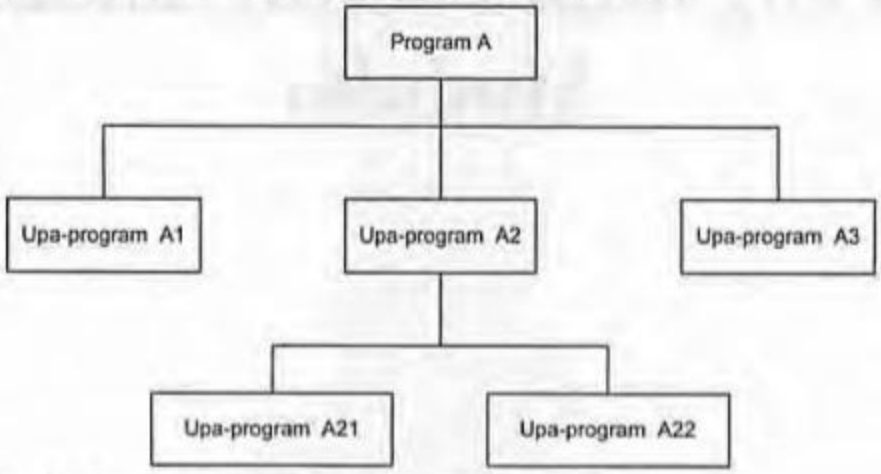
Untuk mencapai suatu tujuan besar, maka tujuan tersebut harus dibagi-bagi menjadi tujuan kecil sampai tujuan kecil itu merupakan tujuan yang dapat dicapai berdasarkan kondisi dan potensi yang dimiliki saat itu
(Al-Khuwarizmi)

Ucapan Al-Khuwarizmi di atas sangat mengena dalam kegiatan memprogram. Program yang besar lebih sulit dimengerti (dibaca) dan lebih sulit lagi dalam melakukan pelacakan kesalahan (jika ada). Oleh karena itu, program sebaiknya dipecah menjadi beberapa upa-program (*subprogram*) yang lebih kecil. Setiap upa-program melakukan komputasi yang spesifik. Upa-program yang baik adalah upa-program yang independen dari program utama sehingga programnya dapat dirancang tanpa mempertimbangkan konteks di mana ia digunakan. Dengan kata lain, pemrogram tidak perlu mempermasalahkan *bagaimana* upa-program tersebut dilakukan, tetapi cukup memikirkan *apa* yang ia lakukan [KER88]. Upa-program yang bagus menyembunyikan detail operasi dari bagian program yang tidak perlu tahu tentang upa-program tersebut. Teknik pemecahan program menjadi sejumlah upa-program dinamakan teknik **pemrograman modular** (*modular programming*). Beberapa bahasa pemrograman menamakan upa-program dengan sebutan **sub-rutin** (*subroutine*), **modul**, **prosedur**, atau **fungsi**.

9.1 Contoh Pemrograman Modular

Gambar 9.1 adalah struktur pohon yang memperlihatkan dekomposisi program A menjadi beberapa buah upa-program. Pada aras 1, program

dipecah menjadi tiga buah upa-program, A1, a2, dan A3. Jika upa-program dianggap masih terlalu besar, maka upa-program tersebut mungkin perlu dipecah lagi menjadi upa-program yang lebih spesifik. Misalnya upa-program A2 dipecah menjadi upa-program A21 dan A22.



Gambar 9.1 Dekomposisi program menjadi beberapa buah upa-program.

Sebagai ilustrasi pemecahan program menjadi sejumlah upa-program, tinjau kembali program pertukaran nilai *A* dan *B* dalam Bahasa C (yang diambil dari Contoh 5.3) yang disalin lagi seperti di bawah ini:

```
/* PROGRAM Pertukaran */
/* Mempertukarkan nilai A dan B. Nilai A dan B dibaca terlebih dahulu. */

#include <stdio.h>

main()
{
   /* DEKLARASI */
   int A, B, temp;

   /* ALGORITMA: */

   /* baca nilai A dan B */
   printf("A = ?"); scanf("%d", &A);
   printf("B = ?"); scanf("%d", &B);

   /* proses pertukaran */
   temp = A;
   A = B;
   B = temp;

   /* Tulis nilai A dan B setelah pertukaran */
   printf("A = %d \n", A);
   printf("B = %d \n", B);
}
```

Algoritma 9.1 Program mempertukarkan nilai A dan B (dalam bahasa C)

Program pertukaran di atas dapat kita pecah menjadi tiga buah upa-program yang spesifik, yaitu upa-program untuk membaca data (*A* dan *B*), upa-program untuk melakukan pertukaran nilai *A* dan *B*, dan upa-program untuk mencetak nilai *A* dan *B* setelah pertukaran. Ketiga upa-program tersebut adalah seperti di bawah ini:

```
/* Upa-program pertama */
void Baca(int *A, int *B)
/* Membaca nilai A dan B. */
{
  /* ALGORITMA:*)
   /* baca nilai A dan B */
   printf("A = ?"); scanf("%d", &A);
   printf("B = ?"); scanf("%d", &B);
}

/* Upa-program kedua */
void Tukar(int *A, int *B)
/* Mempertukarkan nilai A dan B. */
{
  /* DEKLARASI */
  int temp;     { peubah bantu }

  /* ALGORITMA: */
   temp = *A;
   *A = *B;
   *B = temp;
}

/* Upa-program ketiga */
void Tulis(int A, int B)
/* Mencetak nilai A dan B */
{
  /* ALGORITMA: */
  printf("A = %d \n", A);
  printf("B = %d \n", B);
}
```

Masing-masing upa-program mempunyai struktur yang serupa dengan program utama: ada judul upa-program, ada bagian deklarasi, dan bagian algoritma. Setiap upa-program dapat disimpan di dalam berkas terpisah atau bersatu dengan berkas program utama (lihat Algoritma 9.2). Dengan pemecahan program menjadi sejumlah upa-program, program utama terlihat menjadi lebih sederhana. Program utama hanya perlu mendeklarasikan ketiga upa-program tersebut sebelum mereka digunakan (program utama dinyatakan sebagai bagian yang diarsir gelap pada Algoritma 9.2). Upa-program dieksekusi dengan memanggil namanya. Ketika sebuah upa-program dipanggil, pelaksanaan program sekarang berpindah ke upa-program tersebut. Selanjutnya seluruh instruksi di dalam upa-program dilaksanakan. Setelah seluruh instruksi di dalam upa-program selesai dikerjakan, pelaksanan program kembali berpindah ke program pemanggil (dalam hal ini program utama), untuk melaksanakan instruksi berikutnya.

```
/* PROGRAM Pertukaran */
/* Mempertukarkan nilai A dan B. Nilai A dan B dibaca terlebih dahulu. */
#include <stdio.h>
void Baca(int *A, int *B)
void Tukar(int *A, int *B)
```

```
void Tulis(int A, int B)

main()  /* program utama */
{
   /* DEKLARASI */
   int A, B;
   /* ALGORITMA: */
   Baca(A,B);  /* baca nilai A dan B */
   Tukar(&A,&B); /* proses pertukaran */
   Tulis(A,B); /* Tulis nilai A dan B setelah pertukaran */
}

void Baca(int *A, int *B)
/* Membaca nilai A dan B. */
{
  /* ALGORITMA:*)
   /* baca nilai A dan B */
   printf("A = ?"); scanf("%d", &A);
   printf("B = ?"); scanf("%d", &B);
}

void Tukar(int *A, int *B)
/* Mempertukarkan nilai A dan B. */
{
  /* DEKLARASI */
  int temp;     { peubah bantu }

  /* ALGORITMA: */
   temp = *A;
   *A = *B;
   *B = temp;
}

void Tulis(int A, int B)
/* Mencetak nilai A dan B */
{
  /* ALGORITMA: */
  printf("A = %d \n", A);
  printf("B = %d \n", B);
}
```

Algoritma 9.2 Program utama Pertukaran (bagian yang diarsir)

9.2 Keuntungan Pemrograman Modular

Modularisasi program memberikan dua keuntungan. Pertama, untuk aktivitas yang harus dilakukan lebih dari satu kali, modularisasi menghindari penulisan teks program yang sama secara berulangkali. Di sini, upa-program cukup ditulis sekali saja, lalu upa-program dapat dipanggil dari bagian lain di dalam program. Di sini, penggunaan upa-program dapat mengurangi panjang program. Sebagai ilustrasi, tinjau program dalam bahasa *C* di dalam Algoritma 9.3. Hanya bagian algoritma yang sama dan ditulis berulang-ulang yang ditampilkan di sini (bagian yang diarsir). Bagian algoritma lain cukup dinyatakan dengan "...".

```
/* PROGRAM xyz */

#include <stdio.h>

main()
{
   /* DEKLARASI */
   int A, B, C, D, temp

  /* ALGORITMA: */
  ...
  ...

  /* Pertukarkan nilai A dan B */
  temp = A;
  A = B;
  B = temp;
  ...
  ...
  ...

  if (C > D)
  {
     /* Pertukarkan nilai C dan D */
     temp = C;
     C = D;
     D = temp;
  }
  ...
  ...
}
```

Algoritma 9.3 Contoh program yang beberapa kali berisi bagian pertukaran

Di dalam Algoritma 9.3 di atas, ada bagian program yang ditulis berulangkali padahal keduanya melakukan aktivitas yang sama, yaitu pertukaran nilai antara dua buah peubah. Kita dapat menulis bagian yang ditulis berulangkali tersebut ke dalam satu buah upa-program yang diberi nama `Tukar`. Upa-program yang bernama `Tukar` diakses dengan memanggilnya dari dalam program utama. Algoritma 9.3 ditulis kembali dengan penggunaaan upa-program `Tukar` menjadi Algoritma 9.4.

```
/* PROGRAM xyz */
#include <stdio.h>

void Tukar(int *A, int *B);  /* deklarasi upa-program yang digunakan */

main()
{
   /* DEKLARASI */
   int A, B, C, D, temp;
  /* ALGORITMA: */
  ...
  ...

  /* Pertukarkan nilai A dan B */
  Tukar(&A,&B);
  ...
```

```
  ...
  ...
  if (C > D)
  {
     /* Pertukarkan nilai C dan D */
     Tukar(&C,&D);
  }
  ...
  ...
}
void Tukar(int *A, int *B)
/* Mempertukarkan nilai A dan B. */
{
  /* DEKLARASI */
  int temp;     { peubah bantu }

  /* ALGORITMA: */
   temp = A;
   A = B;
   B = temp;
}
```

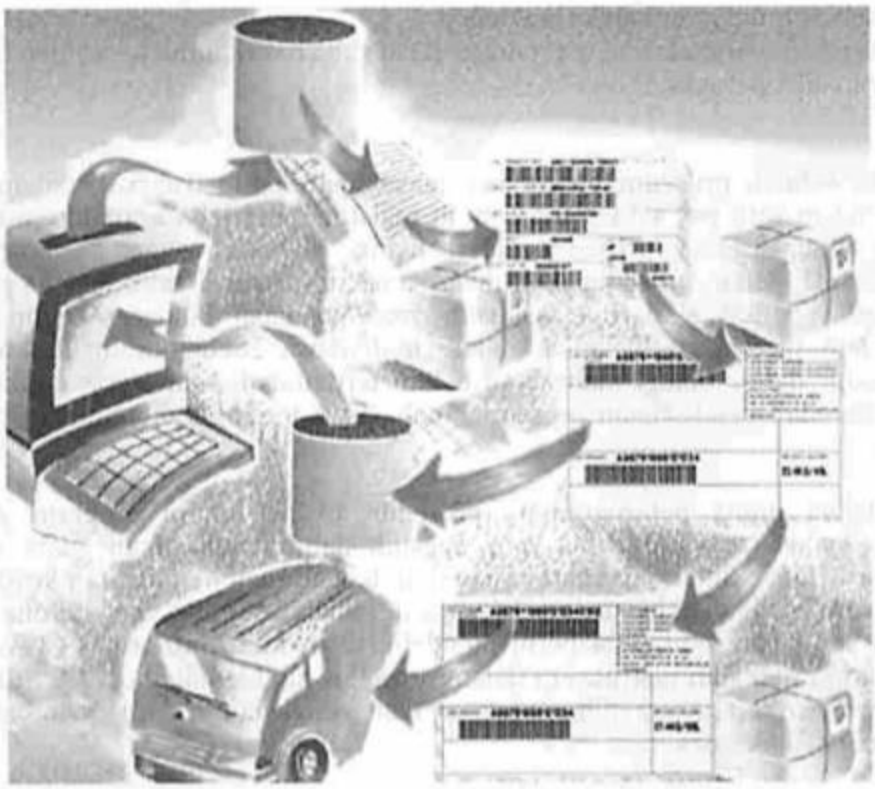
Algoritma 9.4 Contoh program dengan menyatakan bagian pertukaran menjadi upa-program Tukar.

Keuntungan kedua dari modularisasi program adalah kemudahan menulis dan menemukan kesalahan (*debug*) program. Kemudahan menulis akan sangat berguna pada masalah besar yang dikerjakan oleh satu tim pemrogram yang beranggotakan beberapa orang. Masalah yang diprogram dipecah menjadi beberapa masalah yang lebih kecil. Setiap masalah yang lebih kecil tersebut ditulis ke dalam modul yang spesifik dan dikerjakan oleh orang yang berbeda. Satu modul bisa berisi satu atau lebih upa-program. Seluruh modul diintegrasikan menjadi satu buah program yang lengkap. Program yang modular menjadi lebih mudah untuk dibaca dan dimengerti. Program yang tidak modular sulit dipahami, khususnya kalau program tersebut panjang atau terdiri dari puluhan, ratusan, atau ribuan baris instruksi.

Karena setiap upa-program melakukan aktivitas spesifik, maka apabila terdapat kesalahan di dalam program, kesalahan tersebut cukup dilokalisasi di dalam upa-program yang bersangkutan. Sebagian besar program komputer ditulis modular, meskipun program tersebut tidak melibatkan eksekusi yang berulang dari aktivitas yang sama. Pemecahan program menjadi modul-modul program yang lebih kecil umumnya dianggap sebagai praktek pemrograman yang baik.

Terdapat dua bentuk upa-program, pertama **prosedur** (*procedure*) dan kedua **fungsi** (*function*). Struktur setiap upa-program tersebut pada hakikatnya sama dengan struktur program biasa, yaitu ada bagian judul (*header*) yang berisi nama modul, bagian deklarasi, dan badan (*body*) program yang berisi instruksi yang akan dilaksanakan. Pembahasan lengkap mengenai prosedur dan fungsi akan diberikan di dalam Bab 10 dan Bab 11.

10
Prosedur



Prosedur berisi serangkaian kegiatan yang bertujuan spesifik (Sumber: www.fujitsu.com)

Kata "prosedur" sering kita dengar dalam kehidupan sehari-hari. Seorang mahasiswa pada setiap awal semester selalu melakukan pendaftaran ulang (registrasi). Langkah- langkah pendaftaran ulang lazim dinyatakan dalam sebuah prosedur yang dinamakan prosedur daftar ulang. Secara sederhana prosedur daftar ulang dituliskan sebagai berikut:

Prosedur Daftar Ulang

1. Ambil *Form* Rencana Studi (FRS) di kantor Tata Usaha Akademik dengan memperlihatkan Kartu Tanda Mahasiswa (KTM) dan Kartu Studi Mahasiswa (KSM).
2. Lakukan pembayaran SPP di loker pembayaran dan minta kwitansinya.
3. Isi FRS dengan mata kuliah – mata kuliah yang akan diambil di semester ini.
4. Lakukan perwalian untuk mengesahkan FRS oleh Wali Akademik.
5. Jika SPP sudah lunas, maka serahkan FRS yang sudah disahkan oleh Wali Akademik ke petugas di Kantor Tata Usaha Akademik. Jika SPP belum lunas, kembali ke langkah 2.
6. Serahkan foto ukuran 2 × 3 untuk KTM yang baru, untuk kemudian di-cap oleh petugas.
7. Selesai

Ketika sebuah prosedur dieksekusi, maka instruksi-instruksi di dalamnya dikerjakan satu per satu. Pada contoh registrasi mahasiswa, mahasiswalah yang mengeksekusi prosedur daftar ulang. Mahasiswa membaca setiap langkah di dalam prosedur, lalu melaksanakan instruksi yang tertulis pada setiap langkah. Efek dari pelaksanaan prosedur dapat diamati sebelum dan sesudah prosedur dijalankan oleh mahasiswa. Sebelum pelaksanaan prosedur daftar ulang, mahasiswa belum terdaftar di semester yang baru, dan sesudah pelaksanaan prosedur, mahasiswa terdaftar di semester yang baru.

Di dalam dunia pemrograman, prosedur adalah modul program yang mengerjakan tugas/aktivitas yang spesifik dan menghasilkan suatu efek *netto* [LIE96]. Suatu efek *netto* diketahui dengan membandingkan keadaan awal dan keadaan akhir pada pelaksanaan sebuah prosedur. Oleh karena itu, pada setiap prosedur kita perlu mendefinisikan **keadaan awal** (K.Awal) sebelum rangkaian instruksi di dalam prosedur dilaksanakan dan **keadaan akhir** (K.Akhir) yang diharapkan setelah rangkaian instruksi di dalam prosedur dilaksanakan.

10.1 Pendefinisian Prosedur

Pendefinisian prosedur artinya menuliskan nama prosedur, mendeklrasikan nama-nama konstanta, peubah dan tipe (jika ada), dan menjabarkan rangkaian aksi yang dilakukan. Pada dasarnya, struktur prosedur sama dengan struktur algoritma yang sudah Anda kenal, yaitu: ada bagian judul (*header*) yang terdiri atas nama prosedur dan deklarasi parameter (jika ada), bagian deklarasi untuk mengumumkan nama-nama, dan bagian algoritma yang disebut badan prosedur. Setiap prosedur mempunyai nama yang unik. Nama prosedur sebaiknya diawali dengan kata kerja karena prosedur berisi

suatu aktivitas, misalnya `HitungLuas`, `Tukar`, `CariMaks`, `Inisialisasi`, `AktifkanMenu`, dan lain sebagainya.

Parameter adalah nama-nama peubah yang dideklaraskan pada bagian *header* prosedur. Sebagian besar program memerlukan pertukaran data/informasi antara prosedur (atau fungsi) dan titik di mana ia dipanggil. Penggunaan parameter menawarkan mekanisme pertukaran informasi tersebut. Tiap *item* data ditransfer antara **parameter aktual** dan **parameter formal** yang bersesuaian. Parameter aktual (kadang-kadang disebut juga **argumen**) adalah parameter yang disertakan pada waktu pemanggilan prosedur, sedangkan parameter formal adalah parameter yang dideklarasikan di dalam bagian *header* prosedur itu sendiri. Ketika prosedur dipanggil, parameter aktual menggantikan parameter formal. Tiap-tiap parameter aktual berpasangan dengan parameter formal yang bersesuaian.

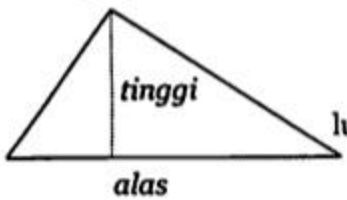
Notasi algoritmik untuk mendefinisikan prosedur (tanpa parameter) adalah:

```
procedure NamaProsedur (deklarasi parameter, jika ada)
{ Spesifikasi prosedur, berisi penjelasan tentang apa yang dilakukan
oleh prosedur ini.
K.Awal : keadaan sebelum prosedur dilaksanakan.
K.Akhir: keadaan setelah prosedur dilaksanakan. }
DEKLARASI
  { semua nama yang dipakai di dalam prosedur dan hanya berlaku lokal
di dalam prosedur didefinisikan di sini }

ALGORITMA:
   { badan prosedur, berisi urutan instruksi }
```

Algoritma 10.1 Struktur prosedur

Pendeklarasian parameter di dalam prosedur bukanlah keharusan. Dengan kata lain, parameter boleh ada atau tidak ada. Kelak Anda akan memahami kapan diperlukan pendefnisian prosedur dengan parameter. Marilah kita mulai dengan sebuah contoh mendefinisikan prosedur tanpa parameter.

Contoh 10.1. Buatlah prosedur yang membaca panjang alas dan tinggi segitiga, menghitung luas segitiga dengan rumus *luas* = (*alas* × *tinggi*)/2, dan mencetak luas segitiga tersebut.



luas segitiga = (*alas* × *tinggi*)/2

Penyelesaian

```
procedure HitungLuasSegitiga
{ Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }
{ K.Awal : sembarang }
{ K.Akhir: luas segitiga tercetak.}

DEKLARASI
  alas    : real   { panjang alas segitiga, dalam cm }
  tinggi  : real   { tinggi segitiga, dalam cm }
  luas    : real   { luas segitiga, dalam cm² }

ALGORITMA:
  read(alas,tinggi)
  luas ← (alas * tinggi)/2
  write(luas)
```

Algoritma 10.2 Prosedur untuk menghitung luas segitiga

Perhatikan Algoritma 10.2. Keadaan awal (K.Awal) prosedur adalah sembarang kondisi, namun setlah prosedur selesai dilaksanakan, keadaan akhirnya (K.Akhir) adalah kondisi di mana luas segitiga tercetak (misalnya ke layar).

10.2 Pemanggilan Prosedur

Prosedur bukan program yang berdiri sendiri, jadi ia tidak dapat dieksekusi secara langsung. Ini berarti, instruksi-instruksi di dalam prosedur baru dapat dilaksanakan hanya bila prosedur tersebut diakses. Prosedur diakses dengan cara memanggil namanya dari program pemanggil (misalnya dari program utama atau dari modul program lainnya). Jika prosedur tanpa parameter, maka pemanggilannya cukup dengan namanya saja:

NamaProsedur

(cara pemanggilan prosedur dengan parameter akan kita bahas di dalam upa-bab 10.4).

Ketika *NamaProsedur* dipanggil, kendali program berpindah secara otomatis ke prosedur tersebut. Seluruh instruksi di dalam badan prosedur dilaksanakan. Setelah semua instruksi selesai dilaksanakan, kendali program berpindah secara otomatis kembali ke instruksi sesudah pemanggilan prosedur.

Agar nama prosedur dikenal oleh program pemangil, maka di dalam program pemanggil, kita harus mendeklarasikan purwarupa (*prototype*) prosedur tersebut. Purwarupa prosedur hanya berisi bagian *header* prosedur. Pendeklarasian purwarupa juga untuk memberitahu program pemanggil bagaimana cara-cara mengakses prosedur (jumlah parameter dan tipe parameternya, jika ada).

Misalkan kita mempunyai *N* buah segitiga dan kita ingin menghitung luas semua segitiga. Prosedur untuk menghitung luas segitiga sudah kita definisikan di dalam Contoh 10.1.

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   procedure HitungLuasSegitiga
  { Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }

ALGORITMA:
   read(N)   { tentukan banyaknya segitiga }
   for i ← 1 to N do
     HitungLuasSegitiga
   endfor
```

Algoritma 10.3 Menghitung luas N buah segitiga.

10.3 Nama Global, Nama Lokal, dan Lingkup

Nama-nama (konstanta, peubah, tipe, dan lain-lain) yang dideklarasikan di dalam prosedur (termasuk parameter, jika ada) hanya "dikenal" di dalam badan prosedur yang bersangkutan. Nama-nama yang dideklarasikan di dalam prosedur tersebut dikatakan lingkupnya (*scope*) "lokal". Nama-nama lokal hanya berlaku di dalam prosedur yang melingkupinya saja. Setelah prosedur selesai dieksekusi, nama-nama tersebut tidak dikenal lagi di luar prosedur. Tinjau prosedur `HitungLuasSegitiga` pada Contoh 10.2. Peubah `alas`, `tinggi`, dan `luas` hanya dapat digunakan di dalam prosedur yang bersangkutan.

Sebaliknya, nama-nama (konstanta, peubah, tipe, dan lain-lain) yang dideklarasikan di dalam program utama dikatakan lingkupnya "global". Nama-nama global dapat digunakan di bagian manapun di dalam program, baik di dalam program utama maupun di dalam prosedur yang dipanggil. Tinjau program `Segitiga` pada Contoh 10.2. Peubah `i` dan `N` dapat digunakan di dalam program utama maupun di dalam prosedur `HitungLuasSegitiga` (kalau diperlukan).

Sekarang, mari kita modifikasi prosedur `HitungLuasSegitiga` dengan meniadakan peubah `alas`, dan `tinggi` di dalam bagian deklarasinya, dan menghilangkan pernyataan pembacaan data di dalam prosedur.

```
procedure HitungLuasSegitiga
{ Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 }
{ K.Awal : sembarang }
{ K.Akhir: luas segitiga tercetak.}

DEKLARASI
  luas : real
```

```
ALGORITMA:
  luas ← (alas * tinggi)/2
  write(luas)
```

Algoritma 10.4 Modifikasi prosedur menghitung luas segitiga.

Selanjutnya, modifikasi program utama `Segitiga` dengan memasukkan peubah `alas`, dan `tinggi` ke dalam bagian deklarasi program utama, dan membaca `alas` dan `tinggi`:

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   alas, tinggi : real
   procedure HitungLuasSegitiga
  { Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }

ALGORITMA:
   read(N)  { tentukan banyaknya segitiga }
   for i ← 1 to N do
     read(alas, tinggi)
     HitungLuasSegitiga
   endfor
```

Algoritma 10.5 Modifikasi program utama untuk menghitung luas N buah segitiga.

Peubah `alas` dan `tinggi` dideklarasikan di dalam bagian deklarasi program utama. Karena itu, `alas` dan `tinggi` adalah peubah global sehingga mereka juga "dikenal" dan dapat digunakan di dalam prosedur `HitungLuasSegitiga`.

Sebaliknya, peubah luas dideklarasikan di dalam prosedur `HitungLuasSegitiga`, jadi peubah tersebut adalah peubah lokal dan ia hanya dapat digunakan di dalam lingkup prosedur itu saja. Anda tidak dapat menuliskan pernyataan `write(luas)` di dalam program utama.

Apakah menggunakan peubah global atau peubah lokal?

Keputusan apakah suatu peubah akan dideklarasikan global atau lokal bergantung kepada penggunaan nama tersebut. Bila suatu peubah digunakan di seluruh bagian program (termasuk di dalam prosedur), maka peubah tersebut harus dideklarasikan global. Sebaliknya, bila peubah tersebut hanya digunakan di dalam prosedur saja, maka nama peubah "sebaiknya" dideklarasikan sebagai peubah lokal (dikatakan "sebaiknya" karena bila dideklarasikan global pun masih tetap benar).

Usahakanlah menggunakan peubah global sesedikit mungkin. Penggunaan peubah lokal akan memberikan keuntungan, sebab peubah lokal membuat program lebih elegan, dan dapat meminimumkan usaha pencarian

kesalahan yang disebabkan oleh nama tersebut, karena peubah lokal hanya "bermain" di dalam lingkup prosedur saja.

Prosedur yang baik adalah prosedur yang independen dari program pemanggilnya. Pernyataan ini menyiratkan bahwa prosedur yang baik tidak menggunakan peubah-peubah global di dalam badan prosedurnya. Jika program utama perlu mengomunikasikan nilai peubah global ke dalam prosedur, maka ada satu cara untuk melakukannya yaitu dengan menggunakan parameter. Lebih jauh mengenai parameter akan dijelaskan di dalam upa-bab 10.4 berikut ini.

10.4 Parameter

Sebagaimana sudah disebutkan di dalam upa-bab 10.1, kebanyakan program memerlukan pertukaran informasi antara prosedur dan titik di mana ia dipanggil. Penggunaan parameter menawarkan mekanisme pertukaran informasi tersebut.

Prosedur dengan parameter diakses dengan cara memanggil namanya dari program pemanggil (program utama atau modul program lain) disertai parameternya. Parameter yang disertakan pada waktu pemanggilan disebut parameter aktual. Cara pemanggilan prosedur dengan parameter adalah:

```
NamaProsedur(parameter aktual)
```

Ketika prosedur dipanggil, parameter aktual berkoresponden dengan parameter formal (parameter yang dideklaarsikan pada bagian *header* prosedur). Tiap-tiap parameter aktual berpasangan dengan parameter formal yang bersesuaian.

Aturan yang harus diperhatikan dalam korespondensi satu-satu antara parameter aktual dan parameter formal adalah:

1. Jumlah parameter aktual pada pemangilan prosedur harus sama dengan jumlah parameter formal pada deklarasi prosedurnya;
2. Tiap parameter aktual harus bertipe sama dengan tipe parameter formal yang bersesuaian;
3. Tiap parameter aktual harus diekspresikan dalam cara yang taat-asas dengan parameter formal yang bersesuaian, bergantung pada jenis parameter formal (dijelaskan di bawah ini).

Berdasarkan maksud penggunaanya, terdapat tiga jenis parameter formal yang disertakan di dalam prosedur:

1. parameter masukan (*input parameter*)
2. parameter keluaran (*output parameter*)
3. parameter masukan/keluaran (*input/output parameter*)

Parameter masukan adalah parameter yang nilainya berlaku sebagai masukan untuk prosedur. Pada bahasa pemrograman, istilah parameter masukan ini sering dinamakan **parameter nilai** (*value parameter* atau *parameter by value*). **Parameter keluaran** adalah parameter yang menampung keluaran yang dihasilkan oleh prosedur. Sedangkan **parameter masukan/keluaran** adalah parameter yang berfungsi sebagai masukan sekaligus keluaran bagi prosedur tersebut.

10.4.1 Parameter Masukan

Pada parameter masukan, nilai (*value*) parameter aktual diisikan (*assign*) ke dalam parameter formal yang bersesuaian. Nilai ini digunakan di dalam badan prosedur yang bersangkutan. Nilai yang dinyatakan oleh parameter masukan tidak dapat dikirim dalam arah sebaliknya. Itulah alasan mengapa parameter jenis ini diacu sebagai parameter masukan. Perubahan nilai parameter di dalam badan prosedur tidak mengubah nilai parameter aktual. Karena yang dipentingkan adalah nilainya, maka nama parameter aktual boleh berbeda dengan nama parameter formal yang bersesuaian.

Tinjau kembali prosedur menghitung luas segitiga di dalam Algoritma 10.3. Jika program utama ingin mengomunikasikan panjang alas dan tinggi segitiga ke prosedur `HitungLuasSegitiga`, maka peubah alas dan tinggi harus dideklarasikan sebagai parameter formal di bagian *header* prosedur:

```
procedure HitungLuasSegitiga(input alas, tinggi : real)
{ Menghitung luas segitiga dengan rumus Luas = (alas × tinggi)/2 }
{ K.Awal : alas dan tinggi sudah terdefinisi nilainya }
{ K.Akhir: luas segitiga tercetak.}

DEKLARASI
  luas : integer

ALGORITMA:
  luas ← (alas * tinggi)/2
  write(luas)
```

Algoritma 10.6 Prosedur menghitung luas segitiga dengan parameter masukan

Perhatikan Algoritma 10.6. Kata kunci `input` pada bagian *header* menyatakan bahwa alas dan tinggi adalah parameter masukan. Keadaan awal (K.Awal) prosedur adalah kondisi di mana `alas` dan `tinggi` sudah terdefinisi nilainya. Ini berarti bahwa `alas` dan `tinggi` sudah harus sudah berisi nilai sebelum pelaksanaan prosedur. Keadaan akhir (K.Akhir) prosedur adalah kondisi di mana luas segitiga tercetak (misalnya ke layar). Program utama yang memanggil prosedur `HitungLuasSegitiga` harus mendeklarasikan prosedur ini dan memanggilnya dengan parameter aktual yang bersesuaian:

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   alas, tinggi : real
   procedure HitungLuasSegitiga(input alas, tinggi : real)
  { Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }

ALGORITMA:
   read(N)  { tentukan banyaknya segitiga }
   for i ← 1 to N do
     read(alas, tinggi)
     HitungLuasSegitiga(alas,tinggi)
   endfor
```

Algoritma 10.7 Program utama untuk menghitung luas N buah segitiga.

Perhatikan Algorima 10.7. Prosedur `HitungLuasSegitiga` dipanggil dari program utama dengan menyertakan panjang alas dan tinggi segitiga:

```
HitungLuasSegitiga(alas, tinggi)
```

Karena yang dipentingkan adalah nilainya, maka nama parameter aktual tidak harus sama dengan nama parameter formal yang bersesuaian asalkan tipenya tetap sama (lihat Algoritma 10.7).

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   a, t : real
   procedure HitungLuasSegitiga(input alas, tinggi : real)
  { Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }

ALGORITMA:
   read(N)  { tentukan banyaknya segitiga }
   for i←1 to N do
     read(alas, tinggi)
     HitungLuasSegitiga(a,t)
   Endfor
```

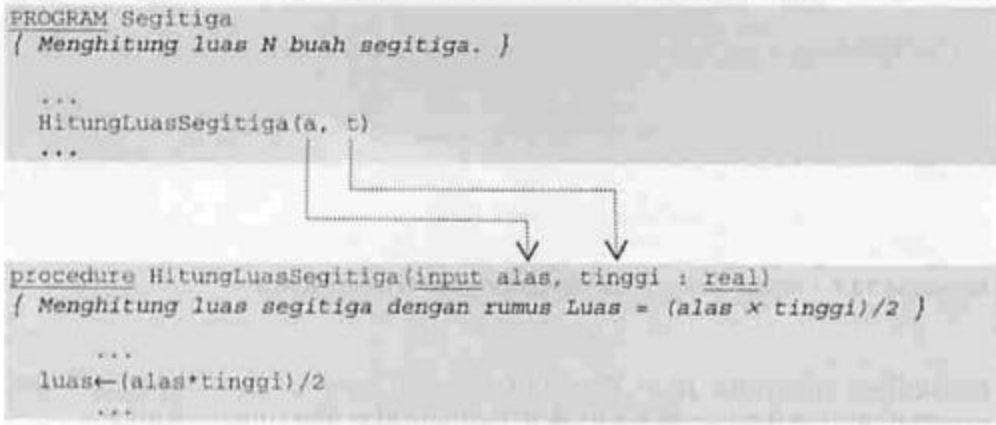
Algoritma 10.8 Program utama untuk menghitung luas N buah segitiga (nama parameter aktual tidak sama dengan nama paramater formal).

Perhatikan Algoritma 10.8. Di sini prosedur `HitungLuasSegitiga` dipanggil dengan menyertakan nilai alas dan tinggi di dalam peubah `a` dan `t`:

```
HitungLuasSegitiga(a,t)
```

Ketika prosedur `HitungLuasSegitiga` dipanggil, maka nilai parameter aktual `a` dan `t` diisikan ke dalam parameter formal alas dan tinggi (lihat Gambar 10.1).

Misalkan nilai a yang dibaca adalah 10 dan nilai t yang dibaca adalah 5, maka parameter alas akan menerima nilai 10 dari a dan parameter tinggi akan menerima 5 dari t. Selanjutnya nilai alas dan tinggi digunakan untuk menghitung luas segitiga di dalam prosedur HitungLuasSegitiga.



Gambar 10.1 Mekanisme korespondensi satu-satu antara parameter aktual dengan parameter formal yang berjenis masukan.

Karena yang dipentingkan adalah nilainya, maka parameter aktual boleh berupa ekspresi atau konstanta. Jadi, pemanggilan berikut:

```
HitungLuasSegitiga(a*0.2, t*0.1)
```

benar (misalnya panjang alas dan tinggi segitiga dikoreksi dengan mengalikannya dengan faktor 0.2 dan 0.1).

Begitu juga pemanggilan

```
HitungLuasSegitiga(12,6)
```

adalah benar.

Contoh 10.2. Buatlah prosedur yang menerima nomor menu program dan menuliskan pesan nomor menu yang dipilih. Misalnya, jika pengguna memilih menu nomor 3, maka tuliskan pesan "Anda memilih menu nomor 3". Jumlah menu seluruhnya ada 5 (lihat Contoh 6.18 di dalam Bab 6).

Penyelesaian

```
procedur AktifkanMenu(input NomorMenu : integer)
{ Mencetak nomor meny yang dipilih }
{ K.Awal: NomorMenu sudah berisi nomor menu yang dipilih }
{ K.Akhir: Pesan yang bersesuaian dengan NomorMenu tercetak. }

DEKLARASI
  { tidak ada }
```

```
ALGORITMA:
   case NomorMenu
     1 : write('Anda memilih menu nomor 1')
     2 : write('Anda memilih menu nomor 2')
     3 : write('Anda memilih menu nomor 3')
     4 : write('Anda memilih menu nomor 4')
     5 : write('Anda memilih menu nomor 5')
   endcase
```

Algoritma 10.10 Mencetak nomor menu program yang dipilih.

Catatan:
pernyataan `write('Anda memilih menu nomor ...')` *dapat diganti dengan pemanggilan prosedur/fungsi yang berkaitan dengan menu yang dipilih.*

10.4.2 Parameter Keluaran

Prosedur mungkin menghasilkan satu atau lebih keluaran yang akan digunakan oleh program pemanggil. Jika ini kasusnya, mak nilai keluaran tersebut ditampung di dalam parameter keluaran. Ketika prosedur yang mengandung parameter keluaran dipanggil, maka nama parameter aktual menggantikan (*substitute*) nama parameter formal yang bersesuaian di dalam prosedur. Selanjutnya, nama parameter aktual akan digunakan selama pelaksanaan prosedur (ini berlawanan dengan parameter masukan, yang dalam hal ini nilai dari parameter aktual yang di-*assign* ke dalam parameter formal). Karena nama parameter merupakan suatu lokasi di memori, maka bila di dalam prosedur parameter aktual diisi suatu nilai, nilai ini akan tetap berada di dalam parameter aktual meskipun prosedur selesai dilaksanakan. Jadi, setelah pemanggilan, parameter aktual berisi suatu nilai yang merupakan keluaran dari prosedur tersebut.

Parameter keluaran dideklarasikan di dalam *header* prosedur, sebagaimana parameter masukan. Tetapi, parameter keluaran harus dideklarasikan dengan kata kunci `output`.

Tinjau prosedur `HitungLuasSegitiga` di dalam Algoritma 10.5. Luas segitiga disimpan di dalam peubah `luas` dan nilainya dicetak di dalam prosedur tersebut. Andaikan kita menginginkan `luas` segitiga dicetak di dalam program pemanggil (jadi, bukan di dalam prosedur), maka kita harus menyatakan `luas` sebagai parameter keluaran, seperti yang dituliskan di dalam Algoritma 10.11 berikut ini:

```
procedure HitungLuasSegitiga(input alas, tinggi : real,
                             output luas : real)
{ Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 }
{ K.Awal : alas dan tinggi sudah terdefinisi nilainya }
{ K.Akhir: luas berisi luas segitiga. }
```

```
DEKLARASI
   { tidak ada }
ALGORITMA:
   luas ← (alas * tinggi)/2
```

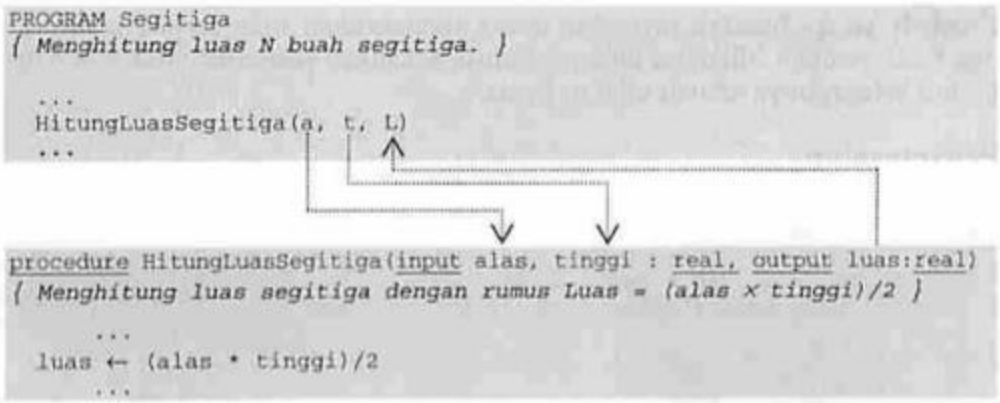
Algoritma 10.11 Prosedur menghitung luas segitiga dengan luas sebagai parameter keluaran

Perhatikan Algoritma 10.11. Kata kunci `input` sebelum alas dan tinggi pada bagian *header* menyatakan bahwa kedua parameter tersebut parameter masukan, dan kata kunci `output` sebelum `luas` menyatakan bahwa `luas` adalah parameter keluaran. Keadaan awal (K.Awal) prosedur adalah kondisi di mana `alas` dan `tinggi` sudah terdefinisi nilainya. Ini berarti bahwa `alas` dan `tinggi` sudah harus sudah berisi nilai sebelum pelaksanaan prosedur. Keadaan akhir (K.Akhir) prosedur adalah kondisi di mana `luas` berisi luas segitiga. Program utama yang memanggil prosedur `HitungLuasSegitiga` harus mendeklarasikan prosedur ini dan memanggilnya dengan parameter aktual yang bersesuaian. Keluaran prosedur, yaitu `luas`, dicetak di dalam program utama:

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   a, t, L : real   { alas, tinggi, dan luas segitiga}
   procedure HitungLuasSegitiga(input alas, tinggi : real,
                                output luas : real)
  { Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }

ALGORITMA:
   read(N)  { tentukan banyaknya segitiga }
   for i ← 1 to N do
     read(a, t)
     HitungLuasSegitiga(a, t, L)
     write(L)
   endfor
```

Algoritma 10.12 Program utama untuk menghitung luas N buah segitiga.

Ketika prosedur `HitungLuasSegitiga` dipanggil, maka nilai parameter aktual `a` dan `t` diisikan ke dalam parameter formal `alas` dan `tinggi` (lihat Gambar 10.2), sedangkan nama parameter aktual `L` menggantikan nama parameter formal `luas` (lihat Gambar 10.3).



Gambar 10.2 Mekanisme korespondensi satu-satu antara parameter actual dengan parameter formal yang berjenis keluaran.

Hal lain yang harus diperhatikan pada jenis parameter keluaran ini adalah parameter aktual harus berupa peubah, tidak boleh berupa konstanta atau ekspresi. Jadi, pemanggilan seperti contoh di bawah ini adalah SALAH:

```
HitungLuasSegitigaIGA(a,t,10)      { karena 10 adalah konstanta }
HitungLuasSegitigaIGA(a,t,10*a)    { karena 10*a adalah ekspresi }
```

Contoh 10.3. Buatlah prosedur untuk menghitung nilai rata-rata *N* buah data bilangan bulat. Data bilangan bulat dibaca dari piranti masukan. Masukan prosedur adalah *N* dan keluarannya adalah nilai rata-rata.

Penyelesaian

Prosedur menghitung nilai rata-rata:

```
procedure HitungRataRata(input N : integer, output u : real)
{ Menghitung rata-rata N buah bilangan bulat yang dibaca dari piranti
  masukan. }
{ K.Awal : N sudah berisi banyaknya bilangan bulat, N > 0. }
{ K.Akhir: u berisi rata-rata seluruh bilangan. }

DEKLARASI
   x : integer    { data bilangan bulat yang dibaca dari papan kunci }
   k : integer    { pencacah banyak bilangan }
   jumlah : integer    { jumlah seluruh bilangan }
ALGORITMA:
   jumlah ← 0    { inisialisasi }
   for k ← 1 to N do
     read(x)
     jumlah ← jumlah + x
   endwhile

   u ← jumlah/N
```

Algoritma 10.13 Menghitung nilai rata-rata .

Contoh 10.4. Buatlah prosedur untuk menentukan nilai terbesar antara dua buah peubah bilangan bulat, *A* dan *B*. Masukan prosedur adalah *A* dan *B*, dan keluarannya adalah nilai terbesar.

Penyelesaian

```
procedure TentukanMaksimum(input A, B : integer, output maks :
integer)
{ Menentukan nilai terbesar dari dua buah peubah, A dan B. }
{ K.Awal : Nilai A dan B sudah terdefinisi. }
{ K.Akhir: maks berisi nilai terbesar antara A dan B. }

DEKLARASI
   { tidak ada }

ALGORITMA:
  if A > B then
    maks ← A
  else
    maks ← B
  endif
```

Algoritma 10.14 Menenetukan nilai terbesar dari dua buah data.

10.4.3 Parameter Masukan/Keluaran

Kita sudah melihat parameter masukan digunakan pada situasi di mana informasi dikirim hanya dari titik pemanggilan prosedur ke prosedur itu sendiri. Sedangkan parameter keluaran hanya mengirim informasi dari prosedur ke titik pemanggilan prosedur. Pada kebanyakan aplikasi, informasi juga harus dikirim dalam kedua arah. Dengan kata lain, prosedur juga harus dapat mengakomodasi baik masukan dari dan keluaran ke blok program pemanggil. Parameter masukan/keluaran umumnya digunakan pada situasi seperti ini.

Ketika prosedur yang mengandung parameter keluaran dipanggil, nama parameter aktual di dalam program pemanggil menggantikan (*substitute*) nama parameter formal yang bersesuaian di dalam prosedur. Selain itu, isi atau nilai parameter aktual juga ikut disalin ke dalam parameter formal. Jadi, nama dan nilai parameter aktual digunakan di seluruh bagian prosedur. Akibat penggunaan parameter masukan/keluaran, bila parameter aktual diubah nilainya di dalam badan prosedur, maka sesudah pemanggilan prosedur nilai parameter aktual di titik pemanggilan juga berubah. Ini berbeda dengan parameter masukan, yang dalam hal ini meskipun nilai parameter aktual di dalam badan prosedur diubah, nilai parameter aktual tersebut tidak berubah di titik pemanggilan.

Parameter masukan/keluaran dideklarasikan di dalam *header* prosedur dengan kata kunci `input/output`. Hal lain yang harus diperhatikan pada

jenis parameter masukan/keluaran ini adalah parameter aktual harus berupa peubah, tidak boleh berupa konstanta atau ekspresi.

Contoh 10.5. Di dalam bahasa *Pascal* terdapat procedur `Inc(x)` yang berkoresponden dengan `x:=x+1`. Tulislah prosedur `Inc` dalam notasi algoritmik.

Penyelesaian

Peubah `x` harus dinyatakan sebagai parameter masukan/keluaran, sebab prosedur `Inc` harus menerima nilai `x` dan menghasilkan nilai `x` yang baru setelah ditambah satu.

```
procedure Inc(input/output x : integer)
{ Menaikkan nilai x sebesar 1. }
{ K.Awal : x sudah terdefinisi nilainya. }
{ K.Akhir: nilai x bertambah 1. }

DEKLARASI

ALGORITMA:
  x ← x + 1
```

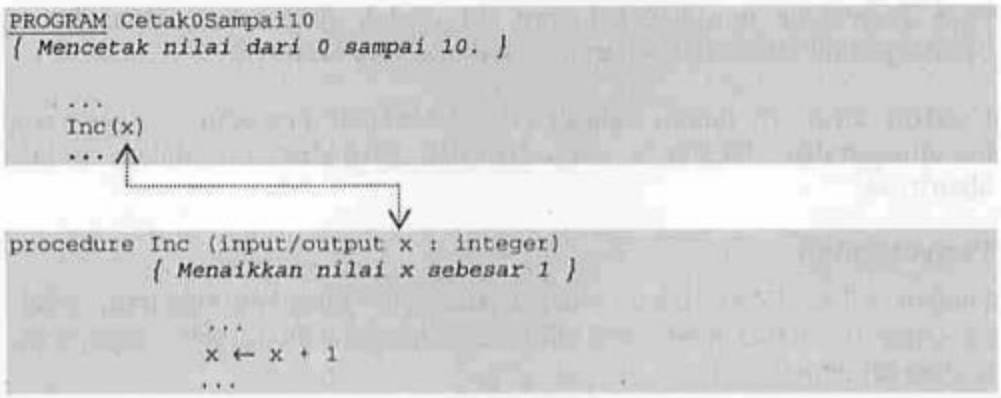
Algoritma 10.15 Menaikkan nilai peubah sebesar 1.

Contoh program yang memanggil prosedur `Inc` ditunjukkan pada Algoritma 10.14. Prosedur mencetak nilai-nilai `x` dari 0 sampai 10.

```
PROGRAM CetakOSampai10
{ Mencetak nilai dari 0 sampai 10. }

DEKLARASI
  x : integer
  procedure Inc(input/output x : integer)
  { Menaikkan nilai x sebesar 1. }

ALGORITMA:
  x ← 0
  repeat
    write(x)
    Inc(x)
  until x > 10
```

Algoritma 10.16 Program utama untuk mencetak 0 sampai 10.

Mekanisme korespondensi satu-satu antara parameter aktual dengan parameter formal yang berjenis masukan/keluaran diperlihatkan pada Gambar 10.3. Arah panah yang bolak-balik menunjukkan bahwa *x* berfungsi sebagai parameter masukan sekaligus parameter keluaran.



Gambar 10.3 Mekanisme korespondensi satu-satu antara parameter actual dengan parameter formal yang berjenis masukan/keluaran.

Untuk memperlihatkan efek penggunaan parameter masukan dan perbandingannya dengan parameter masukan/keluaran:

Prosedur dengan parameter masukan

```
procedure TambahEmpat(input x,y : integer)
{ Menambahkan nilai x dan y masing-masing dengan 4. }
{ K.Awal : x dan y sudah berisi nilai. }
{ K.Akhir: nilai x dan y masing-masing bertambah 4, lalu dicetak. }

DEKLARASI
   { tidak ada }

ALGORITMA:
   x ← x + 4
   y ← y + 4;
   output`nilai x dan y di akhir prosedur Empat:`)
   write(`  x = `, x)
   write(`  y = `, y)
```

Algoritma 10.17 Menambahkan nilai parameter masukan dengan 4.

Program utama:

```
PROGRAM XYZ
{ Program yang memperlihatkan efek penggunaan parameter masukan }

DEKLARASI
  a, b : integer

  procedure TambahEmpat(input x,y : integer)
  { Menambahkan nilai x dan y masing-masing dengan 4 }

ALGORITMA:
  a ← 15
  b ← 10
```

```
  write('nilai a dan b sebelum pemanggilan prosedur TambahEmpat: ')
  write('  a = ', a)
  write('  b = ', b)

  TambahEmpat(a,b)

  write('nilai a dan b sesudah pemanggilan prosedur TambahEmpat: ')
  write('  a = ', a)
  write('  b = ', b)
```

Algoritma 10.18 Program utama untuk memanggil prosedur TambahEmpat.

Keluaran dari Algoritma 10.11 dan Algoritma 10.18:

```
nilai a dan b sebelum pemanggilan prosedur Tambahmpat:
    a = 15
    b = 10
nilai x dan y di akhir prosedur TambahEmpat:
    x = 19
    y = 14
nilai a dan b sesudah pemanggilan prosedur TambahEmpat:
    a = 15
    b = 10
```

Jadi, setelah pemanggilan prosedur, nilai `a` dan `b` tidak berubah.

Prosedur dengan parameter masukan/keluaran

Sekarang, ubah prosedur `TambahEmpat` dengan `x` dan `y` sebagai parameter masukan/ keluaran, seperti contoh berikut ini:

```
procedure TambahEmpat(input/ouput x,y : integer)
{ Menambahkan nilai x dan y masing-masing dengan 4. }
{ K.Awal : x dan y sudah berisi nilai. }
{ K.Akhir: nilai x dan y bertambah masing-masing 4, lalu dicetak. }

DEKLARASI
    { tidak ada }
ALGORITMA:
   x ← x + 4
   y ← y + 4;
   write('nilai x dan y di akhir prosedur Empat:')
   write('  x = ', x)
   write('  y = ', y)
```

Algoritma 10.19 Menambahkan nilai parameter masukan/keluaran dengan 4.

Program utama:

```
PROGRAM XYZ
{ Program yang memperlihatkan efek penggunaan parameter masukan }

DEKLARASI
  a, b : integer

  procedure TambahEmpat(input/output x,y : integer)
```

```
  { Menambahkan nilai x dan y masing-masing dengan 4 }
ALGORITMA:
  a←15
  b←10
  write('nilai a dan b sebelum pemanggilan prosedur TambahEmpat: ')
  write('  a = ', a)
  write('  b = ', b)

  TambahEmpat(a,b)

  write('nilai a dan b sesudah pemanggilan prosedur TambahEmpat: ')
  write('  a = ', a)
  write('  b = ', b)
```

Algoritma 10.20 Program utama untuk memanggil prosedur TambahEmpat.

Keluaran dari Algoritma 10.19 dan Algoritma 10.20:

```
nilai a dan b sebelum pemanggilan prosedur Tambahmpat:
    a = 15
    b = 10
nilai x dan y di akhir prosedur TambahEmpat:
    x = 19
    y = 14
nilai a dan b sesudah pemanggilan prosedur TambahEmpat:
    a = 19
    b = 14
```

Jelaslah, akibat penggunaan `x` dan `y` sebagai parameter masukan/keluaran, bila nilai `a` dan `b` diubah di dalam prosedur `TambahEmpat`, maka perubahan ini dibawa ke titik pemanggilan prosedur `TambahEmpat`.

Contoh 10.6. Tuliskan prosedur untuk mempertukarkan dua buah nilai disimpan di dalam peubah *A* dan *B*.

Penyelesaian

Di sini kita harus menyatakan *A* dan *B* sebagai parameter masukan/keluaran, karena nilai *A* dan *B* dimanipulasi di dalam prosedur dan hasil pertukaran disimpan dalam peubah *A* and *B* juga.

```
procedure Tukar(input/output A, B : integer)
{ Mempertukarkan nilai A dan B. }
{ K.Awal : nilai A dan B sudah terdefinisi. }
{ K.Akhir: A berisi nilai B yang lama, B berisi nilai A yang lama. }

DEKLARASI
  temp : integer    { peubah bantu }

ALGORITMA:
  temp ← A   { simpan nilai A ke dalam temp }
  A ← B      { isikan nilai B ke dalam A }
  B ← temp   { isikan nilai temp ke dalam B }
```

Algoritma 10.21 Mempertukarkan nilai A dan B.

Contoh 10.7. Buatlah prosedur untuk menambah suatu jam dengan satu detik. Misalnya,

Jam	Jam Baru Setelah Ditambah Satu Detik
14:27:31	14:27:32
15:36:59	15:35:00
10:59:59	11:00:00
23:59:59	00:00:00

Penyelesaian

Contoh ini sudah pernah kita bahas di dalam Bab 6 (Contoh 6.16). Jam yang baru (setelah penambahan 1 detik) dapat ditentukan dengan cara perhitungan (komputasi) atau dengan cara analisis kasus. Di sini kita tulis lagi algoritmanya dalam bentuk prosedur untuk masing- masing cara.

Versi 1 (komputasi):

```
procedure JamBerikutnya(input/output J : Jam)
{ Menaikkan jam J dengan satu detik. }
{ K.Awal : J sudah berisi nilai jam (hh:mm:ss). }
{ K.Akhir: J berisi jam yang baru etelah ditambah 1 detik. }

DEKLARASI
    TotalDetik, SisaDetik : integer

ALGORITMA:
  { konversi J ke total detik }
  TotalDetik ← J.hh*3600 + J.mm*60 + J.ss

  { tambahkan totaldetik dengan 1 }
  TotalDetik ← TotalDetik+1

  { konversi totaldetik ke hh:mm:ss }
  J.hh ← TotalDetik div 3600
  SisaDetik ← TotalDetik mod 3600
  J.mm ← SisaDetik div 60
  J.ss ← SisaDetik mod 60
```

Algoritma 10.22 Menambah jam sekarang dengan 1 detik (versi 1).

Versi 2 (analisis kasus):

```
procedure JamBerikutnya(input/output J : Jam)
{ Menaikkan jam J dengan satu detik. }
{ K.Awal : J sudah berisi nilai jam (hh:mm:ss). }
{ K.Akhir: J berisi jam yang baru etelah ditambah 1 detik. }

DEKLARASI
    { tidak ada }

ALGORITMA:
  if J.ss + 1 < 60 then  { tidak ada masalah penambahan 1 detik. }
```

```
    J.ss ← J.ss + 1
  else { J.ss + 1 = 60 }
    J.ss←0     { detik kembali menjadi nol, menit bertambah 1, tapi
                 periksa dulu apakah menit + 1 < 60 }
    if J.mm + 1 < 60 then { tidak ada masalah penambahan 1 menit }
      J.mm ← J.mm + 1
    else { J.mm + 1 = 60 }
      J.mm←0  { menit menjadi nol, jam bertambah 1, tapi periksa dulu
                apakah jam + 1 < 24 }
      if J.hh + 1 < 24 then  { tidak ada masalah penambahan 1 jam }
        J.hh ← J.hh+1
      else { J.hh + 1 = 24 }
        J.hh ← 0
      endif
    endif
  endif
```

Algoritma 10.23 Menambah jam sekarang dengan 1 detik (versi 2).

Misalkan program utamanya adalah mencetak perjalanan waktu yang terus bertambah detik demi detik, dimulai dari jam yang dibaca sebagai masukan, sampai mencapai pukul 0:0:0.

```
PROGRAM JamHidup1
{ Membuat jam "hidup" yang selalu bertambah 1 detik terus menerus
sampai jam 00:00:00. Masukan jam dibaca dari piranti masukan. Setiap
pertambahan 1 detik, jam yang baru ditampilkan sebagai keluaran }

DEKLARASI
  type Jam : record<hh:integer, { 0 ≤ hh ≤ 59 }
                    mm:integer, { 0 ≤ hh ≤ 59 }
                    ss:integer  { 0 ≤ hh ≤ 23 }
                   >
  J : Jam
  procedure JamBerikutnya(input/output J : Jam)
  { Menaikkan jam J dengan satu detik. }

ALGORITMA:
  read(J.hh, J.mm, J.ss)       { baca jam awal }
  repeat
    write(J.hh, J.mm, J.ss)    { cetak jam sekarang }
    JamBerikutnya(J)
  until (J.hh = 0) and (J.mm = 0) and (J.ss = 0)
  write(J.hh, J.mm, J.ss)      { cetak jam 0:0:0 }
```

Algoritma 10.24 Program untuk mencetak jam sampai pukul 0:0:0.

Jika kita ingin membuat jam yang tidak pernah berhenti perjalanan waktunya, maka konstruksi kalangnya harus diubah menjadi *loop forever* (yaitu pengulangan yang tidak pernah berhenti). *Loop forever* dapat dipecahkan dengan paksa (misalnya menggunakan instruksi `break` dan *return* dalam bahasa *C*).

```
PROGRAM JamHidup2
{ Membuat jam "hidup" yang selalu bertambah 1 detik terus menerus
sampai jam 00:00:00. Masukan jam dibaca dari piranti masukan. Setiap
pertambahan 1 detik, jam yang baru ditampilkan sebagai keluaran }

DEKLARASI
  type Jam : record<hh:integer, { 0 ≤ hh ≤ 59 }
                    mm:integer, { 0 ≤ hh ≤ 59 }
                    ss:integer  { 0 ≤ hh ≤ 23 }
                   >
  J : Jam
  procedure JamBerikutnya(input/output J : Jam)
  { Menaikkan jam J dengan satu detik. }

ALGORITMA:
  read(J.hh,J.mm,J.ss)    { baca jam awal }
  while true do
    write(J.hh,J.mm,J.ss)  { cetak jam awal }
    JamBerikutnya(J)
  endwhile
```

Algoritma 10.25 Program untuk mencetak jam yang terus hidup detik demi detik.

10.4.4 Program dengan Prosedur atau Tanpa Prosedur?

Sangat dianjurkan menulis program yang modular. Program yang dipecah menjadi beberapa prosedur (atau fungsi) menunjukkan teknik pemrograman yang baik dan terstruktur.

10.4.5 Prosedur dengan Parameter atau Tanpa Parameter?

Sangat dianjurkan menulis prosedur dengan parameter. Parameter berfungsi sebagai media komunikasi antara modul dengan program pemanggil. Selain itu, parameter dapat mengurangi kebutuhan penggunaan peubah global.

10.4.6 Parameter Masukan atau Parameter Keluaran?

Bergantung pada masalahnya, apakah prosedur tersebut menghasilkan keluaran yang digunakan oleh program pemanggil atau tidak. Bila prosedur menghasilkan keluaran yang digunakan oleh program pemanggil, gunakan parameter keluaran untuk menampung keluaran tersebut. Sebaliknya, bila prosedur tidak menghasilkan keluaran, atau kalaupun menghasilkan keluaran

dan ternyata keluaran tersebut hanya digunakan di dalam prosedur itu saja, gunakan parameter masukan. Bila prosedur menerima masukan sekaligus keluaran pada parameter yang sama, gunakan parameter masukan/keluaran.

10.5 Translasi Notasi Algoritmik Prosedur ke dalam Notasi Bahasa Pascal dan C

Beberapa aturan yang harus diketahui dalam proses translasi:

1. Dalam bahasa *Pascal*, pendefinisian prosedur ditulis bersatu di dalam program utama, kecuali jika direalisasikan sebagai *unit*. Prosedur diletakkan di bawah kata *var*.
2. Dalam bahasa *C*, pendefinisian prosedur ditulis di luar blok program utama (*main*), kecuali jika direalisasikan sebagai *file include*. Prosedur umumnya diletakkan sesudah blok `main()`, sedangkan deklarasi prosedur ditulis sebelum blok `main()` sebagai purwarupa saja (purwarupa prosedur hanya berupa *header* dari prosedur tersebut). Jika pendefinisian prosedur ditulis sebelum blok `main()`, maka pendeklarasian purwarupa prosedur tidak diperlukan lagi.
3. Dalam bahasa *C*, tidak dikenal istilah prosedur. Semua modul program adalah fungsi. Prosedur adalah fungsi yang tidak mengembalikan nilai apa pun. Karena itu, *return value* untuk prosedur adalah `void`, (yang artinya 'kosong')
4. Dalam bahasa *C*, bila prosedur tidak mempunyai parameter, maka tanda kurung '(' dan ')' tetap ditulis setelah nama prosedur tersebut.
5. Semua parameter dalam bahasa *C* adalah parameter masukan. Oleh karena itu, semua argumen parameter aktual dilewatkan sebagai "*by value*". Ini artinya nilai parameter aktual disalin ke parameter formal yang bersesuaian. Suatu cara untuk memperoleh efek parameter keluaran maupun efek parameter masukan/keluaran adalah melewatkan *pointer* terhadap parameter aktual dengan menambahkan karakter "&" di awal nama parameter aktual yang berjenis parameter masukan atau masukan/keluaran. Tanda "&" menyatakan alamat dari peubah. Sedangkan parameter formal yang berjenis masukan atau masukan/keluaran ditambahkan karakter "*" di depannya. Karakter "*" menyatakan operator *indirect*, yang berarti sebuah peubah yang diawali dengan operator ini diakses secara tidak langsung melalui alamatnya yang disimpan di dalam peubah lain (pada contoh prosedur, alamat disimpan oleh parameter aktual diawali oleh karakter "&"). Lihat contoh translasi prosedur yang mengandung parameter keluaran atau masukan /keluaran.
6. Bahasa Pascal yang memungkinkan prosedur mempunyai parameter masukan, parameter keluaran, dan parameter masukan/keluaran. Parameter formal yang bertipe keluaran atau masukan/keluaran harus diawali dengan kata kunci `var`, sedangkan parameter formal yang

bertipe masukan tidak diawali dengan kata kunci `var`. Argumen parameter aktual dilewatkan ke parameter formal yang bertipe masukan sebagai "*by value*", sedangkan bila parameter formalnya bertipe masukan atau masukan/keluaran, maka argumen parameter aktual dilewatkan sebagai "*by reference*".

ALGORITMIK:

```
procedure NamaProsedur (deklarasi parameter, jika ada)
{ Spesifikasi prosedur, berisi penjelasan tentang apa yang dilakukan
oleh prosedur ini.
K.Awal : keadaan sebelum prosedur dilaksanakan.
K.Akhir: keadaan setelah prosedur dilaksanakan. }

DEKLARASI
  { semua nama yang dipakai di dalam prosedur dan hanya berlaku lokal
di dalam prosedur didefinisikan di sini }

ALGORITMA:
   { badan prosedur, berisi urutan instruksi }
```

Cara memanggil prosedur:

NamaProsedur

jika prosedur tidak mempunyai parameter, atau

NamaProsedur(parameter aktual)

jika prosedur mengandung parameter.

PASCAL:

```
procedure NamaProsedur(deklarasi parameter, jika ada) ;
{ Spesifikasi prosedur, berisi penjelasan tentang apa yang dilakukan
oleh prosedur ini. }
{ K.Awal : keadaan sebelum prosedur dilaksanakan. }
{ K.Akhir: keadaan setelah prosedur dilaksanakan. }

(* DEKLARASI *)
  { semua nama yang dipakai dalam prosedur dan hanya berlaku lokal di
    dalam prosedur didefinisikan di sini }

(* ALGORITMA: *)
begin
   { badan prosedur, berisi kumpulan instruksi }
end;
```

Cara memanggil prosedur:

NamaProsedur;

jika prosedur tidak mempunyai parameter, atau

NamaProsedur (parameter aktual);

jika prosedur mengandung parameter.

C:

```
void NamaProsedur(deklarasi parameter, jika ada)
/* Spesifikasi prosedur, berisi penjelasan tentang apa yang dilakukan
   oleh prosedur ini. */
/* K.Awal : keadaan sebelum prosedur dilaksanakan. */
/* K.Akhir: keadaan setelah prosedur dilaksanakan. */

{
  /* DEKLARASI */
  /* semua nama yang dipakai dalam prosedur dan hanya berlaku lokal di
     dalam prosedur didefinisikan di sini */

  /* ALGORITMA: */
  /* badan prosedur, berisi kumpulan instruksi */
}
```

Cara memanggil prosedur:

NamaProsedur();

jika prosedur tidak mempunyai parameter, atau

NamaProsedur(parameter aktual);

jika prosedur mengandung parameter.

1. Contoh translasi prosedur tanpa parameter

ALGORITMIK:

Prosedur:

```
procedure HitungLuasSegitiga
{ Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }
{ K.Awal : sembarang }
{ K.Akhir: luas segitiga tercetak.}

DEKLARASI
  alas    : real   { panjang alas segitiga, dalam cm }
  tinggi  : real   { tinggi segitiga, dalam cm }
  luas    : real   { luas segitiga, dalam cm² }

ALGORITMA:
  read(alas,tinggi)
  luas ← (alas * tinggi)/2
  write(luas)
```

Program utama:

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   procedure HitungLuasSegitiga
  { Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }

ALGORITMA:
   read(N)  { tentukan banyaknya segitiga }
   for i ← 1 to N do
     HitungLuasSegitiga
   endfor
```

PASCAL:

```
PROGRAM Segitiga;
{ Menghitung luas N buah segitiga. }

(* DEKLARASI *)
var
   i, N : integer;

  procedure HitungLuasSegitiga;
  { Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 }
  { K.Awal : sembarang }
  { K.Akhir: luas segitiga tercetak.}

  (* DEKLARASI *)
  var
    alas    : real;   { panjang alas segitiga, dalam cm }
    tinggi  : real;   { tinggi segitiga, dalam cm }
    luas    : real;   { luas segitiga, dalam cm² }

  (* ALGORITMA: *)
  begin
    write('Panjang alas segitiga? '); readln(alas);
    write('Tinggi segitiga?' '); readln(tinggi);
    luas := (alas*tinggi)/2;
    write('Luas segitiga = ',luas);
  end;


(* PROGRAM UTAMA: *)
begin
   write('Banyaknya segitiga? '); readln(N)
   for i := 1 to N do
     HitungLuasSegitiga;
   {endfor}
end.
```

C:

```
/* PROGRAM Segitiga */
/* Menghitung luas N buah segitiga. */

#include <stdio.h>

/* Deklarasi purwarupa prosedur yang digunakan */
void HitungLuasSegitiga();

main()
{
   /* DEKLARASI */)
   int i, N;

  /* PROGRAM UTAMA: */
   printf("Banyaknya segitiga? "); scanf(&N);
   for (i = 1; i<=N; i++)
     HitungLuasSegitiga();
   /*endfor*/
}
void HitungLuasSegitiga()
/* Menghitung luas segitiga dengan rumus L = (alas x tinggi)/2 */
/* K.Awal : sembarang */
/* K.Akhir: luas segitiga tercetak. */
{
  /* DEKLARASI */
  float alas;      /* panjang alas segitiga, dalam cm */
  float tinggi;    /* tinggi segitiga, dalam cm */
  float luas;      /* luas segitiga, dalam cm² */

  /* ALGORITMA: */
  printf("Panjang alas segitiga? "); scanf(&alas);
  printf("Tinggi segitiga?` "); scanf(&tinggi);
  luas=(alas*tinggi)/2.0;
  printf("Luas segitiga = %f ",luas);
}
```

2. Contoh translasi prosedur dengan parameter masukan

ALGORITMIK:

Prosedur:

```
procedure HitungLuasSegitiga(input alas, tinggi : real)
{ Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 }
{ K.Awal : alas dan tinggi sudah terdefinisi nilainya }
{ K.Akhir: luas segitiga tercetak.}

DEKLARASI
  luas    : real   { luas segitiga, dalam cm² }

ALGORITMA:
  luas ← (alas * tinggi)/2
  write(luas)
```

Program utama:

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   a, t : real

   procedure HitungLuasSegitiga(input alas, tinggi : real)
  { Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 }

ALGORITMA:
   read(N)  { tentukan banyaknya segitiga }
   for i ← 1 to N do
     read(a,t)
     HitungLuasSegitiga(a,t)
   endfor
```

PASCAL:

```
PROGRAM Segitiga;
{ Menghitung luas N buah segitiga. }

(* DEKLARASI *)
var
   i, N : integer;
   a, t : real;

  procedure HitungLuasSegitiga(alas, tinggi : real);
  { Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 }
  { K.Awal : alas dan tinggi sudah terdefinisi nilainya }
  { K.Akhir: luas segitiga tercetak.}

  (* DEKLARASI *)
  var
    luas     : real;   { luas segitiga, dalam cm² }

  (* ALGORITMA: *)
  begin
    luas := (alas * tinggi)/2;
    write('Luas segitiga = ',luas);
  end;

(* PROGRAM UTAMA: *)
begin
   write('Banyaknya segitiga? '); readln(N)
   for i:=1 to N do
    begin
      write('Panjang alas segitiga? '); readln(a);
      write('Tinggi segitiga?' '); readln(t);
      HitungLuasSegitiga(a, t);
    end;{for}
end.
```

C:

```c
/* PROGRAM Segitiga */
/* Menghitung luas N buah segitiga. */

#include <stdio.h>

/* Deklarasi purwarupa prosedur yang digunakan */
void HitungLuasSegitiga(float a, float t);

main()
{
   /* DEKLARASI */
   int i, N;
   float a, t;

  /* PROGRAM UTAMA: */
   printf("Banyaknya segitiga? "); scanf(&N);
   for (i = 1; i<=N; i++)
   {
     printf("Panjang alas segitiga? "); scanf(&a);
     printf("Tinggi segitiga?' "); scanf(&t);
     HitungLuasSegitiga(a, t);
   }
}

void HitungLuasSegitiga(float alas, float tinggi)
/* Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 */
/* K.Awal : alas dan tinggi sudah terdefinisi nilainya */
/* K.Akhir: luas segitiga tercetak. */

{  /* DEKLARASI */
  float luas;      /* luas segitiga, dalam cm² */

  /* ALGORITMA: */
  luas = (alas*tinggi)/2.0;
  printf("Luas segitiga = %f ",luas);
}
```

3. Contoh translasi prosedur dengan parameter keluaran

ALGORITMIK:

Prosedur:

```
procedure HitungLuasSegitiga(input alas, tinggi : real,
                             output luas : real)
{ Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 }
{ K.Awal : alas dan tinggi sudah terdefinisi nilainya }
{ K.Akhir: luas berisi luas segitiga.}

DEKLARASI
  { tidak ada }

ALGORITMA:
  luas ← (alas * tinggi)/2
```

Program utama:

```
PROGRAM Segitiga
{ Menghitung luas N buah segitiga. }

DEKLARASI
   i, N : integer
   a, t : real
   L    : real    { luas segitiga }
   procedure HitungLuasSegitiga(input alas, tinggi : real)
  { Menghitung luas segitiga dengan rumus Luas = (alas × tinggi)/2 }

ALGORITMA:
   read(N)  { tentukan banyaknya segitiga }
   for i ← 1 to N do
     read(a, t)
     HitungLuasSegitiga(a,t,L)
     write(L)
   endfor
```

PASCAL:

```
PROGRAM Segitiga;
{ Menghitung luas N buah segitiga. }

(* DEKLARASI *)
var
   i, N : integer;
   a, t : real;
   L    : real;    { luas segitiga }

  procedure HitungLuasSegitiga(alas, tinggi : real; var luas : real);
  { Menghitung luas segitiga dengan rumus Luas = (alas × tinggi)/2 }
  { K.Awal : alas dan tinggi sudah terdefinisi nilainya }
  { K.Akhir: luas segitiga tercetak.}

  (* DEKLARASI *)
      { tidak ada }

  (* ALGORITMA: *)
  begin
    luas := (alas * tinggi)/2;
  end;

(* PROGRAM UTAMA: *)
begin
   write('Banyaknya segitiga? '); readln(N)
   for i:=1 to N do
    begin
      write('Panjang alas segitiga? '); readln(a);
      write('Tinggi segitiga?' '); readln(t);
      HitungLuasSegitiga(a, t, L);
      write('Luas segitiga = ',L);
    end;{for}
end.
```

Perhatikan bahwa `luas` adalah parameter keluaran, sehingga di dalam *header* prosedur luas diawali dengan kata `var`.

C:

```
/* PROGRAM Segitiga */
/* Menghitung luas N buah segitiga. */

#include <stdio.h>

/* Deklarasi purwarupa prosedur yang digunakan */
void HitungLuasSegitiga(float a, float t, float *L);

main()
{
   /* DEKLARASI */)
   int i, N;
   float a, t;
   float L;      /* luas segitiga, dalam cm² */

   /* PROGRAM UTAMA: */
   printf("Banyaknya segitiga? "); scanf(&N);
   for (i = 1; i<=N; i++)
   {
     printf("Panjang alas segitiga? "); scanf(&a);
     printf("Tinggi segitiga?` "); scanf(&t);
     HitungLuasSegitiga(a, t, &L);
     printf("Luas segitiga = %f ",L);
   }

}



void HitungLuasSegitiga(float alas, float tinggi, float *luas)
/* Menghitung luas segitiga dengan rumus Luas = (alas x tinggi)/2 */
/* K.Awal : alas dan tinggi sudah terdefinisi nilainya */
/* K.Akhir: luas segitiga tercetak. */

{
  /* DEKLARASI */

  /* ALGORITMA: */
  *luas = (alas * tinggi) / 2.0;

}
```

Perhatikan bahwa `luas` adalah parameter keluaran, sehingga di dalam *header* prosedur luas diawali dengan karakter * menjadi `*luas`. Semua pernyataan yang menggunakan peubah `luas` juga harus dinyatakan dengan `*luas`:

```
*luas = (alas * tinggi) / 2.0;
```

Ketika prosedur dipanggil, maka parameter aktualnya diawali dengan operator alamat &:

```
HitungLuasSegitiga(a, t, &L);
```

4. Contoh translasi prosedur dengan parameter masukan/keluaran

ALGORITMIK:

Prosedur:

```
procedure Tukar(input/output A, B : integer)
{ Mempertukarkan nilai A dan B. }
{ K.Awal : nilai A dan B sudah terdefinisi. }
{ K.Akhir: A berisi nilai B, B berisi nilai A semula. }

DEKLARASI
  temp : integer    { peubah bantu }

ALGORITMA:
  temp ← A  { simpan nilai A ke dalam temp }
  A ← B     { isikan nilai B ke dalam A }
  B ← temp  { isikan nilai temp ke dalam B }
```

Program Utama:

```
PROGRAM SelisihXY
{ Program untuk menghitung selisih nilai X dan Y, dengan syarat X ≥ Y.
Jika X < Y, maka X dan Y dipertukarkan dengan memanggil prosedur Tukar.}

DEKLARASI
  X, Y, Z : integer

  procedure Tukar(input/output A, B : integer)
  { Mempertukarkan nilai A dan B. }

ALGORITMA:
  read(X,Y)    { baca nilai X dan Y terlebih dahulu }

  { jika X < Y, pertukarkan nilai X dan Y dengan memanggil
    prosedur Tukar }
  if X < Y then
     Tukar(X,Y)
  endif

  Z ← X - Y  { hitung selisih X dan Y }
  write(Z)
```

PASCAL:

```
program SelisihXY;
{ Program untuk menghitung selisih nilai X dan Y, dengan syarat X ≥ Y.
Jika X < Y, maka X dan Y dipertukarkan dengan memanggil prosedur
Tukar.}

(* DEKLARASI *)
var
  X, Y, Z : integer;

 procedure Tukar(var A, B : integer);
 { Mempertukarkan nilai A dan B. }
 { K.Awal : Nilai A dan B sudah terdefinisi. }
 { K.Akhir: A berisi nilai B, B berisi nilai A semula. }

 (* DEKLARASI *)
 var
    temp : integer;    { peubah bantu }

(* ALGORITMA: *)
 begin
    temp := A;      { simpan nilai A di dalam temp }
    A := B;         { isi A dengan nilai B }
    B := temp;      { isi B dengan nilai temp }
 end;


(* PROGRAM UTAMA *)
begin
 { baca nilai X dan Y terlebih dahulu }
  write('X = ?'); readln(X);
  write('Y = ?'); readln(Y);

  { jika X < Y, pertukarkan nilai X dan Y dengan memanggil
    prosedur Tukar }
  if X < Y then
     Tukar(X,Y)
  {endif}

  Z := X - Y;  { hitung selisih X dan Y }
  write('Z = ',Z);

end.
```

Perhatikan bahwa A dan B adalah parameter masukan/keluaran, sehingga di dalam *header* prosedur A dan B diawali dengan kata var.

C:

```
/* PROGRAM SelisihXY */
/* Program utama untuk menghitung selisih nilai X dan Y, dengan syarat
X ≥ Y. Jika X < Y, maka X dan Y dipertukarkan dengan memanggil
prosedur Tukar. */
```

```
#include <stdio.h>

void Tukar(int *X, int *Y)
/* Mempertukarkan nilai X dan Y */

main()
{
  /* DEKLARASI */
  int X, Y, Z;

  /*PROGRAM UTAMA */
  printf("X = ?"); scanf("%d",&X);
  printf("Y = ?"); scanf("%d",&Y);

  /* jika X < Y, pertukarkan nilai X dan Y dengan memanggil
     prosedur Tukar */
  if (X < Y)
    Tukar(&X,&Y);
  /*endif*/

  Z = X - Y;  /* hitung selisih X dan Y */
  printf("Z = %d \n", Z);
}

void Tukar(int *A, int *B)
/* Mempertukarkan nilai B dan B */
/* K.Awal : nilai A dan B sudah terdefinisi. */
/* K.Akhir: A berisi nilai B, B berisi nilai A semula. */
{
  /* DEKLARASI */
  int  temp;   /* peubah bantu */

  /* ALGORITMA: */
  temp = *A;      /* simpan nilai A di dalam temp */
  *A = *B;        /* isi A dengan nilai B */
  *B = temp;      /* isi B dengan nilai temp */
}
```

Perhatikan bahwa A dan B adalah parameter masukan/keluaran, sehingga di dalam *header* prosedur A dan B diawali dengan karakter * menjadi *A dan *B. Semua pernyataan yang menggunakan peubah A dan B juga harus dinyatakan dengan *A dan *B:

```
temp = *A;      /* simpan nilai A di dalam temp */
*A = *B;        /* isi A dengan nilai B */
*B = temp;      /* isi B dengan nilai temp */
```

Ketika prosedur dipanggil, maka parameter aktualnya diawali dengan operator alamat &:

```
Tukar(&X,&Y);
```

5. Contoh translasi lainnya

ALGORITMIK:

Prosedur:

```
procedure HitungRataRata(input N : integer, output u : real)
{ Menghitung rata-rata N buah data bilangan riil. }
{ K.Awal : N sudah berisi banyaknya data (N > 0). }
{ K.Akhir: u berisi rata-rata seluruh bilangan. }

DEKLARASI
   x : real          { data yang dibaca }
   i : integer       { pencacah banyaknya data }
   jumlah : real     { jumlah seluruh data }

ALGORITMA:
   jumlah ← 0    { inisialisasi }
   for i ← 1 to N do
     read(x)
     jumlah ← jumlah + x
   endfor

   u ← jumlah/N     { nilai rata-rata }
```

Program Utama:

```
PROGRAM RerataData
{ Program untuk menghitung nilai rata-rata N orang mahasiswa. }

DEKLARASI
  N : integer          { jumlah peserta ujian }
  rerata : real        { nilai rata-rata seluruh peserta }
  procedure HitungRataRata(input N : integer, output u : real)
  { Menghitung rata-rata N buah data bilangan riil. }

ALGORITMA:
   read(N)     { tentukan jumlah peserta ujian }
   HitungRataRata(N,rerata)
   write(rerata)
   if rerata < 50 then
     write('Nilai rata-rata ujian mahasiswa tidak bagus')
   else
     write('Nilai rata-rata ujian mahasiswa bagus')
   endif
```

PASCAL:

```
program RerataData;
{ Program untuk menghitung nilai rata-rata N orang mahasiswa. }
```

```
(* DEKLARASI *)
var
  N : integer;        { jumlah peserta ujian }
  rerata : real;      { nilai rata-rata seluruh peserta }

  procedure HitungRataRata(N : integer; var u : real);
  { Menghitung rata-rata N buah data bilangan riil. }
  { K.Awal : N sudah berisi banyaknya data (N > 0) }
  { K.Akhir: u berisi rata-rata seluruh bilangan. }

  (* DEKLARASI *)
   var
     x : real;              { data yang dibaca }
     i : integer;           { pencacah banyaknya data }
     jumlah : real;         { jumlah seluruh data }

  (* ALGORITMA: *)
   begin
     jumlah := 0;          { inisialisasi }
     for i:=1 to N do
      begin
         write('x : ?'); readln(x);
         jumlah:=jumlah+x;
      end;{for}
      u := jumlah/N;
   end;

(* PROGRAM UTAMA *)
begin
   write('Jumlah peserta ujian? '); readln(N);
   HitungRataRata(N,rerata);
   writeln('Nilai rata-rata = ', u);
   if rerata < 50 then
     writeln('Nilai rata-rata ujian mahasiswa tidak bagus')
   else
      writeln('Nilai rata-rata ujian mahasiswa bagus')
   {endif}
end.
```

C:

```
/* PROGRAM RerataData */
/* Program untuk menghitung nilai rata-rata N orang mahasiswa. */

#include <stdio.h>

void HitungRataRata(int N, float *u);
/* Menghitung rata-rata N buah data bilangan riil. */

main()
{
  /* DEKLARASI */
  float rerata;       /* nilai rata-rata seluruh data */
```

```
  int N;              /* jumlah peserta ujian, N > 0 */

  /* PROGRAM UTAMA */
  printf("Jumlah peserta ujian? "); scanf("%d",&N);
  HitungRataRata(N,&rerata);
  printf("Nilai rata-rata = %f ", rerata);
  if (rerata < 50)
     printf("Nilai rata-rata ujian mahasiswa tidak bagus \n")
  else
     printf("Nilai rata-rata ujian mahasiswa bagus \n")
  /*endif*/
}

void HitungRataRata(int N, float *u)
/* Menghitung rata-rata N buah data bilangan riil. */
/* K.Awal : N sudah berisi banyaknya data (N > 0). */
/* K.Akhir: u berisi rata-rata seluruh bilangan.. */

   /* DEKLARASI  */
   float x;        /* data yang dibaca */
   int i;          /* pencacah banyaknya data */
   float jumlah;   /* jumlah seluruh data */

   /* ALGORITMA: */
   jumlah = 0;             /* inisialisasi penjumlah */
   for (i=1; i<=N; i++)
   {
     printf("x : ?"); scanf("%f",&x);
     jumlah = jumlah+x;
   }
   *u = jumlah/N;
}
```

Soal Latihan Bab 10

1. Tulislah prosedur untuk menghitung jumlah *N* buah bilangan genap pertama (bilangan genap dimulai dari 0). Prosedur menerima (parameter) masukan *N* dan memberikan (parameter) keluaran jumlah *N* buah bilangan genap pertama.

2. Tulislah prosedur yang menghasilkan nilai rata-rata sekumpulan data bilangan bulat yang dibaca secara berulang-ulang dari papan ketik (akhir pembacaan adalah 9999). Prosedur memiliki parameter keluaran, yaitu nilai rata-rata yang dihasilkan.

3. Ulangi soal nomor 2 tetapi prosedur menghasilkan nilai terkecil.

4. Misalkan Anda menyimpan uang di bank konvensional sejumlah A rupiah pada awal tahun. Jika Anda mendapat bunga tahunan sebesar i persen, maka jumlah uang Anda setelah n tahun adalah

   $$F = A\{(1 + i/100) + (1 + i/100)^2 + (1 + i/100)^3 + ... + (1 + i/100)^n]$$

   Buatlah prosedur yang menerima masukan A, i, n dan memberikan keluaran F.

5. Tulislah prosedur dalam bahasa *Pascal* dan bahasa *C* yang menerima masukan berupa nilai *integer* positif dan menampilkan nilai tersebut dalam kata-kata.

   Contoh:

   ```
   masukan: 15
   keluaran: lima belas
   masukan: 2347
   keluaran: dua ribu tiga ratus empat puluh tujuh
   ```

6. Tuliskan prosedur yang menerima masukan sebuah jam (*hh*:*mm*:ss) dan p menit kemudian memberikan keluaran jam yang lalu setelah jam sekarang dikurangi p menit (latihan ini merupakan kebalikan dari Contoh Masalah 8.1 di dalam Bab 8). Buatlah dua versi algoritmanya, yang pertama menggunakan cara perhitungan, yang kedua menggunakan cara analisis kasus.

7. Tulislah prosedur yang menerima nama hari sekarang dan menentukan nama hari besok. Misalnya, jika hari sekarang "rabu" maka hari besok adalah "kamis".

8. Ulangi nomor 5, tetapi menentukan nama hari sebelumnya.

9. Tuliskan prosedur yang menerima sebuah tanggal dalam bentuk *dd-mm-yyyy* (contoh: 12-8-1996) dan memberikan keluaran tanggal sebelumnya (latihan ini merupakan kebalikan dari Contoh Masalah 8.7 di dalam Bab 8). *Catatan*: parameter tanggal berjenis masukan/keluaran.

10. Tuliskan prosedur yang menerima sebuah tanggal (*dd-mm-yyyy*) lalu menghitung berapa hari jarak tanggal tersebut dari tanggal 1-1-1900.

11. Tulislah prosedur yang menerima jam sekarang (*hh*:*mm*:*ss*), tanggal (*dd-mm-yyyy*), dan nama hari, kemudian jam terus berputar detik demi detik sehingga ketika mencapai pukul 0:0:0, tanggal berikutnya juga berubah, begitu pula nama hari berikutnya. Perhatikan kasus tahun kabisat.

12. Buatlah prosedur dalam bahasa *Pascal* dan bahasa *C* untuk mencetak piramid dengan "ketinggian" n (asumsi: $n \leq 10$) Sebagai contoh, jika $n = 6$, maka piramid yang dihasilkan adalah:

```
     1
    232
   34543
  4567654
 567898765
67890109876
```

13. Buatlah prosedur dalam bahasa *Pascal* dan bahasa *C* untuk mencetak gambar seperti di bawah ini jika diberikan nilai N (yaitu lebar baris terpanjang), dengan asumsi N adalah ganjil.

Contoh: $N = 7$

11
Fungsi

Seperti halnya prosedur, fungsi juga merupakan upa-program yang mempunyai tujuan spesifik. Di dalam Bab 11 ini akan dijelaskan pengertian fungsi, cara mendefinisikan fungsi, cara pemanggilan fungsi, dan translasi fungsi ke dalam bahasa *Pascal* dan bahasa *C*. Pada akhir bab juga dijelaskan konversi prosedur menjadi fungsi dan sebaliknya konversi fungsi menjadi prosedur. Baik fungsi maupun prosedur keduanya merupakan upa-program yang ekivalen, namun pada beberapa masalah adalanya kita lebih tepat menggunakan fungsi ketimbang prosedur, demikian juga sebaliknya.

11.1 Definisi Fungsi

Fungsi adalah upa-program yang memberikan/mengembalikan (*return*) sebuah nilai dari tipe tertentu (tipe dasar atau tipe bentukan). Definisi fungsi di dalam program bersesuaian dengan definisi fungsi di dalam matematika. Di dalam matematika, kita mengenal cara penulisan fungsi seperti pada contoh berikut:

1. $f(x) = 2x^2 + 5x - 8$
2. $H(x, y) = 3x - y + xy$

Pada kedua contoh di atas, f dan H adalah nama fungsi, sedangkan x dan y adalah *parameter* fungsi yang bersangkutan. Nilai yang diberikan oleh fungsi bergantung pada masukan parameter. Sebagai misal:

1. $x = 2$, maka $f(2) = 2 \cdot 2^2 + 5 \cdot 2 - 8 = 10$
2. $x = 1, y = 2$, maka $H(1, 2) = 3 \cdot 1 - 2 + 1 \cdot 2 = 3$

Nilai 10 dan 3 pada kedua contoh di atas adalah nilai yang diberikan (*return value*) oleh masing-masing fungsi *f* dan fungsi *H*.

Sebagaimana halnya dengan prosedur, fungsi diakses dengan memanggil namanya. Seperti pada prosedur, fungsi juga dapat mengandung daftar parameter formal. Parameter pada fungsi selalu merupakan parameter masukan. Dengan kata lain, tidak ada fungsi yang mempunyai parameter keluaran atau parameter masukan/keluaran. Parameter masukan pada fungsi berrati bahwa bahwa parameter tersebut merupakan masukan yang digunakan oleh fungsi tersebut untuk menghasilkan nilai.

11.2 Pendefinisian Fungsi

Sebagaimana halnya pada prosedur, struktur fungsi sama dengan struktur algoritma yang sudah Anda kenal: ada bagian *header* yang berisi nama fungsi (beserta parameter masukan, jika ada) dan spesifikasi tentang fungsi tersebut, bagian deklrasi, dan badan fungsi.

Notasi algoritmik untuk mendefinisikan fungsi adalah:

```
function NamaFungsi( input  deklrasi parameter, jika ada)→tipe
{  spesifikasi fungsi, menjelaskan apa yang dilakukan dan yang
   dikembalikan oleh fungsi. }

DEKLARASI
{ semua nama yang dipakai di dalam fungsi dan hanya berlaku lokal di
dalam prosedur didefinisikan di sini }

ALGORITMA:
  { badan fungsi, berisi instruksi-instruksi untuk menghasilkan nilai
    yang akan dikembalikan oleh fungsi }

  return ekspresi  { pengembalian nilai yang dihasilkan fungsi }
```

Algoritma 11.1 Struktur fungsi

Tipe menspesifikasikan tipe nilai yang diberikan oleh fungsi. Nilai yang diberikan oleh fungsi dapat bertipe dasar maupun bertipe bentukan. Parameter formal selalu berjenis parameter masukan sehingga deklarasi nama parameter selalu diawali dengan kata input. Sebagaimana halnya pada prosedur, parameter fungsi pada fungsi tidak selalu harus ada.

Semua nama peubah/konstanta yang hanya berlaku di dalam fungsi saja diumumkan di bagian deklarasi. Nama yang didefinisikan di dalam bagian deklarasi fungsi hanya dikenal dan berlaku di dalam fungsi yang bersangkutan saja, fungsi lain atau program utama tidak dapat menggunakannya.

Pernyataan

```
return ekspresi
```

di dalam badan fungsi bertujuan untuk mengembalikan nilai yang dihasilkan oleh fungsi tersebut. *Ekspresi* dapat berupa konstanta, atau sebuah peubah, atau sebuah rumus. Di bawah ini diberikan beberapa contoh pendefinisian fungsi.

Contoh 11.1. Tulislah fungsi untuk menghasilkan nilai $F(x) = 2x^2 + 5x - 8, x \in \mathbf{R}$

Penyelesaian

```
function F(input x : real) →real
{ mengembalikan nilai F(x) = 2x² + 5x - 8, x ∈ R }

DEKLARASI
  { tidak ada }

ALGORITMA:
   return 2*x*x + 5*x - 8
```

Algoritma 11.2 Pendefinisian fungsi $F(x) = 2x^2 + 5x - 8$.

Di dalam Algoritma 11.2 di atas,
F adalah nama fungsi, tipe-nya *real*
x adalah parameter (*by value*) formal

dan di dalam badan fungsi, nilai yang dihasilkan oleh fungsi dikembalikan (*return*) ke titik pemanggilan dengan pernyataan:

```
return 2*x*x + 5*x - 8
```

yang dalam hal ini, `return` mengembalikan hasil evaluasi dari ekspresi

```
2*x*x + 5*x - 8
```

Contoh 11.2. Tulislah fungsi untuk menghitung $H(u,v,w) = 2uv^2 + 3vw + 10v$ dengan u, v, dan w bertipe bilangan bulat.

Penyelesaian

```
function H(input u,v,w : integer)→integer
{ mengembalikan nilai H(u,v,w)= 2uv² + 3vw + 10v }

DEKLARASI
    { tidak ada }

ALGORITMA:
    return 2*u*v*v + 3*v*w + 10*v
```

Algoritma 11.3 Pendefinisian fungsi $H(u,v,w) = 2uv^2 + 3vw + 10v$.

Contoh 11.3. Buatlah fungsi untuk menentukan apakah sebuah bilangan bulat merupakan bilangan genap. Bilangan bulat disebut genap jika ia habis dibagi dengan dua.

Penyelesaian

```
function Genap(input n : integer) → boolean
{  true jika n adalah bilangan genap, atau false jika tidak genap }

DEKLARASI
   { tidak ada }

ALGORITMA:
   return (n mod 2 = 0)
```

Algoritma 11.4 Fungsi untuk menentukan bilangan genap atau ganjil.

Perhatikan pernyataan

```
return (n mod 2 = 0)
```

di dalam fungsi Genap di atas. Ekspresi `th mod 4 = 0` adalah ekspresi *boolean* (nilai ekspresi mungkin *true* atau *false*). Pernyataan tersebut ekivalen jika ditulis sebagai berikut:

```
if n mod 2 = 0 then
   return true    { n genap }
else  { berarti, n mod 2 ≠ 0 }
   return false   { n ganjil }
endif
```

Contoh 11.4. Buatlah fungsi untuk menentukan apakah sebuah tahun merupakan tahun kabisat atau bukan kabisat. Suatu tahun disebut tahun kabisat jika tahun tersebut habis dibagi 4 atau habis dibagi 400 jika tahun tersebut kelipatan 100.

Penyelesaian

```
function Kabisat(input th : integer) →boolean
{  true jika th adalah tahun kabisat, atau false jika tidak }

DEKLARASI
   { tidak ada }

ALGORITMA:
   if (th mod 4 = 0 and  th mod 100 ≠ 0) or (th mod 400 = 0) then
      return true    { tahun kabisat }
   else
      return false     { bukan tahun kabisat }
   endif
```

Algoritma 11.5 Fungsi untuk menentukan tahun kabisat.

Contoh 11.5. Tulislah fungsi untuk menentukan nama-nama bulan berdasarkan nomor bulannya (1 sampai 12).

Penyelesaian

```
function NamaBulan(input bln : integer) → string
{ mengembalikan nama bulan berdasarkan nomor bln }

DEKLARASI
  nama : string

ALGORITMA:
  case bln
    1 : nama ← 'januari'
    2 : nama ← 'februari'
    3 : nama ← 'maret'
    4 : nama ← 'april'
    5 : nama ← 'mei'
    6 : nama ← 'juni'
    7 : nama ← 'juli'
    8 : nama ← 'agustus'
    9 : nama ← 'september'
    10 : nama ← 'oktober'
    11 : nama ← 'november'
    12 : nama ← 'desember'
  endcase

  return nama
```

Algoritma 11.6 Fungsi nama bulan (versi 1: menggunakan peubah).

```
function NamaBulan(input bln : integer) →string
{ mengembalikan nama bulan berdasarkan nomor bln }

DEKLARASI
  { tidak ada }

ALGORITMA:
  case bln
    1 : return 'januari'
    2 : return 'februari'
    3 : return 'maret'
    4 : return 'april'
    5 : return 'mei'
    6 : return 'juni'
    7 : return 'juli'
    8 : return 'agustus'
    9 : return 'september'
    10 : return 'oktober'
    11 : return 'november'
    12 : return 'desember'
  endcase
```

Algoritma 11.7 Fungsi nama bulan (versi 2: tanpa menggunakan peubah).

Contoh 11.6. Tulislah fungsi untuk mengonversi karakter digit ('0' .. '9') ke angka (0 .. 9).

Penyelesaian

```
function KarKeInt(input c : char) →integer
{ mengkonversi karakter digit ('0'..'9') ke integer (0..9) }

DEKLARASI
  { tidak ada }

ALGORITMA:
  case c
    '0' : return 0
    '1' : return 1
    '2' : return 2
    '3' : return 3
    '4' : return 4
    '5' : return 5
    '6' : return 6
    '7' : return 7
    '8' : return 8
    '9' : return 9
  endcase
```

Algoritma 11.8 Fungsi konversi karakter ke *integer*.

Contoh 11.7. Buatlah fungsi yang memberikan nilai maksimum dari dua buah nilai bilangan bulat a dan b.

Penyelesaian

```
function Maks(input a, b : integer) →integer
{ mengembalikan nilai terbesar dari a dan b }

DEKLARASI
   { tidak ada }

ALGORITMA:
   if a ≥ b then
     return a
   else
     return b
   endif
```

Algoritma 11.9 Fungsi untuk menentukan nilai terbesar dari dua buah peubah.

Contoh 11.8. Buatlah fungsi untuk menghitung titik tengah dari dua buah titik dalam koordinat kartesian. Jika $P_1(x_1,y_1)$ dan $P_2(x_2,y_2)$ adalah titik, maka titik tengah P_1 dan P_2 adalah $P_t(x_t,y_t)$, yang dalam hal ini:

$x_t = (x_1 + x_2)/2$ dan $y_t = (y_1 + y_2)/2$

Penyelesaian

```
function TitikTengah(input P1, P2 : Titik)→Titik
{mengembalikan titik tengah dari P1 dan P2}

DEKLARASI
   Pt : Titik

ALGORITMA:
   Pt.x ← (P1.x + P2.x)/2
   Pt.y ← (P1.y + P2.y)/2
   return Pt
```

Algoritma 11.10 Fungsi untuk menentukan titik tengah dari dua buah titik.

Contoh 11.9. Buatlah fungsi untuk memeriksa pemeriksaan kebenaran sebuah sandi-lewat (*password*). Sandi-lewat *p* dibandingkan dengan kata sandi yang dirahasiakan. Bila *p* sama dengan kata sandi, maka sandi *p* sah (benar).

Penyelesaian

```
function Valid(input  p : string) → boolean
{ true jika password p benar, atau false jika tidak }

DEKLARASI
  const password = 'abc123'  { sandi-lewat yang benar }

ALGORITMA:
   return (p = password)
```

Algoritma 11.11 Fungsi untuk memeriksa kebenaran sandi-lewat.

Contoh 11.10. Buatlah fungsi untuk menghitung nilai faktorial dari bilangan bulat tidak negatif. Faktorial dari *n* didefinisikan sebagai:

$$n! = \begin{cases} 1 & , \text{jika } n = 0 \\ 1 \times 2 \times 3 \times \ldots \times (n\text{-}1) \times n & , \text{jika } n > 0 \end{cases}$$

Misalnya, $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$

Penyelesaian

```
function Fak(input n : integer) →integer
{ mengembalikan nilai n!, untuk n ≥ 0 }

DEKLARASI
   i, f : integer

ALGORITMA:
   f ← 1
   for i ← 1 to n do
     f ← f  * i
```

```
endfor
return f
```

Algoritma 11.12 Fungsi untuk menghitung faktorial, n!.

Contoh 11.11. Buatlah fungsi untuk menghitung perpangkatan a^n, $n \geq 0$, $a \in \mathbf{R}$. Perpangkatan a^n berarti mengalikan a sebanyak n kali:

$$a^n = a \times a \times a \times a \times ... \times a \quad \{\text{sebanyak } n \text{ kali}\}$$

Contohnya,

$5^4 = 5 \times 5 \times 5 \times 5 = 625$
$2^8 = 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 256$
$3^0 = 1$

Penyelesaian

```
function Pangkat(input a : real, input n: integer) → real
{ mengembalikan nilai perpangkatan x^n }

DEKLARASI
   p : real
   i : integer

ALGORITMA:
   p ← 1
   for i ← 1 to n do
     p ← p * a
   endfor
   return p
```

Algoritma 11.13 Fungsi untuk menghitung perpangkatan, a^n, $n \geq 0$, $x \in \mathbf{R}$.

11.3 Pemanggilan Fungsi

Fungsi diakses dengan cara memangil namanya dari program pemanggil, diikuti dengan daftar parameter aktual (bila ada). Karena fungsi menghasilkan nilai, maka nilai tersebut dapat dieprlakukan dengan dua cara. Pertama, nilai yang dikembalikan oleh fungsi ditampung di dalam sebuah peubah yang bertipe sama dengan tipe fungsi,

```
peubah ← NamaFungsi(parameter aktual, jika ada);
```

misalnya (berdasarkan contoh-contoh fungsi yang sudah diberikan di atas),

```
y ← F(5)          { y harus bertipe real }
z ← H(x,y,z)      { z harus bertipe integer; x, y, dan z harus sudah
                    terdefinisi nilainya}
hh ← Genap(m)     { hh bertipe boolean; m harus sudah terfenisi nilainya }
```

Kedua, nilai yang dikembalikan oleh oleh fungsi dapat langsung dimanipulasi seperti pada contoh-contoh berikut:

```
write(F(5))
p ← 2*H(x,y,z)- x + y
if Genap(m) then ...
```

Contoh 11.12. Misalkan kita ingin menampilkan tabel yang berisi nilai-nilai x dan $F(x)$ di dalam selang [10,15] dengan $\Delta x = 0.2$, seperti contoh berikut:

x	$F(x)$
10.0	242.0
10.2	251.08
10.4	...
...	...
...	...
14.8	...
15.0	...

Fungsi $F(x) = 2x^2 + 5x - 8$ sudah didefinisikan di dalam Algoritma 11.1. Kita akan memanggil fungsi tersebut dari program utama.

Penyelesaian

```
PROGRAM TabelFungsi
{ Program utama yang memperagakan cara pemanggilan fungsi F. Program
ini menampilkan tabel nilai-nilai x dan f(x) di dalam selang [10,15]
dengan Δx = 0.2  }

DEKLARASI
    x : real

    function F(input x : real)→real
    { mengembalikan nilai F(x)= 2x² + 5x - 8, x ∈ R }

ALGORITMA:
    { buat header tabel }
    write('------------------------------')
    write('   x            f(x)          ')
    write('------------------------------')

    x ← 10.0
    while x ≤ 15.0 do
      write(x,'    ',F(x))
      x ← x + 0.2
    endwhile

    { buat garis penutup tabel }
    write('------------------------------')
```

Algoritma 11.14 Program untuk mencetak tabel nilai-nilai fungsi F(x) di dalam selang [10,15].

Perhatikan Algoritma 11.14 ini. Fungsi F dipanggil dari pernyataan

```
write(x,'   ',F(x))
```

yang dalam hal ini, nilai fungsi F untuk x yang diberikan langsung dicetak.

Contoh 11.13. Dibaca sebuah tahun, harus ditentukan apakah tahun tersebut merupakan tahun kabisat atau bukan.

Penyelesaian

```
PROGRAM TahunKabisat
{ Program utama menentukan apakah suatu tahun merupakan tahun kabisat.
}

DEKLARASI
  tahun : integer

  function Kabisat(input th : integer)→boolean
  { true jika th adalah tahun kabisat, atau false jika bukan kabisat }

ALGORITMA:
   read(tahun)
   if Kabisat(tahun) then
      write('tahun kabisat')
   else { berarti Kabisat(th) = false }
      write('bukan tahun kabisat')
   endif
```

Algoritma 11.15 Program untuk menentukan apakah sebuah tahun merupakan tahun kabisat atau bukan.

Contoh 11.14. Dibaca dua buah nilai *integer*, *a* dan *b*. Tentukan nilai yang terbesar dari keduanya.

Penyelesaian

```
PROGRAM TentukanMaksimum
{ Program yang memanggil fungsi Maks untuk menentukan nilai maksimum
dari dua buah masukan nilai bilangan bulat a dan b. }

DEKLARASI
  a, b : integer

   function Maks(input a, b : integer)→integer
  { mengembalikan harga terbesar dari a dan b }

ALGORITMA:
   read(a,b)
   write(Maks(a, b))
```

Algoritma 11.16 Program untuk menentukan nilai terbesar (maksimum) dari dua buah nilai integer.

Contoh 11.15. Dibaca dua buah titik, P_1 (x_1,y_1) dan $P_2(x_2,y_2)$, cetaklah titik tengah dari dua buah titik tersebut.

Penyelesaian

```
PROGRAM MenghitungTitikTengah
{ Program untuk menghitung titik tengah dari dua buah titik di
bidang datar. }

DEKLARASI
   type Titik : record < x:real,
                         y:real
                       >
   P1, P2 : Titik
   Pt     : Titik     { titik tengah P1 dan P2 }

   function TitikTengah(input P1, P2 : Titik)→Titik
   { mengembalikan titik tengah dari P1 dan P2}

ALGORITMA:
    read(P1.x, P1.y)
    read(P2.x, P2.y)
    Pt ← TitikTengah(P1,P2)
    write(Pt.x, Pt.y)
```

Algoritma 11.17 Program untuk menentukan titik tengah dari dua buah titik.

Perhatikan Algoritma 11.17 ini. Nilai yang dikembalikan oleh fungsi `TitikTengah` ditampung di dalam peubah `Pt`:

```
Pt ← TitikTengah(P1,P2)
```

Contoh 11.16. Dibaca sebuah sandi-lewat (*password*), periksa apakah *password* merupakan sandi-lewat yang sah. Bila sandi-lewat tidak sah, ulangi lagi pengisiannya (maksimal hanya boleh 3 kali pengisian sandi-lewat).

Penyelesaian

```
PROGRAM PemeriksaanKebenaranSandiLewat
{ Memeriksa kebenaran sandi-lewat. Sandi-lewat dibaca dari papan
ketik. Bila sandi-lewat salah, pemasukan sandi-lewat dapat diulang
lagi maksimal 3 kali }

DEKLARASI
  SandiLewat : string     { kata sandi yang dibaca dari papan ketik }
  sah : boolean           { true jika password benar, false jika salah }
  i : integer             { pencatat jumlah pembacaan sandilewat }

  function Valid(input p : string)→boolean
  { true jika password p benar, atau false jika tidak }

ALGORITMA:
   i ← 1
   sah ← false
   repeat
     read(SandiLewat)
     if Valid(SandiLewat) then
```

```
      sah ← true
    else { SandiLewat ≠ password }
      i ← i + 1
    endif
  until (sah) or (i > 3)
  if not sah then
    write('sandi lewat salah. Anda tidak punya hak mengakses sistem')
  endif
```

Algoritma 11.18 Program untuk memeriksa kebenaran sandi-lewat.

Contoh 11.17. Tulislah fungsi untuk menghitung nilai hampiran *exp(x)* yang didefinisikan dengan deret berikut:

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

Hampiri nilai e^x dengan deret sampai 10 buah suku (n = 10; perhatikan bahwa semakin besar nilai n semakin teliti nilai hampiran *exp(x)*).

Penyelesaian

Penghampiran nilai *exp(x)* membutuhkan perhitungan pangkat dan faktorial. Kita dapat memanfaatkan fungsi `Fak` dan `Pangkat` yang sudah ditulis pada Contoh 10.9 dan Contoh 10.10. Nilai n dapat dideklarasikan sebagai konstanta, misalnya 10. Perhatikan juga bahwa deret *exp(x)* sebenarnya ditulis lebih jelas sebagai

$$e^x \approx \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

Tiap suku dapat dinyatakan dalam bentuk $\frac{x^k}{k!}$, $k = 0, 1, 2, \ldots, n$.

Jadi, jumlah deret adalah $S \leftarrow S + x^k/k!$, yang dalam hal ini,

S diinisialisasi dengan 0
x^k dihitung dengan fungsi *Pangkat*
$k!$ dihitung dengan fungsi *Fak*

```
function Exp(input x : real) → real
{ mengembalikan nilai exp(x) }

DEKLARASI
  const n : integer = 10
  S : real
  k : integer

  function Fak(input n : integer) → integer
  { mengembalikan nilai n!, untuk n ≥ 0 }
```

```
   function Pangkat(input x : real, input m : integer) → real
   { mengembalikan nilai perpangkatan x^m }

ALGORITMA:
   S ← 0
   for k ← 0 to n do
      S ← S + Pangkat(x,k)/FAK(k)
   endfor
   return S
```

Algoritma 11.19 Fungsi hampiran exp(x).

Perhatikanlah bahwa di dalam Algoritma 11.19, fungsi `Pangkat` dan `Fak` dideklarasikan di dalam fungsi `Exp`. Contoh ini sekaligus memperlihatkan pemanggilan fungsi dari dalam fungsi lain.
Contoh program pemanggil fungsi `Exp`:

```
PROGRAM CetakTabelEXPX
{ Program untuk menghitung nilai exp(x) dari x = 0 sampai x = 100
dengan pertambahan Δx = 0.5 }

DEKLARASI
   x : real
   function Exp(input x:real)→real
   { mengembalikan nilai exp(x) }

ALGORITMA:
   x ← 0.0
   while x ≤ 100.0 do
     write(x, Exp(x))
     x ← x + 0.5
   endwhile
     { x > 100 }
```

Algoritma 11.20 Mencetak tabel nilai x dan exp(x) di dalam selang [0, 100].

Kompilator bahasa pemrograman menyediakan fungsi-fungsi pustaka (*library function*) yang siap pakai. Spesifikasi fungsi pusataka tersebut dijelaskan di dalam buku *reference guide* bahasa pemrograman bersangkutan. Penjelasan tersebut berisi apa yang dilakukan fungsi, parameternya, tipe paramater, dan tipe hasilnya.

Misalnya:

```
function sin(input x:real) → real
{ mengembalikan harga sinus x, x dalam radian }
function cos(input x:real) → real
{ mengembalikan harga cosinus x, x dalam radian }
function sqrt(input x : real) → real
{ mengembalikan harga √x }
function abs(input x:real) → real
```

```
{ mengembalikan harga mutlak x }
function ord(input x:char) → integer
{ mengembalikan nilai ordinal karakter ASCII dari x }
function trunc(input x : real) → integer
{ memotong x pada bagian bilangan bulatnya }
```

dan lain-lain sebagainya.

11.4 Prosedur atau Fungsi?

Pertanyaan yang seringkali muncul dalam pemrograman modular adalah: apakah sebuah modul program akan dibuat sebagai prosedur atau fungsi? Jawaban untuk pertanyaan ini sebenarnya tidak sulit: fungsi digunakan apabila modul program mengembalikan sebuah nilai, sementara prosedur digunakan bila modul menghasilkan efek *netto* dari (satu atau) sekumpulan aksi. Namun dalam praktek, sering perbedaan antara keduanya tidak jelas, karena sebuah prosedur dapat juga ditulis sebagai fungsi, demikian pula sebaliknya. Pemilihan apakah sebuah modul direalisasikan sebagai fungsi atau prosedur bergantung pada kebutuhan dan seni memprogram.

(a) Mengubah fungsi menjadi prosedur

Sebuah fungsi dapat dikonversi sebagai prosedur dengan cara menyatakan nilai yang dikembalikan (*return value*) oleh fungsi tersebut sebagai parameter keluaran pada prosedur.

Sebagai contoh, tinjau kembali fungsi `Maks` untuk menentukan bilangan terbesar di antara dua buah bilangan.

Fungsi:

```
function Maks(input a, b: integer) → integer
{ mengembalikan harga terbesar dari a dan b }

DEKLARASI
  { tidak ada }

ALGORITMA:
   if a ≥ b then
     return a
   else
     return b
   endif
```

Prosedur:

```
procedure TentukanMaks(input a, b: integer, output maks : integer)
{ Menentukan nilai terbesar dari a dan b, dan menyimpannya di max }
{ K.Awal: a dan b sudah terdefinisi nilainya. }
{ K.Akhir: maks berisi nilai terbesar dari a atau b }
```

```
DEKLARASI
   { tidak ada }

ALGORITMA:
   if a ≥ b then
      maks ← a
   else
      maks ← b
   endif
```

(b) Mengubah fungsi menjadi prosedur

Prosedur yang mempunyai satu buah parameter keluaran dapat ditulis sebagai fungsi dengan cara menyatakan parameter keluaran sebagai nilai yang dikembalikan oleh fungsi.

Sebagai contoh, tinjau prosedur menghitung nilai rata-rata dari sejumlah data bilangan bulat sebagai berikut:

Prosedur:

```
procedure HitungRataRata(input Ndata : integer, output U : real)
{ Membaca data, menjumlahkannya, dan menghitung rata-atanya. }
{ K.Awal: Ndata sudah bersisi banyaknya data; Ndata > 0 }
{ K.Akhir: U berisi rata-rata seluruh  data  }

DEKLARASI
    x : integer        { data yang dibaca }
    i : integer        { pencacah banyaknya data }
    jumlah : integer   { jumlah nilai seleuruh data }

ALGORITMA:
    jumlah ← 0      { inisialisasi penjumlah }
    for i ← 1 to Ndata do
      read(x)
      jumlah ← jumlah + i
    endfor

    U ← jumlah/Ndata  { rata-rata seluruh data }
```

Prosedur `HitungRataRata` memiliki satu parameter keluaran, karena itu ia dapat diubah menjadi fungsi. Apabila ditulis sebagai fungsi, maka parameter keluaran `U` tidak diperlukan lagi, karena nilai `U` merupakan nilai yang dihasilkan (*return value*) oleh fungsi.

Fungsi *RataRata*:

```
function RataRata(input Ndata : integer) →real
{ Membaca data, menjumlahkannya, dan menghitung rata-ratanya.
Banyaknya data adalah Ndata (> 0). }

DEKLARASI
    x : integer        { data yang dibaca }
```

```
    i : integer          { pencacah banyaknya data }
    jumlah : integer     { jumlah nilai seluruh data }

ALGORITMA:
    jumlah ← 0    { inisialisasi penjumlah }
    for i ← 1 to Ndata do
      read(x)
      jumlah ← jumlah + x
    endfor
    return jumlah/Ndata   { rata-rata seluruh data }
```

Umumnya modul program direalisasikan sebagai fungsi apabila nilai fungsi digunakan langsung dalam suatu ekspresi misalnya:

(a) `z ← ( f(a)+f(b) )/2`

(b)
```
if f(a)*f(b) > 0 then
  write('Selang [a,b] tidak mengandung akar fungsi')
endif
```

dengan f adalah fungsi yang pendefinisiannya sebagai berikut:

```
function f(input x:real)→real
{ mengembalikan nilai f(x) = 2*x*x + 5*x - 8 }

DEKLARASI
   { tidak ada }
ALGORITMA:
   return 2*x*x + 5*x - 8
```

Contoh-contoh lain seperti fungsi menghitung faktorial dan menghitung pangkat memperlihatkan bahwa masalah-masalah tersebut lebih elegan ditulis sebagai fungsi ketimbang prosedur. Penggunaan fungsi `Fak` dan fungsi `Pangkat` ditunjukkan pada waktu mengitung fungsi `Exp` (Contoh 11.17).

11.5 Contoh-contoh Tambahan

Contoh 11.18. Tinjau kembali algoritma Euclidean untuk menghitung pembagi bersama terbesar (*gcd*) dari dua buah bilangan bulat tak-negatif m dan n ($m \geq n$). Harga m dan n dibaca dari piranti masukan (Pembaca harap membaca kembali contoh algoritma pertama yang diberikan di dalam Bab 1). Tulis kembali algoritma Euclidean ke dalam bentuk fungsi.

Penyelesaian

```
function gcd(input m, n : integer) → integer
{ Mengembalikan pembagi bersama terbesar (gcd) dari dua buah bilangan
bulat tak-negatif m dan n, dengan syarat m ≥ n dan m,n ≥ 0. }

DEKLARASI
   r     : integer    { sisa pembagian m dengan n }
```

```
ALGORITMA:
   while n ≠ 0 do
      r ← m mod n
      m ← n
      n ← r
   endwhile
   { n = 0, maka  gcd = m}
   return m
```

Algoritma 11.21 Menghitung gcd dari dua buah nilai *integer*.

Hal lain yang harus diperhatikan adalah nilai m dan n. Fungsi gcd hanya benar untuk nilai m ≥ n. Karena itu, sebelum pemanggilan fungsi, nilai m dan n harus diperiksa terlebih dahulu. Jika m < n, lakukan pertukaran m dan n.

Contoh program utama:

```
PROGRAM Euclidean
{ Program untuk menghitung pembagi bersama terbesar dari dua buah
  bilangan bulat positif m dan n. }

DEKLARASI
   m, n : integer  { > 0 }
   function gcd(input m, n : integer) → integer
   { Mengembalikan pembagi bersama terbesar dari dua buah bilangan
bulat tak-negatif m dan n, dengan syarat m ≥ n dan m,n ≥ 0. }

   procedure Tukar(input/output a, b : integer)
   { Mempertukarkan nilai a dan b. }

ALGORITMA:
   read(m,n)
   if m < n then
      { pertukarkan m dan n }
      Tukar(m,n)
   endif
   write(gcd(m,n))
```

Contoh 11.19. Tuliskan fungsi untuk menentukan apakah sebuah bilangan bulat positif m merupakan bilangan prima. Misalnya, $m = 10$ bukan bilangan prima, tetapi $m = 17$ adalah bilangan prima. Keluaran dari fungsi adalah *true* jika m prima dan *false* jika m bukan prima.

Penyelesaian

Cara yang paling sederhana memeriksa keprimaan bilangan bulat positif m adalah dengan membagi m dengan 2, 3, ..., $m - 1$. Jika ada suatu pembagian yang habis (yaitu bersisa 0), maka kita simpulkan m bukan bilangan prima.

```
function Prima(input m : integer)→boolean
{ mengembalikan true jika m bilangan prima, false jika m bukan prima }
```

```
DEKLARASI
   i     : integer
   prim  : Boolean

ALGORITMA:
   i ← 2
   prim ← true    { asumsikan m pada awalnya bilangan prima }
   while (i≤ m-1) and (not prim) do
     if m mod i = 0 then    { m habis dibagi dengan bilangan i ≤ m - 1}
       prim ← false          { kesimpulannya: m bukan bilangan prima }
     else
       i ← i + 1             { coba bagi dengan i berikutnya }
     endif
   endwhile

  return prim
```

Algoritma 11.22 Fungsi untuk memeriksa keprimaan bilanagn bulat positif.

Catatan:
Teorema yang lebih maju di dalam matematika menyebutkan bahwa untuk keprimaan bilangan bulat positif $m \geq 2$, kita cukup menguji bilangan bulat dari 2 sampai $\sqrt{m}$ sebagai pembagi yang mungkin.

Contoh 11.20. Bilangan Fibonacci yang ke-*i* didefinisikan sebagai berikut:

$Fib_i = Fib_{i-1} + Fib_{i-2}$

Didefinisikan juga bahwa $Fib_0 = 0$ dan $Fib_1 = 1$. Jadi, 10 bilangan Fibonacci yang pertama adalah

0 1 1 2 3 5 8 13 21 34

Tuliskan sebuah fungsi yang mengembalikan bilangan Fibonacci ke-*n*.

Penyelesaian

Dari definisi bilangan Fibonacci, tampak bahwa bilangan Fibonacci yang ke-*i* adalah jumlah dua bilangan Fibonacci yang sebelumnya. Kasus khusus adalah bila $i \leq 1$, sebab $Fib_i = i$.

```
function Fib(input n : integer)→integer
{ mengembalikan bilangan Fibonacci yang ke-n  (n ≥ 0) }

DEKLARASI
   i, p, q, r  : integer

ALGORITMA:
  if n ≤ 1 then
     return n
  else
     p ← 0    { Fib0 }
```

```
    q ← 1    { Fib1 }
    for i←2 to n do
      r ← p + q    { Fibi = Fibi-2 + Fibi-1 }
      p ← q        { Fibi-2 yang baru }
      q ← r        { Fibi-1 yang baru }
    endfor
    return r
  endif
```

Algoritma 11.23 Fungsi untuk membangkitkan bilangan Fibonacci.

11.6 Translasi Notasi Algoritmik untuk Fungsi ke dalam Notasi Bahasa Pascal dan Bahasa C

ALGORITMIK:

```
function NamaFungsi( deklarasi parameter, jika ada)→tipe
{  spesifikasi fungsi, menjelaskan apa yang dilakukan dan yang
   dikembalikan oleh fungsi. }

DEKLARASI
{ semua nama yang dipakai di dalam fungsi dan hanya berlaku lokal di
dalam prosedur didefinisikan di sini }

ALGORITMA:
  {  badan fungsi, berisi instruksi-instruksi untuk menghasilkan nilai
     yang akan dikembalikan oleh fungsi }

  return ekspresi  { pengembalian nilai yang dihasilkan fungsi }
```

PASCAL:

```
function NamaFungsi( deklarasi parameter, jika ada): tipe;
{  spesifikasi fungsi, menjelaskan apa yang dilakukan dan yang
   dikembalikan oleh fungsi.
}

(* DEKLARASI *)
{ semua nama yang dipakai di dalam fungsi dan hanya berlaku lokal di
dalam prosedur didefinisikan di sini }

(* ALGORITMA: *)
begin
  {  badan fungsi, berisi instruksi-instruksi untuk menghasilkan nilai
     yang akan dikembalikan oleh fungsi }

  NamaFungsi:=ekspresi;  { pengembalian nilai yang dihasilkan fungsi }

End;
```

Catatan:

1. *dalam bahasa Pascal, fungsi hanya dapat mengembalikan nilai bertipe sederhana (integer, real, boolean, char, dan string). Karena itu, tipe hasil dari fungsi haruslah dari tipe sederhana. Manipulasi fungsi yang menghasilkan nilai bertipe bentukan akan dijelaskan pada bagian lain sesudah ini;*
2. *apabila fungsi tidak memiliki daftar parameter formal, maka tanda kurung buka "(" dan kurung tutup ")" tidak ditulis;*
3. *fungsi dideklarasikan sekaligus didefinisikan di dalam blok program pemanggil (sesudah kata kunci* `var`*).*

C:

```
tipe NamaFungsi(deklrasi parameter, jika ada);
/*  spesifikasi fungsi, menjelaskan apa yang dilakukan dan yang
dikembalikan oleh fungsi
*/

{
  /* DEKLARASI */
  /* semua nama yang dipakai di dalam fungsi dan hanya berlaku lokal
   di dalam prosedur didefinisikan di sini  */

  /* ALGORITMA: */
  /*  badan fungsi, berisi instruksi-instruksi untuk menghasilkan
   nilai yang akan dikembalikan oleh fungsi */

   return ekspresi;  /* pengembalian nilai yang dihasilkan fungsi */
}
```

Catatan:

1. *tidak seperti bahasa Pascal, dalam bahasa C fungsi dapat mengembalikan nilai bertipe sederhana (integer, real, boolean, char, dan string) maupun bertipe bentukan;*
2. *apabila fungsi tidak memiliki daftar parameter formal, maka tanda kurung buka "(" dan kurung tutup ")" tetap harus ditulis. Hal yang sama juga berlaku pada waktu pemanggilannya;*
3. *sebagaimana halnya pada prosedur, fungsi didefinisikan di luar blok main(), dan purwarupa fungsi dideklarasikan sebelum blok main().*

Beberapa Contoh Translasi:

1. Fungsi $F(x) = 2x^2 + 5x - 8$ dan program pemanggilnya.

ALGORITMIK:

Fungsi:

```
function F(input x: real)→real
```

```
{ mengembalikan nilai F(x) = 2x² + 5x - 8, x ∈ R }

DEKLARASI
  { tidak ada }

ALGORITMA:
  return 2*x*x + 5*x - 8
```

Program utama:

```
PROGRAM TabelFungsi
{ Program utama yang memperagakan cara pemanggilan fungsi F. Program
ini menampilkan tabel nilai-nilai x dan f(x) di dalam selang [10,15]
dengan Δx = 0.2  }

DEKLARASI
    x : real

    function F(input x : real)→real
    { mengembalikan nilai F(x)= 2x² + 5x - 8, x ∈ R }

ALGORITMA:
    { buat header tabel }
    write('------------------------------')
    write('   x            f(x)          ')
    write('------------------------------')

    x ← 10.0
    while x ≤ 15.0 do
      write(x,'    ',F(x))
      x ← x + 0.2
    endwhile

    { buat garis penutup tabel }
    write('------------------------------')
```

PASCAL:

```
PROGRAM TabelFungsi;
{ Program utama yang memperagakan cara pemanggilan fungsi F. Program
ini menampilkan tabel nilai-nilai x dan f(x) di dalam selang [10,15]
dengan Δx = 0.2  }

(* DEKLARASI GLOBAL *)
var
   x : real;

   { fungsi yang akan digunakan dideklarasikan purwarupanya di sini }

   function F(x: real):real;
   { mengembalikan nilai F(x)= 2x*x + 5x - 8, x bertipe real }

   (* DEKLARASI *)
      {'tidak ada }
(* ALGORITMA: *)
   begin
```

```
      F:=2*x*x + 5*x - 8;
   end;

(* PROGRAM UTAMA *)
begin
    { buat header tabel }
    writeln('--------------------------------');
    writeln('   x            f(x)           ');
    writeln('--------------------------------');

    x:=10.0
    while x <= 15.0 do
     begin
       writeln(x:4:2,'    ',F(x):10:4);
       x := x + 0.2;
     end;{while}

    { buat garis penutup tabel }
    writeln('--------------------------------');
end.
```

C:

```
/* PROGRAM TabelFungsi */
/* Program utama yang memperagakan cara pemanggilan fungsi F. Program
ini menampilkan tabel nilai-nilai x dan f(x) di dalam selang [10,15]
dengan Δx = 0.2  */

#include <stdio.h>

/* purwarupa fungsi */
float F(float x);
/* mengembalikan nilai F(x)=2x*x + 5x - 8, x bertipe real */

main()
{
    /* DEKLARASI */
    float x;

    /*ALGORITMA:*/

    /* buat header tabel */
    printf("-------------------------------- \n");
    printf("   x            f(x)             \n");
    printf("-------------------------------- \n");
    x = 10.0;
    while (x <= 15.0)
     {
       printf("%f:4:2      %f:10:4 \n", x,F(x));
       x = x + 0.2;
     } /*while*/

    /* buat garis penutup tabel */
    printf("-------------------------------- \n");
 }

float F(float x);
```

```
/* mengembalikan nilai F(x)=2x*x + 5x - 8, x bertipe real */
{
  /* DEKLARASI */
     /* tidak ada */

  /* ALGORITMA: */
   return 2*x*x + 5*x - 8;
}
```

2. Fungsi untuk menentukan bilangan genap dan program utamanya.

ALGORITMIK:

Fungsi:

```
function Genap(input n : integer) →boolean
{ true jika n adalah bilangan genap, atau false jika sebaliknya }

DEKLARASI
  { tidak ada }

ALGORITMA:
   return (n mod 2 = 0)
```

Program pemanggil:

```
PROGRAM GenapGanjil
{ Program untuk menentukan apakah sebuah bilangan genap atau ganjil }

DEKLARASI
  x : integer

  function Genap(input n : integer)→boolean
  { true jika n adalah bilangan genap, atau false jika sebaliknya }

ALGORITMA:
   read(x)
   if Genap(x) then
      write('genap')
   else
      write('ganjil')
   endif
```

PASCAL:

```
program GenapGanjil;
{ Program utama menentukan apakah sebuah bilangan genap atau ganjil }

(* DEKLARASI *)
var
  x : integer;

  function Genap(n:integer):boolean;
  { true jika n adalah bilangan genep, atau false jika sebaliknya }
```

```
  begin
     Genap:=(n mod 2 = 0);
  end;

(* ALGORITMA: *)
begin
   write('Ketikkan sembarang bilangan bulat : '); readln(x);
   if Genap(x) then
      writeln('genap' )
   else
      writeln('ganjil');
   {endif}
end.
```

C:

```
/* Program GenapGanjil */
/* Program utama menentukan apakah sebuah bilangan genap atau ganjil */

#include <stdio.h>

typedef enum{0=false,1=true) boolean;

boolean Genap(int n);

main()
{
  /* DEKLARASI */
  int x;

/* ALGORITMA: */
  printf("Ketikkan sembarang bilangan bulat : "); scanf("%d",&x);
  if (Genap(x))
     printf("genap \n");
  else
     printf("ganjil \n");
  /*endif}*/
}

boolean Genap(int n)
{ true jika n adalah bilangan genap, atau false jika sebaliknya }
{
   return (n % 2 == 0);
}
```

3. Fungsi yang mengembalikan hasil bertipe bentukan.

ALGORITMIK:

Fungsi:

```
function TitikTengah(input P1, P2 : Titik)→Titik
{ mengembalikan titik tengah dari P1 dan P2}

DEKLARASI
   Pt : Titik
```

```
ALGORITMA:
   Pt.x ← (P1.x + P2.x)/2
   Pt.y ← (P1.y + P2.y)/2
   return Pt
```

Program pemanggil:

```
PROGRAM HitungTitikTengah
{ Program untuk menghitung titik tengah dari dua buah titik di bidang datar }

DEKLARASI
   type Titik : record <x:real,
                        y:real
                       >
   P1, P2 : Titik
   Pt     : Titik    { titik tengah P1 dan P2 }

   function TitikTengah(input P1, P2 : Titik)→Titik
   { mengembalikan titik tengah dari P1 dan P2 }

ALGORITMA:
    read(P1.x, P1.y)
    read(P2.x, P2.y)
    Pt ← TitikTengah(P1,P2)
    write(Pt.x, Pt.y)
```

PASCAL

Fungsi di dalam bahasa *Pascal* tidak dapat mengembalikan nilai yang bertipe terstruktur. Karena itu, fungsi yang mengembalikan tipe terstruktur harus dimanipulasi dengan cara mengubah tipe hasilnya menjadi tipe dasar (misalnya *integer* atau *boolean*).

Di bawah ini adalah contoh yang SALAH mentranslasi fungsi dari notasi algoritmik di atas menjadi fungsi dalam bahasa *Pascal* seperti di bawah ini:

```
function TitikTengah(P1, P2 : Titik):Titik;   { SALAH! }
{ mengembalikan titik tengah dari titik P1 dan P2 }

begin
  TitikTengah.x := (P1.x + P2.x)/2;
  TitikTengah.y := (P1.y + P2.y)/2;
end;
```

Penulisan fungsi `TitikTengah` yang mengembalikan nilai bertipe `Titik` pada fungsi di atas salah, karena fungsi di dalam *Pascal* hanya dapat mengembalikan nilai yang bertipe dasar (*integer*, *real*, *char*, *string*, dan *boolean*).

Agar fungsi `TitikTengah` dapat ditranslasi dalam bahasa *Pascal*, fungsi tersebut harus dimanipulasi sehingga ia mengembalikan nilai yang bertipe sederhana sembarang (misal *integer*), sementara titik tengah disimpan dalam sebuah *parameter by reference*.

```
function TitikTengah(P1, P2 : Titik; var Pt : Titik):integer; { BENAR!}
{ mengembalikan titik tengah dari titik P1 dan P2 }

begin
  Pt.x := (P1.x + P2.x)/2;
  Pt.y := (P1.y + P2.y)/2;

    { karena fungsi harus mengembalikan nilai, isi TitikTengah dengan
      nilai sembarang, misalkan 1 }

   TitikTengah := 1;

end;
```

Dengan demikian, program menghitung titik tengah di atas menjadi seperti di bawah ini:

```
program HitungTitikTengah;
{ Program untuk menghitung titik tengah dari dua buah titik di bidang
datar }

(* DEKLARASI *)
   type Titik = record
                  x:real;
                  y:real;
                end;

   P1, P2 : Titik;
   Pt     : Titik;   { titik tengah P1 dan P2 }

   y : integer;

   function TitikTengah(P1, P2 : Titik; var Pt : Titik):integer;
   { mengembalikan titik tengah dari titik P1 dan P2 }
   begin
     Pt.x := (P1.x + P2.x)/2;
     Pt.y := (P1.y + P2.y)/2;

      { karena fungsi harus mengembalikan nilai, isi
        TitikTengah dengan nilai sembarang, misalkan 1 }
      TitikTengah:=1;
    end;

(* ALGORITMA: *)
begin
   writeln(' Titik P1:');
   write('x = ?'); readln(P1.x);
   write('y = ?'); readln(P1.y);

   writeln(' Titik P2:');
   write('x = ?'); readln(P2.x);
   write('y = ?'); readln(P2.y);

   y := TitikTengah(P1,P2,Pt);

   writeln('Titik tengah = (',Pt.x, ',' , Pt.y, ')');

    { Perhatikanlah, nilai y sendiri tidak dipakai }
end.
```

C:

Tidak seperti bahasa *Pascal*, fungsi dalam bahasa *C* dapat mengembalikan nilai bertipe bentukan (terstruktur).

```
/* PROGRAM HitungTitikTengah */
/* Program untuk menghitung titik tengah dari dua buah titik di bidang. */

#include <stdio.h>

(* DEKLARASI GLOBAL *)
typedef struct { float x;
                 float y;
               } Titik;

Titik TitikTengah(Titik P1, Titik P2);
/* mengembalikan true. Pt adalah titik tengah dari P1 dan P2 */

main()
{
    /* DEKLARASI */
    Titik P1, P2;
    Titik Pt;     /* titik tengah P1 dan P2 */
    boolean y;

    /* ALGORITMA: */
    printf(" Titik P1: \n");
    printf(" x = "); scanf("%f", &P1.x);
    printf(" y = "); scanf("%f", &P1.y);

    printf(" Titik P1: \n");
    printf(" x = "); scanf("%f", &P2.x);
    printf(" y = "); scanf("%f", &P2.y);
    Pt = TitikTengah(P1,P2);
    printf("Titik tengah : (%f%,%f) \n",Pt.x, Pt.y);
}

Titik TitikTengah(Titik P1, Titik P2)
/* Mengembalikan titik tengah dari P1 dan P2 */
{
   /* DEKLARASI */
   Titik Pt;

   /* ALGORITMA: */
   Pt.x = (P1.x + P2.x)/2;
   Pt.y = (P1.y + P2.y)/2;
   return Pt;
}
```

Soal Latihan Bab 11

1. Realisasikan fungsi-fungsi berikut:

```
function abs(input x:real) → real
{ mengembalikan harga mutlak x. Contoh: jika x = -10 maka nilai
  mutlaknya 10, jika x = 10 maka nilai mutlaknya 10 }
function double(input x:real) → real
```

```
{ mengembalikan 2 kali nilai x. Contoh: jika x = 5 maka nilai
  double-nya adalah 10 }
function odd(input x:integer) → boolean
{ mengembalikan true jika x bilangan ganjil dan false jika
  x bukan ganjil }
```

2. Buatlah fungsi `jarak` yang menerima masukan dua buah titik $P_1(x,y)$ dan $P_2(x,y)$ dan menghitung jarak kedua titik tersebut. Gunakan rumus Euclidean untuk menghitung jarak, d:

   $$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

3. Buatlah fungsi apakah_a yang memberikan nilai *true* jika karakter yang diterimanya adalah huruf '*a*' dan *false* jika tidak.

4. Buatlah fungsi `durasi` yang memberikan durasi antara dua buah jam (dengan format *hh*:*mm*:*yy*).

5. Buatlah fungsi `nextday` yang menerima masukan nama hari sekarang (senin..ahad) dan mengembalikan nama hari besoknya.

6. Buatlah fungsi `lastday` yang menerima masukan nama hari sekarang (senin..ahad) dan mengembalikan nama hari kemarennya.

7. Buatlah fungsi `roman` yang menerima angka dalam sistem desimal (1..10) dan mengembalikan angka romawinya.

8. Buatlah fungsi `lower` yang mengubah huruf kecil menjadi huruf besar (kapital).

9. Buatlah fungsi `upper` yang mengubah huruf besar menjadi huruf kecil.

10. Buatlah fungsi `phytagoras` yang menerima tiga buah bilangan bulat *a*, *b*, *c* dan menentukan apakah ketiga bilangan tersebut merupakan tripel *phytagoras*.

12
Larik

Sebuah peubah hanya menyimpan sebuah nilai. Ia tidak dapat menyimpan beberapa buah nilai yang bertipe sejenis. Dalam pemrograman, sering kita mengolah sekumpulan data yang bertipe sama, misalnya hasil ujian 100 orang mahasiswa, tabel harga barang di pasar swalayan, daftar kode wilayah dalam percakapan interlokal, dan sebagainya. Karena setiap elemen data bertipe sama, maka elemen tersebut diacu dengan nama, dan untuk membedakan elemen data yang satu dengan elemen data yang lainnya, maka elemen diacu dengan penggunaan indeks (*subscript*). Misalnya, jika data nilai ujian kita lambangkan dengan x, maka x_i menyatakan nilai ujian mahasiswa yang ke-i.

Dalam matematika, statistika, atau bidang eksakta lainnya, kita sering menemui besaran yang menggunakan nama peubah berindeks seperti

- $u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}$
- $a_1, a_2, ..., a_n$
- $v_k \geq 0$, untuk $k = 0, 1, 2, ..., n$

dan sebagainya. Besaran-besaran tersebut adalah kumpulan nilai yang bertipe sama. Nama peubah yang menyatakan kumpulan nilai itu masing-masing u, a, dan v. Nilai tertentu di dalam kumpulan diacu dengan menggunakan indeksnya, misalnya u_3, a_8, a_k, atau v_j.

Dalam kegiatan pemrograman, sekumpulan data yang bertipe sama perlu disimpan sementara di dalam memori komputer untuk sewaktu-waktu dimanipulasi. Misalnya kita ingin menghitung nilai rata-rata kumpulan data tersebut dengan rumus:

$$\text{rata-rata} = (a_1 + a_2 + \ldots + a_n)/n = \sum_{i=1}^{n} a_i$$

Sekumpulan data yang bertipe sama disimpan secara beruntun di dalam memori komputer, setiap elemen data diacu dengan menggunakan indeks. Indeks menyatakan posisi data relatif di dalam kumpulannya. Struktur penyimpanan data seperti ini dinamakan **larik** (*array*). Nama lain untuk larik adalah **tabel, vektor**, atau **peubah majemuk** (satu peubah mempunyai banyak elemen).

12.1 Apakah Larik Itu?

Larik adalah struktur data yang menyimpan sekumpulan elemen yang bertipe sama, setiap elemen diakses langsung melalui indeksnya. Indeks larik haruslah tipe data yang menyatakan keterurutan, misalnya *integer* atau karakter.

Sebuah larik yang bernama *A* dengan delapan buah elemen dapat dibayangkan secara lojik sebagai sekumpulan kotak yang terurut (baik tersusun secara vertikal atau horizontal) seperti yang diperlihatkan pada Gambar 12.1. Tiap kotak pada larik tersebut diberi indeks 1, 2, 3, ..., 8. Setiap elemen larik ditulis dengan notasi:

$A[1], A[2], A[3], A[4], A[5], A[6], A[7], A[8]$

Angka di dalam tanda kurung siku menyatakan indeks larik (notasi di atas sama saja dengan penulisan peubah ber-*subscript* $A_1, A_2, A_3, \ldots, A_8$).

Gambar 12.1 Larik A dengan 8 elemen.

Setiap elemen larik menyimpan sebuah nilai. Karena seluruh elemen larik bertipe sama, maka nilai yang disimpan oleh setiap elemen juga harus bertipe sama. Gambar 12.2 memperlihatkan larik yang bernama *A* yang setiap elemennya berisi tinggi badan (dalam centimeter) 8 orang siswa.

Tinggi siswa pertama, 158, disimpan di dalam *A*[1], tinggi siswa kedua, 157, disimpan di dalam *A*[2], demikian seterusnya.

	A
1	158
2	157
3	162
4	169
5	172
6	155
7	170
8	163

Gambar 12.2 Larik A yang setiap elemennya sudah berisi nilai.

12.2 Mendeklarasikan Larik

Larik adalah struktur data yang *statis*, artinya jumlah elemen larik harus sudah diketahui sebelum program dieksekusi. Jumlah elemen larik tidak dapat diubah, ditambah, atau dikurangi selama pelaksanaan program. Mendeklarasikan larik di dalam bagian deklarasi berarti:

1. mendefinisikan banyaknya elemen larik (ukuran larik), dan
2. mendefinisikan tipe elemen larik.

Mendefinisikan banyaknya elemen larik (atau ukuran larik) berarti memesan sejumlah tempat di memori. Komputer mengalokasikan sejumlah lokasi memori sebanyak elemen larik yang bersangkutan. Mendefinisikan tipe elemen larik berarti menetapkan tipe nilai yang dapat disimpan oleh larik. Tipe elemen larik dapat berupa tipe sederhana (*integer*, *real*, *char*, *boolean*, *string*), tipe bentukan (tipe terstruktur seperti *record*), atau bahkan bertipe larik yang lain.

Berikut contoh-contoh mendeklarasikan larik di dalam bagian deklarasi:

a. Sebagai Peubah

Misalkan:

- `A` adalah larik yang berukuran 50 buah elemen yang bertipe *integer*. Indeks larik dimulai dari 1.
- `NamaMhs` adalah larik yang berukuran 10 buah elemen yang bertipe *string*. Indeks larik dimulai dari 1.
- `NilUjian` adalah peubah larik yang berukuran 75 buah elemen yang bertipe *real*. Indeks larik dimulai dari 0.

Cara mendefinisikan ketiga buah peubah larik tersebut di dalam bagian deklarasi adalah:

```
DEKLARASI
  A        : array[1..100] of integer
  NamaMhs  : array[1..12] of string
  NilUjian : array[0..74] of real
```

Algoritma 12.1 Mendeklarasikan 3 buah larik sebagai nama peubah.

b. Sebagai Tipe Bentukan

Misalkan `LarikInt` didefinisikan sebagai nama sebuah tipe baru untuk larik yang bertipe *integer*. Ukuran larik adalah 100 buah elemen. Kita bisa mendeklarasikan sebuah peubah yang bertipe `LarikInt`, misalnya peubah `A`.

```
DEKLARASI
  type LarikInt : array[1..100] of integer    { nama tipe baru }
  A : LarikInt { A adalah sebuah peubah larik integer dengan 100 elemen }
```

Algoritma 12.2 Mendeklarasikan larik sebagai tipe bentukan.

c. Mendefinisikan ukuran larik sebagai sebuah konstanta

Misalkan `LarikInt` dideklarasikan sebagai nama sebuah tipe bentukan untuk larik yang bertipe *integer*. Ukuran maksimum larik adalah 100 buah elemen. Ukuran maksimum larik dinyatakan sebagai konstanta. Kemudian, kita bisa mendeklarasikan sebuah peubah yang bertipe `LarikInt`, misalnya peubah `A`.

```
DEKLARASI
  const Nmaks = 100    { ukuran maksimum elemen larik }
  type LarikInt : array[1..Nmaks] of integer
  A : LarikInt { A adalah sebuah peubah larik integer dengan 100 elemen }
```

Algoritma 12.3 Mendeklarasikan ukuran larik sebagai konstanta.

Perhatikan bahwa pendeklarasian larik pada Algoritma 12.3 sama dengan pendeklarasian larik pada Algoritma 12.2. Penggunaan konstanta memungkinkan kita mengubah ukuran larik cukup pada konstanta itu saja.

Tetapi, pendeklarasian larik `A` seperti di bawah ini tidak boleh karena nilai `N` tidak diketahui di awal program (ingat bahwa larik adalah struktur statis!):

```
DEKLARASI
  A : array[1..N] of integer
```

Algoritma 12.4 Pendeklarasin larik yang salah, nilai N tidak diketahui.

12.3 Cara Mengacu Elemen Larik

Elemen larik diacu melalui indeksnya. Nilai indeks harus terdefinisi. Dengan mengacu pada larik yang sudah dideklarasikan sebelum ini, berikut diberikan beberapa contoh cara mengacu elemen larik adalah:

```
A[4]                    { megacu elemen keempat dari larik A }
NamaMhs[2]              { mengacu elemen kedua dari larik NamaMhs }
A[i]                    { mengacu elemen ke-i dari larik A, asalkan
                          nilai i sudah terdefinisi }
NamaMhs[i+1]            { asalkan nilai i sudah terdefinisi }
```

Contoh-contoh memanipulasi atau menggunakan elemen larik:

```
A[4] ← 10               { mengisi elemen keempat dari larik A dengan
                          nilai 10 }

NamaMhs[i] ← 'Achmad'   { mengisi elemen ke-i dari larik NamaMhs
                          dengan string 'Achmad' }
read(A[i])              { membaca elemen ke-i dari larik A }

if A[i] < 10 then
  A[i]←A[i] + 10
else
   ... { pernyataan lainnya }
```

12.4 Pemrosesan Larik

Elemen larik tersusun di memori secara beruntun (sekuensial). Karena itu, elemennya diproses secara beruntun melalui indeksnya yang terurut. Memproses larik artinya mengunjungi (traversal) setiap elemen larik dan memanipulasi dinilai di dalamnya. Kunjungan dimulai dari elemen pertama larik, berturut-turut pada elemen berikutnya, sampai elemen terakhir dicapai, yaitu elemen dengan indeks terbesar.

Skema umum algoritma memproses larik diperlihatkan pada Algoritma 12.5.

```
PROGRAM PemrosesanLarik
{ Skema pemrosesan larik secara beruntun }

DEKLARASI
   const Nmaks = 100  { ukuran maksimum larik }
   type LarikInt  : array[1..Nmaks] of integer

   A : LarikInt
   i : integer      { indeks larik }

ALGORITMA:
  Inisialisasi
  i ← 1         { mulai dari elemen pertama }
  while i ≤ Nmaks do
```

```
    pemrosesan terhadap A[i]
    i ← i + 1  { tinjau elemen berikutnya }
  endwhile
  { i > Nmaks }
  Terminasi
```

Algoritma 12.5 Skema umum pemrosesan larik.

Pemrosesan terhadap $A[i]$ adalah aksi spesifik bergantung pada persoalan yang akan dipecahkan, misalnya pengisian nilai, pembacaan, penulisan, komputasi, atau manipulasi lainnya.

Oleh karena jumlah elemen larik sudah diketahui di awal proses, maka jumlah pengulangan juga dapat ditentukan. Oleh karena itu, kita lebih menyukai struktur *FOR* digunakan untuk untuk memproses larik, seperti ditunjukkan pada Algoritma 12.6.

```
PROGRAM PemrosesanLarik
{ Skema pemrosesan larik secara beruntun }

DEKLARASI
   const Nmaks = 100  { ukuran maksimum larik }
   type LarikInt  : array[1..Nmaks] of integer

   A : LarikInt
   i : integer      { indeks larik }

ALGORITMA:
  for i ← 1 to Nmaks do
     pemrosesan terhadap A[i]
  endfor
```

Algoritma 12.6 Skema umum pemrosesan larik dengan kontruksi FOR.

12.4.1 Ukuran Efektif Larik

Meskipun kita mendefinisikan jumlah elemen larik, seringkali kita tidak menggunakan semuanya. Bila larik A didefinisikan 100 elemen, mungkin tidak seratus elemen yang dipakai, mungkin hanya 15, 40, atau 70. Banyaknya elemen larik yang dipakai kita sebut *ukuran efektif larik*. Ukuran efektif itu kita catat di dalam peubah tertentu, misalnya n.

Di bawah ini diberikan beberapa algoritma pemrosesan larik yang disajikan dalam bentuk prosedur. Larik yang digunakan adalah larik yang bernama A dan bertipe *integer*. Kita asumsikan ukuran maksimum elemen larik adalah 100, dan ukuran efektif larik yang digunakan kita simpan di dalam peubah n. Selain itu, skema yang digunakan menggunakan struktur *FOR*, dan pada masalah tertentu menggunakan struktur *WHILE*.

```
DEKLARASI
  const Nmaks = 100    { ukuran maksimum larik }
  type LarikInt : array[1..Nmaks] of integer
  A             : LarikInt
  n             : integer      { mencatat ukuran larik yang digunakan }
```

Algoritma 12.7 Pendeklarasian beberapa tipe dan peubah yang akan digunakan.

12.4.2 Menginisialisasi Larik

Menginisialisasi larik adalah memberikan nilai awal untuk seluruh elemen larik atau mungkin sebagian saja. Inisialisasi larik mungkin diperlukan, misalnya "mengosongkan" elemen larik sebelum dipakai untuk proses tertentu, tetapi hal ini bukan keharusan, bergantung pada permasalahan yang akan dipecahkan. "Mengosongkan" larik bertipe numerik dapat dilakukan dengan mengisi seluruh elemen dengan nol (atau nilai lainnya, bergantung kebutuhan), sedangkan pada larik karakter, "mengosongkan" larik berarti mengisi elemen larik dengan spasi atau karakter kosong (*null*).

a. Menginisialisasi elemen-elemen larik dengan nilai 0

```
procedure InisDengan0(output A : LarikInt, input n : integer)
{ Menginisialisasi setiap elemen larik A[1..n] dengan nol. }
{ K.Awal: n adalah jumlah elemen efektif larik, nilainya terdefinisi. }
{ K.Akhir: seluruh elemen larik A bernilai nol. }

DEKLARASI
   i : integer   { pencatat indeks larik }

ALGORITMA:
   for i ← 1 to n do
      A[i] ← 0
   endfor
```

Algoritma 12.8 Menginisialisasi elemen-elemen larik A dengan 0.

Catatan:
untuk mempersingkat penulisan, maka larik A yang berukuran n elemen dan diacu dengan indeks dari 1 sampai n kita tulis sebagai A[1..n].

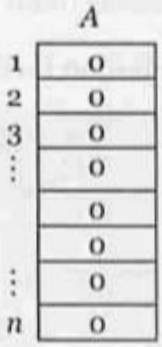
Contoh program yang memanggil `InisDengan0`:

```
PROGRAM PemroseaanLarik
{ Program untuk mengisi elemen larik dengan nilai 0 }

DEKLARASI
   const Nmaks = 100  { ukuran maksimum larik }
   type LarikInt  : array[1..Nmaks] of integer

   A  : LarikInt
   i  : integer   { indeks larik }
```

```
   n  : integer   { ukuran efektif larik }

   procedure InisDengan0(output A : LarikInt, input n : integer)
   { Menginisialisasi setiap elemen larik A[1..n] dengan nol. }

ALGORITMA:
   read(n)          { tentukan jumlah elemen larik yang akan digunakan,
                      dengan syarat 1 ≤ n ≤ Nmaks }
   InisDengan0(A, n)

   { cetak hasil inisialisasi }
   for i ← 1 to n do
     write(A[i])
   endfor
```

Algoritma 12.9 Program pemanggi inisialisasi larik.

Hasil inisialisasi larik *A* dengan 0 diperlihatkan pada Gambar 12.3. Setiap elemen larik berisi nilai yang sama, yaitu 0.



Gambar 12.3 Larik A yang setiap elemennya sudah diinisialisasi dengan 0

b. Menginisialisasi setiap elemen larik ke-*i* dengan nilai *i*.
Misalkan kita ingin setiap elemen larik yang ke-*i* diinisialisasi dengan nilai *i*. Jadi, $A[1] = 1$, $A[2] = 2$, ..., $A[n] = n$. Algoritma inisialisasinya adalah seperti berikut ini:

```
procedure InisDengan(output A : LarikInt, input n : integer)
{ Menginisialisasi setiap elemen A[i] dengan nilai i = 1, 2, ..., n. }
{ K.Awal: n adalah jumlah elemen efektif larik, nilainya terdefinisi. }
{ K.Akhir: A[1] = 1, A[2] = 2, ..., A[n] = n }

DEKLARASI
   i : integer { pencatat indeks larik }

ALGORITMA:
   for i ← 1 to n do
     A[i] ← i
   Endfor
```

Algoritma 12.10 Menginisialisasi elemen larik a dengan nilai 1 sampai n.

Hasil inisialisasi larik *A* dengan nilai 1 sampai *n* diperlihatkan pada Gambar 12.3. Setiap elemen *A[i]* berisi nilai *i*.

Gambar 12.4 Larik A yang setiap elemen ke-i sudah diinisialisasi dengan nilai i.

12.4.3 Mengisi Elemen Larik dengan Pembacaan

Selain dengan pengisian nilai secara langsung (seperti pada inisialisasi), elemen larik diisi dengan cara pembacaan (misalnya dengan mengetikkan nilainya dari papan ketik). Dalam hal ini, elemen larik dibaca satu per satu mulai untuk elemen pertama sampai elemen ke-*n*.

Algoritma pembacaan elemen larik adalah seperti di bawah ini (kita buat dalam 3 versi sesuai spesifikasinya):

Versi 1: jika jumlah elemen efektif ditentukan di awal

```
procedure BacaLarik1(output A : LarikInt, input n : integer)
{ Mengisi elemen-elemen larik A[1..n] dengan pembacaan. }
{ K.Awal: n adalah jumlah elemen efektif larik, nilainya terdefinisi. }
{ K.Akhir: setelah pembacaan, seluruh elemen larik A berisi nilai-
  nilai yang dibaca dari piranti masukan. }

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   for i ← 1 to n do
      read(A[i])
   endfor
```

Algoritma 12.11 Mengisi elemen-elemen larik dengan cara pembacaan (versi 1).

Versi 2: jika jumlah elemen efektif baru diketahui di akhir pembacaan

Adakalanya jumlah elemen efektif tidak ditentukan di awal, tetapi baru diketahui pada akhir pembacaan. Dalam hal ini, jumlah elemen efektif, *n*, merupakan parameter bertipe keluaran. Prosedur BacaLarik2 di bawah ini

membaca elemen-elemen larik, setiap kali selesai pembacaan satu buah elemen, pengguna program dikonfirmasi apakah masih ada lagi elemen larik yang akan dimasukkan.

```
write('Lagi?(y/t)')
read(jawab)
```

Jika jawabannya 'y' maka pembacaan dilanjutkan, jika 't' maka proses pembacaan dihentikan. Jumlah elemen yang telah dibaca dicatat di dalam peubah *n*.

```
procedure BacaLarik2(output A : LarikInt, output n : integer )
{ Mengisi elemen-elemen larik A[1..n] dengan cara pembacaan. }
{ K.Awal: sembarang. }
{ K.Akhir: sebanyak n buah elemen larik A berisi nilai-nilai yang
  dibaca; n berisi jumlah elemen larik yang diisi. }

DEKLARASI
   jawab : char

ALGORITMA:
   N ← 0
   repeat
     n ← n + 1
     read(A[n])
     write('Lagi?(y/t)')
     read(jawab)
   until jawab = 't'
```

Algoritma 12.12 Mengisi elemen-elemen larik dengan cara pembacaan (versi 2).

Versi 3: jika jumlah elemen efektif baru diketahui di akhir pembacaan (variasi dari versi 3)

Pada versi 3 ini, proses pembacaan dianggap selesai jika nilai yang dibaca adalah suatu tanda, misalnya 9999.

```
procedure BacaLarik3(output A : LarikInt, output n : integer )
{ Mengisi elemen-elemen larik A[1..n] dengan cara pembacaan. Akhir
  pembacaan ditandai jika nilai yang dibaca adalah 9999 }
{ K.Awal: sembarang. }
{ K.Akhir: sebanyak n buah elemen larik A berisi nilai-nilai yang
  dibaca; n berisi jumlah elemen larik yang diisi. }

DEKLARASI
   x   : integer   { menyimpan sementara nilai yang dibaca }

ALGORITMA:
   n ← 0
   read(x)
   while x ≠ 9999 do
     n ← n + 1
     A[n] ← x
     read(x)
   endwhile
   { x = 9999 }
```

Algoritma 12.13 Mengisi elemen-elemen larik dengan cara pembacaan (versi 3).

12.4.4 Mencetak Elemen-elemen Larik

Isi elemen larik dicetak ke piranti dengan pernyataan `write`. Dalam hal ini, elemen larik dicetak satu per satu mulai untuk elemen pertama sampai elemen ke-*n*.

Algoritma pencetakan elemen-elemen larik dinyatakan oleh prosedur di bawah ini.

```
procedure CetakLarik(input A : LarikInt, input n : integer )
{ Mencetak elemen-elemen larik A[1..n]. }
{ K.Awal: n sudah berisi jumlah elemen larik yang dipakai. Elemen-
elemen larik A sudah terdefinisi.}
{ K.Akhir: elemen-elemen larik A tercetak. }

DEKLARASI
   i : integer  { pencatat indeks larik }

ALGORITMA:
   for i ← 1 to n do
      write(A[i])
   endfor
```

Algoritma 12.14 Mencetak elemen-elemen larik.

12.4.5 Menghitung Nilai Rata-rata

Karena data yang hendak kita proses sudah disimpan di dalam larik, maka kita dapat memanipulasi data tersebut. Misalkan larik *A* sudah terdefinisi elemen-elemennya. Kita ingin menghitung nilai rata-rata seluruh elemen. Algoritmanya dinyatakan seperti di bawah ini:

```
Procedure HitungRataRata(input A : LarikInt, input n : integer,
                         output u : real)
{ Menghitung nilai rata-rata elemen larik A. }
{ K.Awal: elemen-elemen larik A sudah terdefinisi; n adalah jumlah
elemen larik, nilainya sudah terdefinisi. }
{ K.Akhir: u berisi nilai rata-rata seluruh elemen. }

DEKLARASI
   i      : integer   {indeks larik }
   jumlah : real      {jumlah total seluruh nilai }

ALGORITMA:
   i ← 1          { dimulai dari elemen pertama }
   jumlah ← 0     { jumlah total nilai mula-mula }
   for i ← 1 to n do
     jumlah ← jumlah + A[i]
   endfor
   u ← jumlah/N
```

Algoritma 12.15 Menghitung nilai rata-rata elemen di dalam larik.

Contoh program yang menggunakan prosedur `HitungRataRata` sebagai berikut:

```
PROGRAM Rerata
{ Program utama untuk menghitung nilai rata-rata seluruh elemen larik. }

DEKLARASI
   const Nmaks = 100      { ukuran maksimum larik }
   type LarikInt  : array[1..Nmaks] of integer

   A : LarikInt
   n : integer       { jumlah elemen larik yang digunakan }
   u : integer       { nilai rata-rata }

   procedure BacaLarik1(output A : LarikInt, input n : integer )
   { Mengisi elemen larik A[1..n] dengan pembacaan. }

   procedure HitungRataRata(input A : LarikInt, input n : integer,
                            output u : real)
   { Menghitung nilai rata-rata larik A. }

ALGORITMA:
   read(N)     { tentukan jumlah elemen larik yang akan digunakan }
   BacaLarik1(A, n)          { baca elemen-elemen larik }
   HitungRataRata(A, n, u)   { hitung nilai rata-rata }
   write(u)                  { cetak nilai rata-rata }
```

Algoritma 12.16 Program pemanggil prosedur hitung nilai rata-rata.

12.4.6 Kapan Menggunakan Larik?

Pertanyaan yang sering diajkan oleh orang yang baru belajar pemrograman adalah: kapan menggunakan larik? Larik digunakan bila kita mempunyai sejumlah data yang bertipe sama, dan kita perlu perlu menyimpan sementara data tersebut, untuk selanjutnya data tersebut kita proses. Selama pelaksanaan program, larik tetap menyimpan nilai. Hal ini bermanfaat bila kita ingin menggunakan nilai-nilai di dalam larik tersebut untuk diproses lebih lanjut di bagian lain di dalam algoritma.

Dengan menggunakan larik, kita dapat menghindari penggunaan nama-nama peubah yang banyak. Sebagai ilustrasi, program `HitungRerata_TanpaLarik` di bawah ini memerlukan 6 buah peubah (x1, x2, ..., x6) untuk menyimpan nilai-nilai *x* yang bertipe sama, *integer*. Bagaimana kalau program kita mengolah 1000 buah data yang bertipe sama? Tentu menggunakan 1000 buah peubah. Jelas cara ini tidak praktis. Selain itu, akan ada 1000 buah pernyataan `input` atau `output` untuk membaca/menuliskan nilai-nilai *x* tersebut, dan ekspresi yang panjangnya 1000 peubah untuk menghitung nilai rata-rata.

```
PROGRAM HitungRerata_TanpaLarik
{ Program yang membaca 6 buah data, mencetaknya, dan menghitung rata-
  ratanya. Tanpa menggunakan larik }
```

```
DEKLARASI
  x1, x2, x3, x4, x5, x6 : integer
  u : real

ALGORITMA:
  { Baca 6 buah nilai integer, simpan di x1, x2, ..., x6 }
  read(x1)
  read(x2)
  read(x3)
  read(x4)
  read(x5)
  read(x6)

  { Cetak nilai x1, x2, ..., x6 }
  write(x1)
  write(x2)
  write(x3)
  write(x4)
  write(x5)
  write(x6)

  { hitung nilai rata-rata }
  u ← (x1 + x2 + x3 + x4 + x5 + x6)/6
  write(u)
```

Algoritma 12.17 Program menghitung nilai rata-rata, tanpa menggunakan larik.

Dengan menggunakan larik, menyimpan 6 buah nilai x cukup memerlukan hanya satu peubah larik saja (x), yang dalam hal ini x terdiri atas 6 buah elemen. Untuk menyimpan 1000 buah elemen x juga membutuhkan satu buah peubah larik yang terdiri atas 1000 elemen. Selain itu, dengan larik, instruksi pembacaan/penulisan seluruh elemen larik cukup ditulis satu kali saja di dalam sebuah konstruksi pengulangan, seperti Algoritma 12.18 berikut ini:

```
PROGRAM HitungRerata_DenganLarik
{ Program yang membaca 6 buah data, mencetaknya, dan menghitung rata-
  ratanya. Menggunakan larik }

DEKLARASI
  x : array[1..6] of integer
  i : integer
  jumlah, u : real

ALGORITMA:
  { Baca 6 buah nilai integer, simpan di x[i], i = 1, 2, ...,6 }
  for i ← 1 to 6 do
    read(x[i])
  endfor

  { Cetak setiap nilai x[i], i = 1, 2, ..., 6 }
  for i ← 1 to 6 do
    write(x[I])
  endfor
```

```
{ hitung nilai rata-rata }
jumlah ← 0
for i ← 1 to 6 do
  jumlah ← jumlah + x[i]
endfor
u ← jumlah/6
write(u)
```

Algoritma 12.18 Program menghitung nilai rata-rata, menggunakan larik.

12.4.7 Mencari Nilai Maksimum Larik

Nilai maksimum pada larik *integer/real* adalah elemen larik yang mempunyai nilai terbesar di antara elemen larik lainnya. Pada Gambar 12.2, elemen maksimum adalah 172.

Persoalan mencari nilai maksimum sering ditemukan pada beberapa masalah, misalnya juara pertama dari sekumpulan skor sebuah pertandingan, mencari juara kelas, dan sebagainya.

Bila elemen larik sudah terurut menaik, harga maksimum langsung diperoleh pada elemen terakhir. Sebaliknya bila elemen larik sudah terurut menurun, harga maksimum langsung diperoleh pada elemen pertamanya.

Bila elemen larik tersusun acak, kita harus mengunjungi seluruh elemen larik untuk mencari nilai maksimum. Terdapat tiga versi algoritma pencarian harga maksimum, masing-masing kita sebut algoritma versi 1, versi 2, dan versi 3.

(a) Versi 1

Algoritma versi 1 ini kita ambil nilai maksimum sementara (*maks*) adalah nilai yang sangat kecil, misalnya -9999. Selanjutnya, larik dikunjungi mulai dari elemen pertama. Setiap kali mengunjungi elemen larik, bandingkan elemen tersebut dengan nilai maksimum sementara. Jika elemen larik yang sedang ditinjau lebih besar dari nilai maksimum sementara, maka *maks* diganti dengan elemen tersebut. Pada akhir kunjungan (yaitu setelah seluruh elemen larik dikunjungi), nilai maksimum sementara menjadi nilai maksimum yang sebenarnya.

Sebagai ilustrasi, tinjau kembali contoh larik pada Gambar 12.2 (di bawah ini diperagakan kembali):

	A
1	158
2	157
3	162
4	169
5	172
6	155
7	170
8	163

Asumsi: *maks* = -9999 (nilai maksimum sementara)
A[1] > *maks*? Ya → *maks* ← *A*[1] (*maks* = 158)
A[2] > *maks*? Tidak → (*maks* tidak berubah)
A[3] > *maks*? Ya → *maks* ← *A*[3] (*maks* = 162)
A[4] > *maks*? Ya → *maks* ← *A*[4] (*maks* = 169)
A[5] > *maks*? Ya → *maks* ← *A*[5] (*maks* = 172)
A[6] > *maks*? Tidak → (*maks* tidak berubah)
A[7] > *maks*? Tidak → (*maks* tidak berubah)
A[8] > *maks*? Tidak → (*maks* tidak berubah)
(proses pembandingan selesai)

maks = 172 (nilai maksimum larik)

Algoritma mencai nilai maksimum adalah seperti di bawah ini:

(1) sebagai prosedur:

```
procedure CariMaks1(input A : LarikInt, input n : integer,
                    output maks : integer)
{ Mencari elemen terbesar di dalam larik A[1..n]. }
{ K.Awal: n sudah berisi ukuran efektif larik; seluruh elemen larik
  A[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: maks berisi elemen larik yang bernilai maksimum }

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   maks ← -9999     { nilai maksimum sementara }
   for i ← 1 to N do
     if A[i] > maks then
        maks ← A[i]
     endif
   endfor
```

Algoritma 12.19 Mencari elemen maksimum larik (versi 1).

(2) sebagai fungsi:

```
function Maks1(input A : LarikInt, input n : integer)→integer
{ Mengembalikan elemen terbesar dari larik A[1..n]. }

DEKLARASI
   i    : integer   { pencatat indeks larik }
   maks : integer

ALGORITMA:
   maks ← -9999     { nilai maksimum sementara }
   for i ← 1 to N do
     if A[i] > maks then
        maks ← A[i]
     endif
   endfor
   return maks
```

Algoritma 12.20 Mencari elemen maksimum larik (versi 1, dalam bentuk fungsi).

Algoritma versi 1 ini hanya benar jika seluruh elemen larik bernilai lebih besar dari -9999 (sesuai dengan inisialisasi yang diberikan), karena nilai *maks* = -9999 selalu tergantikan dengan nilai elemen pertama larik. Bila ada elemen larik yang bernilai kurang dari -9999 (misalnya -12000), maka ada kemungkinan -9999 dapat menjadi elemen maksimum larik. Bila demikian, maka nilai maksimum yang diperoleh bisa salah.

(b) Versi 2

Pada algoritma versi 2 ini, nilai maksimum sementara diinisialisasi dengan elemen pertama larik. Selanjutnya, larik dikunjungi mulai dari elemen kedua. Setiap kali mengunjungi elemen larik, bandingkan elemen tersebut dengan nilai maksimum sementara. Jika elemen larik yang sedang ditinjau lebih besar dari nilai maksimum sementara, maka *maks* diganti dengan elemen tersebut. Pada akhir kunjungan (yaitu setelah seluruh elemen larik dikunjungi), nilai maksimum sementara menjadi nilai maksimum yang sebenarnya.

Sebagai ilustrasi, tinjau kembali contoh larik pada Gambar 12.2 (di bawah ini diperagakan kembali):

	A
1	158
2	157
3	162
4	169
5	172
6	155
7	170
8	163

Asumsi: *maks* = *A*[1] (nilai maksimum sementara, maks = 158)

A[2] > *maks*? Tidak → (*maks* tidak berubah)
A[3] > *maks*? Ya → *maks* ← *A*[3] (*maks* = 162)
A[4] > *maks*? Ya → *maks* ← *A*[4] (*maks* = 169)
A[5] > *maks*? Ya → *maks* ← *A*[5] (*maks* = 172)
A[6] > *maks*? Tidak → (*maks* tidak berubah)
A[7] > *maks*? Tidak → (*maks* tidak berubah)
A[8] > *maks*? Tidak → (*maks* tidak berubah)
(proses pembandingan selesai)

maks = 172 (nilai maksimum larik)

Algoritma mencai nilai maksimum, seperti di bawah ini:

```
procedure CariMaks2(input A : LarikInt, input n : integer,
                    output maks : integer)
{ Mencari elemen terbesar di dalam larik A[1..n]. }
{ K.Awal: n sudah berisi ukuran efektif larik; seluruh elemen larik
  A[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: maks berisi elemen larik yang bernilai maksimum }

DEKLARASI
   i : integer { pencatat indeks larik }

ALGORITMA:
   maks ← A[1]     { nilai maksimum sementara }
   for i ← 2 to n do
     if A[i] > maks then
        maks ← A[i]
     endif
   endfor
```

Algoritma 12.21 Mencari elemen maksimum larik (versi 2).

Algoritma versi 2 juga dapat dinyatakan sebagai fungsi, seperti di bawah ini:

```
function Maks2(input A : LarikInt, input n : integer)→integer
{ Mengembalikan elemen terbesar dari larik A[1..n]. }

DEKLARASI
   i : integer { pencatat indeks larik }

ALGORITMA:
   maks←A[1]     { nilai maksimum sementara }
   for i←2 to n do
     if A[i] > maks then
        maks←A[i]
     endif
   endfor
   return maks
```

Algoritma 12.22 Mencari elemen maksimum larik (versi 2, dalam bentuk fungsi).

Algoritma versi 2 lebih umum dan akan selalu benar untuk nilai larik apa pun (baik bernilai positif ataupun negatif). Selain itu, algoritma tetap benar meskipun larik cuma berukuran 1 elemen (yaitu, $n = 1$).

(c) Versi 3

Sering di dalam program kita tidak membutuhkan nilai maksimum larik, namun yang diperlukan adalah posisi atau indeks elemen larik yang bernilai maksimum itu. Pada contoh ilustrasi di atas, nilai maksimum (`maks`) adalah 172, terletak pada indeks nomor 5. Bila informasi ini yang diinginkan, maka algoritmanya adalah seperti di bawah ini (indeks elemen larik yang bernilai maksimum disimpan di dalam peubah `IdxMaks`):

```
procedure CariMaks3(input A : LarikInt, input n : integer,
                    output IdxMaks : integer)
{ Mencari indeks elemen larik bernilai terbesar dari larik A[1..n] }
{ K.Awal: n sudah berisi ukuran efektif larik; seluruh elemen larik
  A[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: IdxMaks berisi indeks eleman larik yang bernilai maksimum. }

DEKLARASI
   i : integer     { pencatat indeks larik }
ALGORITMA:
   IdxMaks ← 1     { Indeks larik yang berisi nilai minimum sementara }
   for i ← 2 to n do
     if A[i] > A[IdxMaks] then
        IdxMaks ← i
     endif
   endfor
```

Algoritma 12.23 Mencari elemen maksimum larik (versi 3).

Contoh program yang memanggil prosedur `CariMaks3`, seperti di bawah ini:

```
PROGRAM Maksimum
{ Program untuk mencari elemen maksimum dari sebuah larik. }

DEKLARASI
   const Nmaks = 100  { ukuran maksimum larik }
   type LarikInt  : array[1..Nmaks] of integer

   A : LarikInt
   n : integer       { ukuran efektif larik }
   imaks : integer   { indeks larik yang elemennya terbesar }

   procedure BacaLarik1(output A : LarikInt, input n : integer )
   { Membaca elemen-elemen larik A }

   procedure CariMaks3(input A : LarikInt, input n : integer,
                       output IdxMaks : integer)
   { Mencari indeks elemen larik bernilai terbesar dari larik A[1..n] } }
```

```
ALGORITMA:
   read(n)                  { tentukan ukuran larik }
   BacaLarik1(A,n)          { baca elemen-elemen larik }
   CariMaks3(A,n,imaks)     { cari indeks elemen yang bernilai terbesar }
     write('Elemen terbesar : ', A[imaks])
```

Algoritma 12.24 Program pemanggil CariMaks3 .

Prosedur `CariMaks3` memberikan indeks dari kemunculan pertama elemen yang bernilai terbesar. Jika terdapat lebih dari satu elemen maksimum, maka hanya indeks elemen terbesar yang pertama kali ditemukan yang dihasilkan. Hal ini karena kita menggunakan tanda ">" dalam operasi perbandingan:

```
if A[i] > A[IdxMaks] then
   IdxMaks ← i
endif
```

Jika kita menganti tanda ">" dengan "≥", maka kita akan memperoleh indeks elemen terbesar yang terakhir kali ditemukan:

```
if A[i] ≥ A[IdxMaks] then
   IdxMaks ← i
endif
```

Dapatkah Anda mengerti mengapa demikian?

Prosedur mencari indeks larik yang elemennya terbesar nanti akan bermanfaat pada waktu kita membicarakan algoritma pengurutan.

12.4.8 Mencari Nilai Minimum Larik

Algoritma untuk mencari nilai minimum larik analog dengan algoritma mencari nilai maksimum larik.

Versi 1: Nilai minimum (*min*) awal = 9999

```
procedure CariMin1(input A : LarikInt, input n : integer,
                   output min : integer)
{ Mencari elemen terkecil di dalam larik A[1..n]. }
{ K.Awal: n sudah berisi ukuran efektif larik; seluruh elemen larik
  A[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: min berisi elemen larik yang bernilai terkecil }

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   min ← 9999    { nilai minimum sementara }
   for i←1 to n do
     if A[i] < min then
```

```
        min ← A[i]
      endif
    endfor
```

Algoritma 12.25 Mencari elemen minimum larik (versi 1).

Versi 2: Nilai minimum (min) awal = elemen pertama larik = A[1]

```
procedure CariMin2(input A : LarikInt, input n : integer,
                   output min : integer)
{ Mencari elemen terkecil di dalam larik A[1..n]. }
{ K.Awal: n sudah berisi ukuran efektif larik; seluruh elemen larik
  A[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: min berisi elemen larik yang bernilai terkecil }

DEKLARASI
   i : integer { pencatat indeks larik }

ALGORITMA:
   min ← A[1]     { nilai minimum sementara }
   for i ← 2 to n do
     if A[i] < min then
        min ← A[i]
     endif
   endfor
```

Algoritma 12.26 Mencari elemen minimum larik (versi 2).

Versi 3: Mencari indeks elemen larik yang bernilai minimum

```
procedure CariMin3(input A : LarikInt, input N : integer,
                   output IdxMin : integer)
{ Mencari indeks elemen larik bernilai terkecil dari larik A[1..n] }
{ K.Awal: n sudah berisi ukuran efektif larik; seluruh elemen larik
  A[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: IdxMin berisi indeks elemen larik yang bernilai terkecil. }

DEKLARASI
   i : integer { pencatat indeks larik }

ALGORITMA:
   IdxMin ← 1   { Indeks larik yang berisi nilai minimum sementara }
   for i ← 2 to n do
     if A[i] < A[IdxMin] then
        IdxMin ← i
     endif
   endfor
```

Algoritma 12.27 Mencari elemen minimum larik (versi 3).

Pencarian nilai maksimum dan minimum dapat dilakukan sekaligus dalam sekali 'jalan' seperti yang ditunjukkan pada algoritma berikut:

```
procedure MinMaks(input A : LarikInt, input N : integer,
                  output min : integer, output maks : integer)
{ Mencari elemen terkecil dan elemen terbesar di dalam larik A[1..N]. }
{ K.Awal: elemen-elemen larik A[1..N] sudah terdefinisi. }
{ K.Akhir: min berisi elemen larik yang bernilai terkecil,
           maks berisi elemen larik yang bernilai terbesar }

DEKLARASI
   I : integer    { pencatat indeks larik }

ALGORITMA:
   min ← A[1]
   maks ← A[1]
   for i ← 2 to N do
      if A[i] < min then
         min ← A[i]
      endif
      if A[i] > maks then
         maks ← A[i]
      endif
   endfor
```

Algoritma 12.28 Mencari sekaligus elemen minimum dan maksimum larik.

Prosedur `MinMaks` dapat Anda modifikasi jika yang ingin dihasihkan adalah indeks larik yang bernilai minimum dan maksimum (tinjau prosedur `CariMaks3`).

12.4.9 Menyalin Larik

Diberikan sebuah larik *integer*, *A*, yang sudah terdefinisi elemen-elemennya. Ukuran larik adalah *n* elemen. Kita ingin menyalin (*copy*) elemen-elemen larik *A* ke dalam larik yang lain, *B*.

Algoritma menyalin larik sangat sederhana. Inilah algoritmanya:

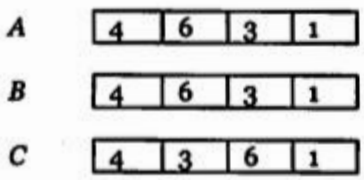
```
procedure SalinLarik(input A : LarikInt, input n : integer,
                     output B : LarikInt)
{ Menyalin (copy) larik A[1..n] ke dalam larik B[1..n] }
{ K.Awal: Larik A[1..n] sudah terdefinisi.}
{ K.Akhir: B[1..n] adalah larik yang elemen-elemennya sama dengan
larik A[1..n]. }

DEKLARASI
   i : integer          { indeks larik }

ALGORITMA:
    for i ← 1 to n do
       B[i] ← A[i]
    endfor
```

Algoritma 12.29 Menyalin (*copy*) larik A ke dalam larik.

12.4.10 Menguji Kesamaan Dua Buah Larik

Diberikan dua buah larik *integer*, *A* dan *B*, yang jumlah elemennya sama (*n* elemen). Kita akan membuat fungsi untuk memeriksa apakah kedua larik tersebut sama. Dua buah larik dikatakan sama bila elemen-elemen yang berpadanan urutannya sama. Fungsi tersebut mengembalikan nilai *true* jika *A* dan *B* sama, sebaliknya mengembalikan *false* jika tidak sama.

Sebagai contoh, diberikan tiga buah larik di bawah ini, *A*, *B*, dan *C*:



Larik *A* dan *B* sama, tetapi *A* tidak sama dengan *C*, karena $A[2] \neq C[2]$.

Pemeriksaan kesamaan dua buah larik dilakukan dengan membandingkan setiap elemen yang berpadanan pada larik. Jika ditemukan satu saja elemen yang tidak sama, maka pemeriksaan dihentikan dan fungsi mengembalikan nilai *false*. Inilah algoritmanya:

```
function LarikSama(input A, B : LarikInt, input n : integer) → boolean
{ memberikan nilai true jika larik A[1..n] dan B[1..n] sama, atau
false
  jika tidak sama. }

DEKLARASI
   i    : integer
   sama : boolean

ALGORITMA:
  i ← 1
  sama ← true
  while (i ≤ n) and (sama) do
    if A[i] = B[i] then
      i ← i + 1    { tinjau elemen berikutnya }
    else    { A[i] ≠ B[i], maka dapat disimpulkan A dan B tidak sama }
      sama ← false
    endif
  endwhile
  { i > n or not sama}

  return sama
```

Algoritma 12.30 Memeriksa kesamaan dua buah larik yang berukuran sama.

12.5 Larik Bertipe Terstruktur

Contoh-contoh algoritma yang dibahas sebelum ini menggunakan larik dengan elemen-elemen bertipe sederhana. Elemen larik juga dapat bertipe terstruktur.

Sebagai contoh, misalkan kita akan mengolah data 100 orang mahasiswa. Data setiap mahasiswa terdiri dari NIM (Nomor Induk Mahasiswa), nama mahasiswa, dan IPK (indeks prestasi kumulatif yang nilainya berkisar antara 0 sampai 4). Struktur lojik larik *Mhs* ditunjukkan pada Gambar 12.4.

Struktur lojik larik *Mhs* ditunjukkan pada Gambar 12.4.

	NIM	*NamaMhs*	*IPK*
1	29801	Heru Satrio	3.04
2	29804	Amirullah Satya	2.75
3			
4			
5			
6			
7			
.			
.			
100	29887	Yanti Siregar	2.19

Gambar 12.5 Larik Mhs dengan 100 elemen. Setiap elemen larik bertipe terstruktur (*record*). Tiap *record* terdiri atas *field* NIM, NamaMhs, dan *field* IPK.

Struktur larik yang menyimpan keseluruhan data tersebut dideklarasikan di bawah ini. Nama larik adalah *Mhs*.

```
DEKLARASI
  const Nmaks = 100
  type Mahasiswa : record <NIM   : integer,  { Nomor Induk Mahasiswa }
                          NamaMhs : string,   { nama mahasiswa }
                          IPK    : real       { 0.00 sampai 4.00 }
                         >

  type TabMhs : array[1..Nmaks] of Mahasiswa
  Mhs   : TabMhs
```

Algoritma 12.31 Deklarasi larik Mhs yang bertipe terstruktur.

Cara mengacu elemen Mhs:

```
Mhs[2]          { elemen kedua dari larik Mhs }
Mhs[2].NIM      { mengacu field NIM dari elemen kedua larik }
Mhs[2].IPK      { mengacu field IPK dari elemen kedua larik }
```

Karena *record* merupakan struktur, maka kita tidak dapat melakukan pencetakan elemen larik *Mhs* seperti di bawah ini:

```
write(Mhs[i])
```

tetapi haruslah seperti berikut ini:

```
write(Mhs[i].NIM, Mhs[i].NamaMhs, Mhs[i].IPK)
```

Namun, cara mengisi nilai elemen ke elemen lainnya dengan penulisan pernyataan seperti di bawah ini adalah benar:

```
Mhs[i] ← Mhs[i+1]
```

yang sama saja dengan pengisian per *field* sebagai berikut:

```
Mhs[i].NIM ← Mhs[i+1].NIM
Mhs[i].NamaMhs ← Mhs[i+1].NamaMhs
Mhs[i].KodeMK ← Mhs[i+1].IPK
```

Algoritma untuk mengisi larik `Mhs` adalah seperti di bawah ini:

```
procedure BacaDataMahasiswa(input n : integer, output Mhs : TabMhs)
{ Membaca data mahasiswa (NIM, nama, IPK).
{ K.Awal  : n berisi jumlah data mahasiswa }
{ K.Akhir : Mhs berisi data hasil pembacaan }

DEKLARASI
   i : integer      { pencatat indeks larik }

ALGORITMA:
   for i←1 to N do
      read(Mhs[i].NIM)
      read(Mhs[i].NamaMhs)
      read(Mhs[i].IPK)
   endfor
```

Algoritma 12.32 Membaca data n orang mahasiswa (nama, NIM, IPK).

Selain bertipe terstruktur, elemen larik juga dapat bertipe larik lain. Contoh berikut menyajikan struktur tipe bentukan yang cukup kompleks. Misalkan kita ingin mengolah data nilai semua mata kuliah yang diambil setiap mahasiswa di semester genap. Asumsikan setiap mahasiswa mengambil 4 buah mata kuliah yang berbeda. Setiap elemen larik berisi data sebagai berikut:

1. *NIM* (Nomor Induk mahasiswa)
2. *NamaMhs* (nama mahasiswa)
3. Mata kuliah (*MK*) yang diambil mahasiswa tersebut (4 buah), berupa larik:
   a. Kode mata kuliah ke-1
      Nama mata kuliah ke-1
      Nilai mata kuliah ke-1

b. Kode mata kuliah ke-2
   Nama mata kuliah ke-2
   Nilai mata kuliah ke-2
c. Kode mata kuliah ke-3
   Nama mata kuliah ke-3
   Nilai mata kuliah ke-3
d. Kode mata kuliah ke-4
   Nama mata kuliah ke-4
   Nilai mata kuliah ke-4

Struktur larik yang menyimpan keseluruhan data tersebut dideklarasikan di bawah ini. Nama larik adalah *Mhs2*.

```
DEKLARASI
  const Nmaks = 100
  type MataKuliah : record <KodeMK : string, { kode mata kuliah }
                            Nilai  : char   { indeks nilai: A/B/C/D/E)}
                           >

  type Mahasiswa  : record <NIM    : integer,    { Nomor Induk Mahasiswa }
                            NamaMhs : string,   { nama mahasiswa }
                            MK     : array[1..4] of MataKuliah
                           >
  type TabMhs : array[1..Nmaks] of Mahasiswa
  Mhs2    : TabMhs
```

Algoritma 12.33 Deklarasi larik Mhs2 yang elemennya bertipe larik.

Cara mengacu elemen `Mhs2`:

```
Mhs2[2]                 { elemen kedua dari larik Mhs2 }
Mhs2[2].NIM             { mengacu field NIM dari elemen kedua larik }
Mhs2[2].MK[3].KodeMK    { mengacu field KodeMK ketiga dari elemen kedua
                          larik Mhs2 }
Mhs2[2].MK[3].Nilai     { mengacu field Nilai ketiga dari elemen kedua
                          larik Mhs2 }
```

Contoh algoritma untuk mengisi larik `Mhs2`:

```
procedure BacaDataMahasiswa(input n : integer, output Mhs2 : TabMhs)

{ Membaca data mahasiswa (NIM, nama, daftar mata kuliah, nilai tiap
mata kuliah }
{ K.Awal  : n berisi jumlah data mahasiswa }
{ K.Akhir : Mhs2 berisi data hasil pembacaan }

DEKLARASI
  i,j : integer    { pencatat indeks larik }

ALGORITMA:
  for i ← 1 to n do
      read(Mhs2[i].NIM)
      read(Mhs2[i].NamaMhs)
      for j ←1 to 4 do
```

```
        read(Mhs2[i].MK[j].KodeMK)
        read(Mhs2[i].MK[j].Nilai)
    endfor
endfor
```

Algoritma 12.34 Membaca data n orang mahasiswa (nama, NIM, daftar mata kuliah, nilai setiap mata kuliah).

12.6 Bekerja dengan Dua Buah Larik

Dua buah larik atau lebih dapat dioperasikan sekaligus. Perhatikan contoh berikut ini: misalkan nilai ujian *n* orang mahasiswa telah disimpan di dalam larik *NilUjian*. Kita akan menghitung nilai indeks (*A*/*B*/*C*/*D*/*E*) mahasiswa tersebut dan menyimpan nilai indeks tersebut di dalam larik *Indeks*. Indeks nilai ujian ditentukan sebagai berikut:

```
NilUjian[k]  ≥ 80              , indeks nilai = A
70 ≤ NilUjian[k] < 80          , indeks nilai = B
55 ≤ NilUjian[k] < 70          , indeks nilai = C
45 ≤ NilUjian[k] < 55          , indeks nilai = D
NilUjian[k] < 45               , indeks nilai = E
```

Deklarasi data:

```
DEKLARASI
  const Nmaks = 200  { jumlah maksimum elemen larik }
  type LarikUjian   : array [1..Nmaks] of real
  type LarikIndeks  : array [1..Nmaks] of char
```

Algoritma 12.35 Deklarasi larik nilai ujian.

Prosedur untuk menghitung nilai indeks sebagai berikut:

```
procedure HitungIndeksNilai(input NilUjian : LarikUjian,
                            input n : integer,
                            output Indeks : LarikIndeks )

{ Menghitung indeks nilai ujian n orang mahasiswa. }
{ K.Awal: n sudah berisi jumlah mahasiswa;
          elemen larik NilUjian[1..n] sudah terdefinisi harganya. }
{ K.Akhir: larik Indeks[1..n] berisi nilai indeks ujian. }

DEKLARASI
   i : integer          { indeks larik }

DESKRIPSI:
   for k ← 1 to n do
     if NilUjian[i] ≥ 80 then
        Indeks[i]←'A'
     else
        if (NilUjian[i] ≥ 70) and (NilUjian[i] < 80) then
          Indeks[i]←'B'
```

```
        else
            if (NilaUjian[i] ≥ 55) and (NilUjian[i] < 70) then
                Indeks[i]←'C'
            else
                if (NilUjian[i] ≥ 45) and (NilUjian[i] < 55) then
                    Indeks[i]←'D'
                else
                    Indeks[i]←'E'
                endif
            endif
        endif
    endif
endfor
```

Algoritma 12.36 Menghitung indeks nilai ujian.

Selanjutnya, kita ingin mencetak tabel yang menampilkan nilai ujian beserta indeksnya seperti contoh berikut:

i	Nilai ujian	Indeks Nilai
1	76	*B*
2	87	*A*
.	.	.
.	.	.
.	.	.
n	35	*E*

Algoritma pencetakan tabel di atas adalah seperti di bawah ini:

```
procedure CetakNilai(input NilUjian : LarikUjian,
                     input Indeks   : LarikIndeks,
                     input n : integer)

{ Mencetak tabel yang berisi nilai ujian dan indek nilainya. Nomor
  peserta ujian sesuai dengan nomor indeks larik. }
{ K.Awal: NilUjian[1..n] dan Indeks[1..n] sudah terdefinisi elemen-
  elemennya. }
{ K.Akhir: nilai ujian dan nilai indeksnya tercetakn. }

DEKLARASI
  i : integer          { indeks larik }

ALGORITMA:
   { cetak header tabel }
   write('-------------------------------------------')
   write(' i   Nilai ujian            Indeks       ')
   write('-------------------------------------------')

   { cetak nilai ujian dan indeks mahasiswa ke-i }
   for i ← 1 to N do
      write(i, '          ', NilUjian[i], '       ', Indeks[i])
   endfor
```

```
   { cetak garis penutup tabel }
   write('-----------------------------------------')
```

Algoritma 12.37 Mencetak tabel nilai ujian dan indek nilainya.

12.7 Translasi Notasi Algoritmik Larik ke dalam Notasi Bahasa Pascal dan Bahasa C

Setiap bahasa pemrograman berbeda-beda dalam menyatakan awal indeks larik. Bahasa *C* memulai indeks dari 0, sedangkan dalam bahasa *Pascal* awal indeks dapat didefinisikan oleh pemrogram (boleh dimulai dari 1 , 0, atau bahkan negatif).

Di bawah ini diberikan cara mentranslasi larik dalam notasi algoritmik menjadi notasi larik dalam bahasa *Pascal* dan bahasa *C*.

ALGORITMIK

```
DEKLARASI
   NamaLarik : array[1..JumlahElemen] of TipeElemen
```

Cara mengacu elemen larik: *NamaLarik*[*indeks*]

PASCAL

```
(* DEKLARASI *)
var
   NamaLarik : array[1..JumlahElemen] of TipeElemen;
```

Cara mengacu elemen larik: *NamaLarik*[*indeks*]

C

```
/* DEKLARASI */
typedef TipeElemen NamaLarik[JumlahElemen + 1]
```

Cara mengacu elemen larik: *NamaLarik*[*indeks*]

Catatan:
Indeks larik dalam bahasa C dimulai dari 0. Algoritma-algoritma larik yang kita tulis di dalam bab ini memulai indeks dari 1. Jadi, tanpa mengubah algoritma, maka JumlahElemen pada bahasa C kita tambah dengan 1, sehingga kita tetap dapat mengindeks dari 1 sampai JumlahElemen. Elemen ke-0 boleh tidak kita gunakan.

Contoh-contoh Translasi:

1. Mendeklarasikan larik

ALGORITMIK

a. Sebagai peubah

```
DEKLARASI
  A        : array[1..100] of integer
  NamaMhs  : array[1..12] of string
  NilUjian : array[0..74] of real
```

Cara mengacu elemen larik:

```
A[4]              { megacu elemen keempat dari larik A }
NamaMhs[2]        { mengacu elemen kedua dari larik NamaMhs }
A[i]              { mengacu elemen ke-i dari larik A, asalkan
                    nilai i sudah terdefinisi }
NilUjian[i+1]     { asalkan nilai i sudah terdefinisi }
```

b. Sebagai tipe bentukan

```
DEKLARASI
  type LarikInt : array[1..100] of integer   { nama tipe baru }
  A : LarikInt  { A adalah sebuah larik integer dengan 100 elemen }
```

c. Mendeklarasikan ukuran larik sebagai sebuah konstanta

```
DEKLARASI
  const Nmaks = 100   { ukuran maksimum larik }
  type LarikInt : array[1..Nmaks] of integer

  A : LarikInt  { A adalah sebuah larik integer dengan 100 elemen }
```

d. Larik bertipe terstruktur

```
DEKLARASI
  const Nmaks = 100
  type Mahasiswa : record <NIM   : integer,   { Nomor Induk Mahasiswa }
                          NamaMhs : string,{ nama mahasiswa }
                          IPK     : real      { 0.00 sampai 4.00 }
                         >

  type TabMhs : array[1..Nmaks] of Mahasiswa
  Mhs   : TabMhs
```

PASCAL

a. Sebagai peubah

```
(* DEKLARASI *)
var
  A        : array[1..100] of integer ;
```

```
  NamaMhs  : array[1..12] of string[25]; { panjang string = 25
                                          karakter }
  NilUjian : array[0..74] of real ;
```

Cara mengacu elemen larik:

```
A[4]                { megacu elemen keempat dari larik A }
NamaMhs[2]          { mengacu elemen kedua dari larik NamaMhs }
A[i]                { mengacu elemen ke-i dari larik A, asalkan
                      nilai i sudah terdefinisi }
NilUjian[i+1]       { asalkan nilai i sudah terdefinisi }
```

b. Sebagai tipe bentukan

```
(* DEKLARASI *)
  type LarikInt = array[1..100] of integer ;  { nama tipe baru }
  var A : LarikInt ;{ A adalah sebuah larik integer dengan 100 elemen }
```

c. Mendeklarasikan ukuran larik sebagai sebuah konstanta

```
(* DEKLARASI *)
  const Nmaks = 100;    { ukuran maksimum larik }
  type LarikInt = array[1..Nmaks] of integer  ;
  var A : LarikInt ;{ A adalah sebuah larik integer dengan 100 elemen }
```

d. Larik bertipe terstruktur

```
(* DEKLARASI *)
  const Nmaks = 100;
  type Mahasiswa = record
                     NIM : integer;      { Nomor Induk Mahasiswa }
                     NamaMhs : string;   { nama mahasiswa }
                     IPK  : real;        { 0.00 sampai 4.00 }
                   end;

       TabMhs = array[1..Nmaks] of Mahasiswa;
  var Mhs    : TabMhs;
```

C

a. Sebagai peubah

```
/* DEKLARASI */
 int A[101];
 char NamaMhs[13][25];   /* panjang string = 25 karakter */
 float NilUjian[75];
```

Penjelasan:

1. `int A[101]`, berati elemen-elemen larik `A` adalah `A[0]`, `A[1]`, `A[2]`, ..., `A[100]`. Jika indeks larik dimulai dari 1 sampai 100, maka elemen `A[0]` tidak digunakan.

2. **char** NamaMhs[13][25], berati larik NamaMhs terdiri atas 13 buah elemen, setiap elemen bertipe *string* yang panjangnya 25 karakter. Elemen larik tersebut adalah NamaMhs[0], NamaMhs[1], NamaMhs[2], ..., NamaMhs[12].
3. indeks larik dalam bahasa *C* selalu bertipe *integer*.

Cara mengacu elemen larik:

```
A[4]                /* megacu elemen keempat dari larik A */
NamaMhs[2]          /* mengacu elemen kedua dari larik NamaMhs */
A[i]                /* mengacu elemen ke-i dari larik A, asalkan
                       nilai i sudah terdefinisi */
NilUjian[i+1]       /* asalkan nilai i sudah terdefinisi */
```

b. Sebagai tipe bentukan

```
/* DEKLARASI */
typedef int LarikInt[101];
LarikInt A;  /* A adalah sebuah larik integer dengan 100 elemen */
```

Kita juga dapat membuat sebuah tipe yang bernama *string*. Misalkan didefinisikan sebuah tipe *string* yang panjangnya 25 karakter.

```
/* DEKLARASI*/
typedef char String[25];
```

Maka, kita dapat mendeklarasikan larik NamaMhs pada bagian a di atas yang bertipe *String* sebagai berikut:

```
/* DEKLARASI*/
typedef char String[25];
String NamaMhs[13];
```

c. Mendeklarasikan ukuran larik sebagai sebuah konstanta

```
/* DEKLARASI */
const int Nmaks = 100;  /* ukuran maksimum larik */
typedef int LarikInt[Nmaks+1];
LarikInt A;  /* A adalah sebuah larik integer dengan 100 elemen */
```

d. Larik bertipe terstruktur

```
/* DEKLARASI */
const int Nmaks = 100;

typedef struct  { int NIM;            /* Nomor Induk Mahasiswa */
                  char NamaMhs[25];   /* nama mahasiswa */
                  float IPK;          /* 0.00 sampai 4.00 */
                } Mahasiswa;

typedef Mahasiswa TabMhs[Nmaks+1];
TabMhs Mhs;
```

2. Larik sebagai parameter prosedur/fungsi

ALGORITMIK

(a) Larik sebagai parameter masukan

```
procedure CetakLarik(input A : LarikInt, input n : integer )
{ Mencetak elemen-elemen larik A[1..n]. }
```

Cara pemanggilan: `CetakLarik(A, n)`

Purwarupa prosedur di atas ekivalen dengan penulisan sebagai berikut:

```
procedure CetakLarik(input A:array[1..100] of integer,input n :
integer)
{ Mencetak elemen-elemen larik A[1..n]. }
```

(b) Larik sebagai parameter keluaran

```
procedure BacaLarik(output A : LarikInt, input n : integer)
{ Mengisi elemen-elemen larik A[1..n] dengan pembacaan. }
```

Cara pemanggilan: `BacaLarik(A, n)`

(c) Larik sebagai parameter masukan/keluaran

```
procedure UrutLarik(input/output A : LarikInt, input n : integer)
{ Mengurut (sort) elemen-elemen larik A[1..n] sehingga tersusun
menaik.}
```

Cara pemanggilan: `UrutLarik(A, n)`

PASCAL

(a) Larik sebagai parameter masukan

```
procedure CetakLarik(A : LarikInt; n : integer);
{ Mencetak elemen-elemen larik A[1..n]. }
```

Cara pemanggilan: `CetakLarik(A,n)`

Purwarupa prosedur di atas ekivalen dengan penulisan sebagai berikut:

```
procedure CetakLarik(A : array[1..100] of integer; n : integer);
{ Mencetak elemen-elemen larik A[1..n]. }
```

(b) Larik sebagai parameter keluaran

```
procedure BacaLarik(var A : LarikInt; n : integer);
{ Mengisi elemen-elemen larik A[1..n] dengan pembacaan. }
```

Cara pemanggilan: `BacaLarik(A,n)`

(c) Larik sebagai parameter masukan/keluaran

```
procedure UrutLarik(var A : LarikInt; n : integer);
{ Mengurut (sort) elemen-elemen larik A[1..n] sehingga tersusun
menaik.}
```

Cara pemanggilan: `UrutLarik(A,n)`

C

(a) Larik sebagai parameter masukan

```
void CetakLarik(LarikInt A, int n)
/* Mencetak elemen-elemen larik A[1..n]. */
```

Cara pemanggilan: `CetakLarik(A,n)`

Purwarupa prosedur di atas ekivalen dengan penulisan sebagai berikut:

```
void CetakLarik(int A[100], int n)
/* Mencetak elemen-elemen larik A[1..n]. */
```

atau ekivalen dengan:

```
void CetakLarik(int A[], int n) /* ukuran larik A boleh tidak perlu
karena di-set di program utama */
/* Mencetak elemen-elemen larik A[1..n]. */
```

atau ekivalen dengan:

```
void CetakLarik(int *A, int n)
/* Mencetak elemen-elemen larik A[1..n]. */
```

Catatan:
Larik di dalam bahasa C adalah pointer, sehingga dapat ditulis `int *A` *saja.*

(b) Larik sebagai parameter keluaran

```
void BacaLarik(LarikInt A, int n);
/* Mengisi elemen-elemen larik A[1..n] dengan pembacaan. */
```

Cara pemanggilan: `BacaLarik(A, n)`

Catatan:
Karena larik di dalam bahasa C adalah pointer, maka tidak diperlukan karakter "" di awal nama parameter tersebut.*

(c) Larik sebagai parameter masukan/keluaran

```
void UrutLarik(LarikInt A, int n)
/*Mengurut (sort) elemen-elemen larik A[1..n] sehingga tersusun
menaik.*/
```

Cara pemanggilan: `UrutLarik(A, n)`

Catatan:
Karena larik di dalam bahasa C adalah pointer, maka tidak diperlukan karakter "" di awal nama parameter tersebut.*

3. Contoh-contoh translasi algoritma

ALGORITMIK

Prosedur BacaLarik:

```
procedure BacaLarik(output A : LarikInt, input n : integer)
{ Mengisi elemen-elemen larik A[1..n] dengan pembacaan. }
{ K.Awal: n adalah ukuran efektif larik, nilainya terdefinisi. }
{ K.Akhir: setelah pembacaan, seluruh elemen larik A berisi nilai-
  nilai yang dibaca dari piranti masukan. }

DEKLARASI
   i : integer   { pencatat indeks larik }

ALGORITMA:
   for i←1 to n do
      read(A[i])
   endfor
```

Prosedur CetakLarik:

```
procedure CetakLarik(input A : LarikInt, input n : integer )
{ Mencetak elemen-elemen larik A[1..n]. }
{ K.Awal: n sudah berisi jumlah elemen larik yang dipakai. Elemen-
elemen larik A sudah terdefinisi.}
{ K.Akhir: elemen-elemen larik A tercetak. }

DEKLARASI
   i : integer  { pencatat indeks larik }

ALGORITMA:
   for i←1 to N do
      write(A[i])
   endfor
```

Prosedur HitungRataRata:

```
Procedure HitungRataRata(input A : LarikInt, input n : integer,
                         output u : real)
{ Menghitung nilai rata-rata larik A[1..n]. }
{ K.Awal: elemen-elemen larik A sudah terdefinisi; n adalah jumlah
elemen larik, nilainya sudah terdefinisi. }
{ K.Akhir: u berisi nilai rata-rata seluruh elemen. }

DEKLARASI
   i      : integer    {indeks larik }
   jumlah : real       {jumlah total seluruh nilai }
```

```
ALGORITMA:
   i ← 1                  { dimulai dari elemen pertama }
   jumlah ← 0             { jumlah total nilai mula-mula }
   for i←1 to n do
     jumlah ← jumlah + A[i]
   endfor
   u ← jumlah/N
```

Program utama:

```
PROGRAM Rerata
{ Program utama untuk membaca elemen-elemen larik, mencetak elemen-
eleemn larik, dan menghitung nilai rata-ratanya }

DEKLARASI
   const Nmaks = 100  { ukuran maksimum }
   type LarikInt  : array[1..Nmaks] of integer

   A  : LarikInt
   i  : integer    { indeks larik }
   n  : integer    { banyaknya elemen larik yang dipakai, n ≤ Nmaks }
   u  : real       { nilai rata-rata seluruh elemen larik }

   procedure BacaLarik(output A : LarikInt, input n : integer)
   { Mengisi elemen-elemen larik A dengan pembacaan. }

   procedure CetakLarik(input A : LarikInt, input n : integer )
   { Mencetak elemen-elemen larik A. }

   procedure HitungRataRata(input A : LarikInt, input n : integer,
                            output u : real)
   { Menghitung nilai rata-rata elemen larik A. }

ALGORITMA:
   read(n)           { tentukan jumlah elemen larik, n ≤ Nmaks }
   BacaLarik(A,n)
   CetakLarik(A,n)
   HitungRataRata(A,n,u)
   write(u)
```

PASCAL

```
PROGRAM Rerata;
{ Program utama untuk membaca elemen-elemen larik, mencetak elemen-
elemen larik, dan menghitung nilai rata-ratanya }

(* DEKLARASI *)
const
   Nmaks = 100;  { banyaknya elemen larik }
type
   LarikInt  = array[1..Nmaks] of integer;
var
   A : LarikInt;
   i : integer;    { indeks larik }
   n : integer;    { banyaknya elemen larik yang dipakai, n <= Nmaks }
   u : real;       { nilai rata-rata seluruh elemen larik }
```

```
  procedure BacaLarik(var A : LarikInt; n : integer);
  { Mengisi elemen-elemen larik A[1..n] dengan pembacaan. }
  { K.Awal: n adalah ukuran efektif larik, nilainya terdefinisi. }
  { K.Akhir: setelah pembacaan, seluruh elemen larik A berisi nilai-
    nilai yang dibaca dari piranti masukan. }

(* DEKLARASI *)
  var
    i : integer;     { pencatat indeks larik }
    jawab : char;

  (* ALGORITMA: *)
  begin
     for i := 1 to n do
      begin
        write('Masukkan nilai A[',i,']:'); readln(A[i]);
      end;
  end;

  procedure CetakLarik(A : LarikInt; n : integer);
  { Mencetak elemen-elemen larik A[1..n]. }
  { K.Awal: n sudah berisi jumlah elemen larik yang dipakai. Elemen-
    elemen larik A sudah terdefinisi.}
  { K.Akhir: elemen-elemen larik A tercetak. }

(* DEKLARASI *)
  var
    i : integer;  { pencatat indeks larik }

  (* ALGORITMA: *)
  begin
    for i := 1 to N do
      writeln('A[',i,']= ',A[i]);
    {endfor}
  end;

  procedure HitungRataRata(A :LarikInt; n : integer; var u : real);
  { Menghitung nilai rata-rata elemen larik A. }
  { K.Awal: elemen-elemen larik A sudah terdefinisi; n adalah jumlah
    elemen larik, nilainya sudah terdefinisi. }
  { K.Akhir: u berisi nilai rata-rata seluruh elemen larik. }

  (* DEKLARASI *)
  var
    i      : integer; {indeks larik }
    jumlah : real;    {jumlah total seluruh nilai }

  (* ALGORITMA: *)
  begin
     i := 1;          { dimulai dari elemen pertama }
     jumlah := 0;     { jumlah total nilai mula-mula }
     for i := 1 to N do
       jumlah := jumlah + A[i];
     {endfor}
     u:=jumlah/N;
  end;

(* PROGRAM UTAMA: *)
begin
```

```
   writeln('Baca jumlah data (n): '); readln(n);

   writeln('Baca data:');
   BacaLarik(A,n)

   writeln('Cetak data:');
   CetakLarik(A,n);

   writeln('Hitung rata-rata:');
   HitungRataRata(A,n,u);
   writeln('Rata-rata data = ', u:10:6);
end.
```

C

```
/* PROGRAM Rerata */
/* Program utama untuk membaca elemen-elemen larik, mencetak elemen-
elemen larik, dan menghitung nilai rata-ratanya */

/* Deklarasi prosedur yang digunakan */
#include <stdio.h>

const int Nmaks = 100;   /* ukuran maksimum larik */
typedef int LarikInt[Nmaks+1];

void BacaLarik(LarikInt A, int n);
/* Mengisi elemen-elemen larik A[1..n] dengan pembacaan. */

void TulisLarik(LarikInt A, int n);
/* Mencetak elemen-elemen larik A[1..n].*/

void HitungRataRata(LarikInt A, int n, float *u);
/* Menghitung nilai rata-rata seluruh elemen larik A. */


/* PROGRAM UTAMA */
main()
{
 /* DEKLARASI */
   LarikInt A;
   int k;        /* indeks larik */
   int n;        /* banyaknya elemen larik yang dipakai */
   float u;      /* nilai rata-rata */

 /* ALGORITMA: */
   printf("Baca jumlah data (n): "); scanf("%d", &n);

   printf("Baca data: \n");
   BacaLarik(A,n);

   printf("Cetak data: \n");
   CetakLarik(A,n);

   printf("Hitung rata-rata: \n");
   HitungRataRata(A,n,&u)
   printf("Rata-rata data =  %f10:6 \n", u);
}
```

```
void BacaLarik(LarikInt A, int n)
/* Mengisi elemen-elemen larik A[1..n] dengan pembacaan */
/* K.Awal: n adalah ukuran efektif larik, nilainya terdefinisi. */
/* K.Akhir: setelah pembacaan, seluruh elemen larik A berisi nilai-
   nilai yang dibaca dari piranti masukan. */
{
  /* DEKLARASI */
  int i;  /* pencatat indeks larik */

  /* ALGORITMA: */
  for (i=0; i<=n; i++)
  {
    printf("Masukkan nilai A[%d] : ", i); scanf("%d", &A[i]);
  }
}


void CetakLarik(LarikInt A, int n)
/* Mencetak elemen-elemen larik A[1..n]. */
/* K.Awal: n sudah berisi jumlah elemen larik yang dipakai. Elemen-
   elemen larik A sudah terdefinisi.  */
/* K.Akhir: elemen-elemen larik A tercetak. */

{
  /* DEKLARASI */
  int  i;  /* pencatat indeks larik */

 /* ALGORITMA: */
  for (i=1; i<=n; i++)
     printf("A[%d] = %d", i, A[i]);
  /*endfor*/
}

void HitungRataRata(LarikInt A, int n, float *u)
/* Menghitung nilai rata-rata seluruh elemen larik A. */
/* K.Awal: elemen-elemen larik A sudah terdefinisi; n adalah jumlah
   elemen larik, nilainya sudah terdefinisi. */
/* K.Akhir : u berisi nilai rata-rata seluruh elemen.  */

{
  /* DEKLARASI */
   int  i;          /* indeks larik */
   float jumlah;    /* jumlah total seluruh nilai */

  /* ALGORITMA: */
   i = 1;           /* dimulai dari elemen pertama */
   jumlah = 0;      /* jumlah total nilai mula-mula */
   for (i=1; i<= n; i++)
      jumlah = jumlah + A[i];
   /*endfor*/

   *u = jumlah/N;
}
```

12.8 *String* sebagai Larik Karakter

Tipe *string* sudah kita pelajari ketika membicarakan tipe data yang diprogram. *String* pada hakikatnya adalah larik karakter dengan panjang dinamis, artinya ukuran larik baru ditentukan pada saat program *running*. Karena *string* adalah larik, maka elemen-elemen *string* yang berupa karakter dapat diakses melalui indeks. Sebagai contoh, jika *s* adalah peubah bertipe *string* dan *s* berisi konstanta *string* berikut

```
'ini string'
```

maka dapat kita katakan bahwa s panjangnya 10 karakter dan elemen-elemennya adalah:

```
s[1] = 'i'
s[2] = 'n'
s[3] = 'i'
s[4] = ' '
s[5] = 's'
s[6] = 't'
s[7] = 'r'
s[8] = 'i'
s[9] = 'n'
s[10] = 'g'
```

Representasi *string* bergantung pada bahasa pemrograman yang digunakan. Di dalam bahasa *Pascal*, peubah bertipe *string* dideklarasikan dengan nama tipe *string* beserta panjangnya, seperti contoh berikut:

```
var s : string[20];
```

Jika panjang *string* tidak dideklarasikan seperti contoh di bawah ini,

```
var s : string;
```

maka ukuran *string* dianggap *default*, yaitu 255 karakter.

Indeks *string* di dalam bahasa *Pascal* adalah dari 0 sampai *n*, yang dalam hal ini *n* adalah panjang *string*. Karakter pertama dari peubah *string* (pada indeks 0) berisi panjang *string* dinamis; yaitu *Length(s)* sama dengan *Ord(s[0])*. Jadi, jika *string* s dideklarasikan panjangnya 20 karakter, itu berarti ukuran larik karakter sebenarnya adalah 21 karakter , yang dalam hal ini elemen ke-0 berisi panjang *string* tersebut. Catat juga bahwa meskipun *string* s dideklarasikan panjangnya 20, belum tentu seluruh elemennnya terpakai (konsep ini sama seperti pada larik umumnya).

Adapun di dalam bahasa *C*, peubah bertipe *string* dideklarasikan dengan larik bertipe *char* beserta panjangnya, seperti contoh berikut:

```
char s[20];
```

Tidak seperti bahasa *Pascal* yang panjang *string* disimpan di dalam elemen ke-0, maka di dalam bahasa *C*, larik karakter diakhiri dengan karakter *null* '\0' sehingga program dapat menemukan akhir *string*. Sebagai contoh, *string*

```
'ini string'
```

disimpan di dalam larik seperti berikut:

i	n	i		s	t	r	i	n	g	\0

Sebagaimana larik dalam bahasa *C*, indeks *string* adalah dari 0 sampai sampai *n*, yang dalam hal ini *n* adalah panjang *string*. Jadi, jika s adalah peubah bertipe *string* dan s berisi *string* 'ini string', maka elemen-elemen larik s adalah:

```
s[0] = 'i'
s[1] = 'n'
s[2] = 'i'
s[3] = ' '
s[4] = 's'
s[5] = 't'
s[6] = 'r'
s[7] = 'i'
s[8] = 'n'
s[9] = 'g'
```

Hal lain yang harus diperhatikan, ada kemungkinan *string* kosong. Di dalam bahasa *Pascal*, *string* kosong adalah larik karakter yang panjangnya 0, sedangkan di dalam bahasa *C string* kosong adalah larik yang panjangnya 0 dan isinya hanya karakter '\0'.

Di bawah ini kita akan membahas algoritma-algoritma untuk memanipulasi *string* di dalam bahasa *Pascal* dan bahasa *C*, yang tidak lain adalah memanipulasi elemen-elemen larik karakter. Beberapa algoritma manipulasi *string* sudah dijadikan sebagai fungsi/prosedur standar dalam kedua bahasa tersebut.

(a) Menghitung panjang *string*

Diberikan sebuah *string s*, tulislah fungsi untuk menghitung panjangnya.

PASCAL:

Di atas sudah dijelaskan bahwa karakter pertama dari peubah *string* (pada indeks 0) berisi panjang *string* dinamis. Jadi, fungsi untuk menghitung panjang *string s* cukup hanya mengembalikan nilai ordinal dari karakter ke-0, *Ord(s[0])*.

```
function Panjang(s : string):integer;
{ Mengembalikan panjang string }

begin
```

```
  Panjang := ord(s[0]);
end;
```

Catatan:
Bahasa Pascal sudah menyediakan fungsi standard untuk mengembalikan panjang string, yaitu `Length` *Algoritma fungsi* `Length` *sama seperti algoritma fungsi* `Panjang` *ini.*
Contoh: `n := Length(s);`

C:

Di atas sudah dijelaskan bahwa larik karakter diakhiri dengan karakter *null* '\0' sehingga program dapat menemukan akhir *string*. Panjang *string* dihitung dengan memindai (*scan*) elemen-elemen larik karakter sampai ditemukan karakter '\0'.

```
int Panjang(char s[])
/* Mengembalikan panjang string */
{
  int i;

  i = 0;
  while (s[i] != '\0')
    i = i + 1;
  /* endwhile */
  /* s[i] = '\0' */

  return i;
}
```

Catatan:
Bahasa C sudah menyediakan fungsi standard untuk mengembalikan panjang string, yaitu `strlen`. *Fungsi ini didefinisikan di dalam file header* `<string.h>`.

Contoh: n = strlen(s);

(b) Menyalin (*copy*) *string*

Diberikan sebuah *string s1*, salin *s1* ke *string s2*. Algoritma menyalin *string* sama seperti menyalin larik ke larik lain (lihat 12.4.10).

PASCAL:

```
procedure SalinString(s1 : string; var s2 : string);
{ Menyalin (copy) string s1, menghasilkan string s2 }
  K.Awal: s1 sudah terdefinisi
  K.Akhir: s2 adalah salinan s1
}
var
```

```
  i : integer;

begin
  for i := 1 to Length(s1) do
    s2[i] := s1[i];
  {endfor}
end;
```

Catatan:
Di dalam bahasa Pascal, penyalinan string s1 menghasilkan s2 juga dapat dilakukan secara langsung dengan assignment:

Contoh: `s2:=s1;`

C:

```
void SalinString(char s1[], char s2[])
/* Menyalin (copy) string s1, menghasilkan string s2 }
  K.Awal: s1 sudah terdefinisi
  K.Akhir: s2 adalah salinan s1, termasuk '\0'
*/
{
  int i;

  i = 0;
  while (s1[i] != '\0')
  {
    s2[i] = s1[i];
    i = i + 1;
  }
  /*endwhile*/
  /* s1[i] = '\0' */

  s2[i] = '\0'; /*string hasil copy harus diakhiri dengan karakter null*/
}
```

Catatan:
1. *Fungsi di atas tidak menggunakan fungsi* `strlen` *atau* `Panjang` *supaya tidak memindai (scan) atau menelusuri larik yang sama dua kali.*
2. *Bahasa C sudah menyediakan fungsi standard untuk menyalin string, yaitu* `strcpy`*. Fungsi ini didefinisikan di dalam file header* `<string.h>`.

   Contoh: `strcp(t,s);` */* copy s ke t */*

(c) Membandingkan dua buah *string*
Diberikan dua buah *string s1* dan *s2*. Bandingkan apakah kedua *string* tersebut sama. Algoritma membandingkan dua buah *string* mirip dengan membandingkan dua buah larik (lihat 12.4.11).

PASCAL:

```
function ApakahSama(s1, s2 : string);
{ true jika s1 = s2, atau false jika s1 <> s2 }

var
  i : integer;
  sama : boolean;

begin
  { cek panjang string }
  if Length(s1) <> Length(s2) then  { s1 dan s2 tidak sama panjangnya }
     ApakahSama := false
  else
  { panjang s1 dan s2 sama, cek kesamaan elemen-elemennya }
   begin
     i := 1;
     sama:=true;    { asumsikan sementara bahwa s1 dan s2 sama }
     while (i <= Length(s1)) and (sama) do
        if s1[i] = s2[i] then
           i := i + 1    { tinjau elemen berikutnya }
        else  { s1[i] <> s2[i], maka dapat disimpulkan s1 <> s2 }
          sama := false;
        {endif}
     {endwhile}
     { i > Length(s1) or not sama}

     ApakahSama := sama;
   end;
end;
```

Catatan:
Di dalam bahasa Pascal, membandingkan string s1 dan s2 juga dapat dilakukan secara langsung dengan operator kesamaan:

```
if s1 = s1 then
   ...  { aksi yang dilakukan jika s1 = s2 }
```

C:

```
int ApakahSama(char s1[], char s2[])
{ 1 jika s1 = s2, atau 0 jika s1 <> s2 }
*/
{
 int i;
 boolean sama;

  /* cek panjang string */
  if (strlen(s1) != strlen(s2))  /* s1 dan s2 tidak sama panjangnya */
     return 0
  else
   /* panjang s1 dan s2 sama, cek kesamaan elemen-elemennya */
   {
    i = 1;
    sama = true;    /* asumsikan sementara bahwa s1 dan s2 sama */
    while (s1[i] != '\0' && sama)
```

```
      if (s1[i] == s2[i])
         i = i + 1;   /* tinjau elemen berikutnya */
      else  /* s1[i] != s2[i], maka dapat disimpulkan s1 != s2 */
         sama = false;
      /*endif*/
   /*endwhile*/
   /* s[i] = '\0' || not sama */

   return sama;
  }
}
```

Catatan:

1. *Fungsi di atas memindai larik dua kali (satu kali pemindaian untuk menghitung panjang string dengan fungsi strlen dan satu kali pemindaian untuk membandingkan kesamaan elemen-elemen string). Bisakah anda buat algoritma yang lebih mangkus dengan hanya memindai larik satu kali saja?*
2. *Bahasa C sudah menyediakan fungsi standard untuk memabndingkan dua buah string, yaitu strcmp. Fungsi ini didefinisikan di dalam file header* `<string.h>` *. Fungsi* `strcmp` *mengembalikan nilai 0 jika s1 = s2; mengembalikan nilai < 0 jika panjang s1 < panjang s2; mengembalikan nilai > 0 jika panjang s1 > panjang s2.*

   Contoh: `n = strcmp(s1,s2);`

Soal Latihan Bab 12

1. Diberikan larik *integer A* yang berukuran *n* elemen. Larik *A* sudah terdefinisi elemen-elemennya. Tuliskan prosedur untuk mencari nilai *X* di dalam larik, dengan parameter keluaran adalah indeks dari kemunculan *terakhir* elemen *X*. Jika *X* tidak terdapat di dalam larik, prosedur mengeluarkan indeks 0.

2. Diberikan larik *integer A* yang berukuran *n* elemen. Elemen-elemen larik A sudah terurut menaik (dari kecil ke besar). Tuliskan fungsi untuk menghitung jangkauan (*range*) nilai di dalam larik. Jangkauan adalah selisih nilai terbesar dengan nilai terkecil.

3. Diberikan larik *integer A* yang berukuran *n* elemen. Larik *A* sudah terdefinisi elemen-elemennya. Tuliskan prosedur yang keluarannya adalah elemen terbesar pertama dan elemen terbesar kedua di dalam larik *A*.

4. Diberikan larik *integer* A yang berukuran n elemen. Larik A sudah terdefinisi elemen-elemennya. Tuliskan prosedur untuk menampilkan elemen-elemen larik A yang lebih kecil dari elemen terkecilnya.

5. Diberikan larik *integer* A dan *integer* larik B yang masing-masing berukuran N elemen. Larik A dan B sudah terdefinisi elemen-elemennya. Tuliskan prosedur untuk mempertukarkan elemen larik A dan elemen B pada indeks yang bersesuaian, sedemikian sehingga larik A berisi elemen-elemen larik B semula dan larik B berisi elemen-elemen larik A semula.

6. Diberikan larik karakter A yang berukuran n elemen. Larik A sudah terdefinisi elemen-elemennya. Tuliskan prosedur yang membalikkan elemen-elemen larik A sedemikian sehingga elemen terakhir pada larik semula menjadi elemen pertama pada larik akhir.
Contoh:

sebelum pembalikan:

m	*a*	*r*	*a*	*h*

setelah pembalikan:

h	*a*	*r*	*a*	*m*

7. Diberikan n buah data statistik yang sudah diisikan ke dalam larik *real* X. **Simpangan baku** atau standar deviasi dari data statistik tersebut didefinisikan dengan rumus berikut:

$$d = \sqrt{\frac{\sum_{i=1}^{n}(X_i - \overline{X})^2}{n-1}}$$

yang dalam hal ini, $\overline{X}$ adalah nilai rata-rata dari seluruh data. Tuliskan prosedur untuk menghitung simpangan baku.

8. Diberikan larik *integer* A yang berukuran n elemen. Larik A sudah terdefinisi elemen-elemennya. Tuliskan prosedur untuk menampilkan semua elemen larik yang lebih besar dari jumlah seluruh elemen yang sebelumnya (*previous elements*).

9. Diberikan larik *integer* A yang berukuran n elemen. Larik A sudah terdefinisi elemen-elemennya. Tuliskan prosedur untuk menampilkan elemen-elemen larik A yang lebih kecil dari elemen terkecilnya.

10. Diberikan larik *integer* A yang berukuran n elemen. Larik A sudah terdefinisi elemen-elemen yang sudah terurut menaik (*ascending order*). Tuliskan prosedur untuk menghasilkan median dari elemen-elemen tersebut (median adalah elemen tengah dari sekumpulan elemen yang sudah tersusun terurut).

11. Diberikan larik *integer A* yang berukuran *n* elemen. Larik *A* sudah terdefinisi elemen-elemennya. Tuliskan prosedur untuk menentukan apakah seluruh elemen di dalam larik *A* tersebut sama.

12. Diberikan sebuah *string s*. Tuliskan fungsi dalam bahasa *Pascal* dan bahasa *C* untuk menyalin elemen *string s* dari posisi *i* sebanyak *n* karakter. Jika *i* lebih besar dari panjang *string*, maka *string* kosong yang dihasilkan. *String* kosong juga dihasilkan jika *n* lebih besar dari panjang *string* yang tersisa (dihitung dari posisi *i*).

13. Diberikan dua buah *string*, *s1* dan *s2*. Tuliskan prosedur dalam bahasa *C* untuk menyambung (*concat*) *s1* dengan *s2*.

14. Diberikan sebuah *string s*. Tuliskan fungsi dalam bahasa-bahasa *C* untuk menghilangkan kemunculan semua karakter *c* dari *s*.

13

Contoh-contoh Pemecahan Masalah II

Bab 13 ini menyajikan contoh-contoh pemecahan masalah yang mencakup materi larik, termasuk di dalamnya prosedur dan fungsi.

Contoh Masalah 13.1: Diskon harga jual barang

Dalam rangka merayakan ulang tahunnya, sebuah toserba memberi diskon sebesar *p* % untuk seluruh barang. Misalkan data kode barang dan harga jual barang tersebut telah diisimpan di dalam larik *HargaJual*. *HargaJual* bertipe *LarikBarang* yang deklarasi datanya sebagai berikut:

```
DEKLARASI
   const Nmaks = 100
   type Barang : record
                   <KodeBrg : integer,
                    HargaBrg: real
                   >
   type LarikBarang : array [1..Nmaks] of Barang

   HargaJual : LarikBarang
```

Tulislah prosedur untuk menurunkan harga jual barang-barang di toserba dengan diskon yang diberikan. Jumlah *item* barang ada *N* buah.

Penyelesaian

```
procedure TurunkanHarga(input/output HargaJual : LarikBarang,
                        input p : real, input N : integer)
{ Menurunkan harga seluruh barang sebesar diskon p %. }
{ K.Awal: N adalah jumlah item barang yang ada di toserba; larik
  HargaJual[1..N] sudah terdefinisi elemen-elemennya.
{ K.Akhir: elemen-elemen larik HargaJual berkurang p % dari semula }

DEKLARASI
```

```
  i : integer          { indeks larik }
  diskonHarga : real

ALGORITMA:
   for i ← 1 to N do
       diskonHarga ← (p/100)*HargaJual[i]
       HargaJual[i] ← HargaJual[i] - diskonHarga
   Endfor
```

Algoritma 13.1 Mengubah harga jual barang setelah pemberian diskon.

Contoh Masalah 13.2: Menghitung jumlah kemunculan '1'

Diberikan sebuah larik karakter yang berukuran *n* elemen. Karakter di dalam larik hanya ada dua macam: '0' dan '1'. Buatlah prosedur untuk menghitung jumlah kemunculan '1' di dalam larik tersebut.

Penyelesaian

Untuk menghitung jumlah kemunculan '1' di dalam larik, kita harus memeriksa setiap elemen larik mulai dari elemen pertama sampai elemen ke-n. Jika elemen larik sama dengan '1', maka pencacah jumlah '1' dinaikkan 1. Misalkan nama larik adalah *C* dan tipenya adalah *LarikKar*. Deklarasi *LarikKar* adalah seperti di bawah ini:

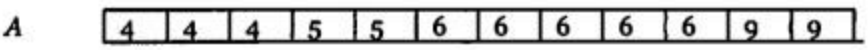
```
DEKLARASI
  const Nmaks = 100    { jumlah elemen larik }
  type LarikKar : array[1..Nmaks] of char

procedure HitungJumlah_1(input C : LarikKar, input n : integer,
                         output m : integer)
{ Menghitung banyaknya karakter '1' di dalam larik C. }
{ K.Awal: Elemen larik C[1..n] sudah berisi karakter '0' dan '1' }
{ K.Akhir: m berisi banyaknya karakter '1' di dalam larik C }

DEKLARASI
   i : integer

ALGORITMA:
   m ← 0
   for i ← 1 to n do
     if C[i] = '1' then
        m ← m + 1
     endif
   endfor
```

Algoritma 13.2 Menghitung jumlah kemunculan karakter '1'.

Contoh Masalah 13.3: Menghitung modus

Misalkan larik *integer A* berisi *n* buah elemen elemen-elemen yang sudah terurut menaik/ membesar. Buatlah prosedur untuk menghitung **modus**. Modus adalah elemen yang paling sering muncul di dalam larik.

Misalnya, modus dari larik *A* di bawah ini adalah 6.



modus = 6

Penyelesaian

Karena larik sudah terurut, maka menentukan modus elemennya relatif mudah daripada larik yang tidak terurut. Mula-mula frekuensi modus sementara (*FrekModus*) kita inisialisasi dengan 0. Selanjutnya elemen pertama kita anggap kandidat modus (*KandidatModus*) dengan frekuensi kemunculanya (*FrekKandidatModus*) adalah 1. Jika elemen-elemen berikutnya sama dengan *KandidatModus*, maka *FrekKandidatModus* selalu dinaikkan 1. Jika elemen berikutnya sudah berbeda, maka modus dari elemen pertama sudah selesai dihitung, selanjutnya kita akan menghitung modus elemen yang baru, namun sebelumnya bandingkan dulu *FrekKandidatModus* dengan *FrekModus* modus sementara; jika *FrekKandidatModus* lebih besar atau sama dengan *FrekModus*, maka *KandidatModus* menjadi modus sementara yang baru.

Prosedurnya menentukan modus sebagai berikut:

```
procedure HitungModus(input A : LarikInt, input n : integer,
                      output modus : integer)

{ Menghitung modus dari elemen-elemen larik A. }
{ K.Awal: elemen-elemen larik A sudah berisi nilai integer yang sudah
  terurut menaik/membesar; n adalah ukuran latik A }
{ K.Akhir: modus berisi elemen yang paling banyak muncul di dalam A }

DEKLARASI
   i : integer    { indeks larik }
   MasihSama : boolean
   FrekModus, KandidatModus, FrekKandidatModus : integer

ALGORITMA:
   FrekModus ← 0    { frekuensi kemunculan modus }
   i ← 1            { tinjau kandidat modus mulai dari elemen pertama }
   while i ≤ N do
     KandidatModus ← A[i]      { A[i] dianggap calon modus }
     FrekKandidatModus ← 1     { frekuensi kemunculan kandidat modus }
     i ← i+1                   { tinjau elemen i+1 }
     MasihSama ← true
     while (MasihSama) and (i ≤ N) do
        if A[i] = KandidatModus then
           FrekKandidatModus ← FrekKandidatModus + 1
        endif
        i ← i + 1

     endwhile
     { not MasihSama or i > N }
     { mutakhirkan data modus sementara }
     if FrekKandidatModus ≥ FrekModus then
```

```
        modus ← KandidatModus
        FrekModus ← FrekKandidatModus
    endif
  endwhile
```

Algoritma 13.3 Menghitung modus dari larik *integer* yang sudah terurut menaik.

Contoh Masalah 13.4: Menghitung jumlah kemunculan elemen-elemen di dalam larik

Misalkan larik *integer A* berisi *n* buah elemen yang nilai-nilainya berada di dalam rentang 1 sampai *m*. Elemen-elemen larik *A* tidak terurut. Buatlah prosedur untuk menghitung frekuensi kemunculan setiap elemen. Frekuensi kemunculan elemen larik *A* disimpan di dalam larik *countA*[1..*m*] sedemikian sehingga *countA*[*i*] berisi kemunculan nilai *i*.

Sebagai contoh, misalkan larik *A* di bawah ini dicatat frekuensi kemunculan elemen-elemennya di dalam larik *countA*. Nilai *countA*[1] = 2, artinya jumlah kemunculan 1 ada 2 buah, *countA*[2] = 3 artinya jumlah kemunculan 2 ada 3 buah, *countA*[5] = 0 karena jumlah kemunculan 5 tidak ada.

A	4	2	4	3	6	3	3	2	6	3	1	1

countA	2	3	4	1	0	2

Penyelesaian

Kita akan mencacah frekuensi kemunculan setiap elemen di larik *A* dan menyimpannya di dalam larik *countA*. Mula-mula seluruh elemen larik *countA* kita inisialisasi dengan 0. Selanjutnya, elemen-elemen larik *A* dikunjungi lalu frekuensi kemunculannya dihitung dengan pernyataan berikut:

```
countA[A[i]] ← countA[A[i]] + 1
```

```
procedure HitungFrek(input  A : LarikInt, input n : integer,
                     output countA : LarikInt, input m : integer)

{ Menghitung frekuensi kemunculan elemen-elemen larik A[1..n].
Hasilnya disimpan di dalam larik countA[1..m] }
{ K.Awal: elemen-elemen larik A sudah berisi nilai integer yang nilai-
  nilainya terletak dari 1 sampai m }
{ K.Akhir: countA[1..m] berisi frekuensi kemunculan elemen di dalam
  larik A sedemikian sehingga countA[i] berisi frekuensi kemunculan
  nilai i. }
DEKLARASI
   i: integer    { indeks larik }

ALGORITMA:
   { inisialisasi seluruh elemen larik countA dengan 0 }
   for i ← 1 to m do
      countA[i] ← 0
   endfor
```

```
{ hitung frekuensi kemunculan elemen larik A }
for i←1 to N do
   countA[[A[i]]←countA[A[i]]+1
endfor
```

Algoritma 13.4 Menghitung frekuensi kemunculan elemen-elemen di dalam larik.

Contoh Masalah 13.5: Menggambar histogram jumlah kemunculan elemen-elemen di dalam larik

Misalkan *countA*[1..*m*] sudah berisi jumlah kemunculan nilai-nilai yang berada di dalam rentang 1 sampai *m* sedemikian sehingga *countA*[*i*] berisi kemunculan nilai *i* (larik *countA* dihasilkan oleh algoritma pada Contoh Masalah 13.4). Buatlah prosedur dalam bahasa *Pascal* dan bahasa *C* untuk mencetak histogram yang menyatakan jumlah kemunculan nilai-nilai di dalam larik. Perhatikan bahwa lebih mudah menggambar histogram secara mendatar (menggambar histogram secara vertikal jauh lebih sukar).

Sebagai contoh, tinjau larik *countA* dari Contoh Masalah 13.4. Histogram yang digambarkan adalah seperti di bawah ini (satu unit dinyatakan dengan karakter bintang):

```
1 **
2 ***
3 ****
4 *
5
6 **
```

Penyelesaian

countA[*i*] berisi jumlah kemunculan nilai *i*. Untuk setiap baris *i*, cetak karakter "*" sebanyak nilai *countA*[*i*].

```
procedure CetakHistogram(countA : LarikInt; m : integer);
{ Menggambar histogram dari countA[1..m]. countA[i] berisi jumlah
kemunculan nilai i. }
{ K.Awal: elemen-elemen larik countA sudah berisi nilai integer; m
sudah berisi ukuran larik countA }
{ K.Akhir: histogram yang menyatakan countA[1..m] tercetak. }

(* DEKLARASI *)
var
   i, j : integer;  { indeks larik }
(* ALGORITMA: *)

begin
   for i:=1 to m do
    begin
      write(i,' ');
      for j:=1 to count[i] do
         write('*');
      {endfor}
```

```
      writeln;  { ganti baris baru }
    end;{for}
end;
```

Algoritma 13.5 Menggambar histogram dari larik countA[1..m].

Contoh Masalah 13.6: Palindrom

Diberikan sebuah larik karakter yang bernama *Huruf* dan berukuran *n* elemen. yang sudah terdefinisi setiap elemennya (sudah berisi karakter). Susunan elemen-elemen larik karakter itu membentuk sebuah 'kata'. Kita ingin menentukan apakah kata yang dibentuk oleh rangkaian elemen tersebut bersifat *palindrom*. Sebuah kata bersifat *palindrom* jika dibaca dari kiri atau kanan menghasilkan kata yang sama. Tuliskan algoritmanya.

Contoh:

m	a	l	a	m	palindrom
k	a	t	a	k	palindrom
t	a	a	t		palindrom
k	u	l	a	k	bukan palindrom

Penyelesaian

Gagasan algoritma palindrom adalah memindai (*scan*) elemen larik dari arah "kiri" dan dari arah "kanan" satu per satu. Bandingkan elemen kiri dan elemen kanan, apakah sama. Jika sama, teruskan pembandingan sampai pemindaian bertemu di tengah larik. Jika pada pemeriksaan ditemukan huruf yang tidak sama, pemeriksaan dihentikan dan dapat disimpulkan kata tersebut bukan palindrom. Sebuah kata adalah *palindrom* jika pada pemeriksan sampai bertemu di tengah-tengah, huruf di kiri dan di kanan selalu sama.

Kita menyatakan algoritma palindrom dengan dua versi fungsi sebagai berikut, versi pertama tanpa menggunakan peubah *boolean*, sedangkan versi kedua menggunakan peubah *boolean*:

(a) Versi 1 (tanpa menggunakan peubah *boolean*)

```
function Palindrom(input Huruf : LarikKar, input n : integer)→boolean
{ Mengembalikan nilai true jika rangkaian huruf di dalam larik Huruf
  membentuk kata yang palindrom, atau false jika tidak palindrom. }

DEKLARASI
  i : integer          { indeks penelusuran dari kiri }
  j : integer          { indeks pemindaian dari kanan }

ALGORITMA:
```

```
i ← 1
j ← n
while (i ≤ j) and (Huruf[i]=Huruf[j]) do
  i ← i + 1
  j ← j - 1
endwhile
{ i > j or Huruf[i]≠Huruf[j] }

if i > j  then   {pengujian berhasil melewati tengah larik }
  return true
else  {i < j, ada huruf kiri yang ≠ huruf kanan }
  return false
endif
```

Algoritma 13.6 Menguji palindrom (versi 1).

(b) Versi 2 (dengan menggunakan peubah *boolean*)

```
function Palindrom(input Huruf : LarikKar, input n : integer)→boolean
{ Mengembalikan nilai true jika rangkaian huruf di dalam larik Huruf
  membentuk kata yang palindrom, atau false jika tidak palindrom. }

DEKLARASI
  i    : integer      { indeks pemindaian dari kiri }
  j    : integer      { indeks pemindaian dari kanan }
  pal  : boolean      { true jika palindrom, false jika tidak }

ALGORITMA:
  i ← 1
  j ← n
  pal ← true
  while (i ≤ j) and (pal) do
     if Huruf[i] ≠ Hiruf[j] then
        pal ← false
     else
        i ← i + 1
        j ← j - 1
     endif
  endwhile
  { i > j or (not pal)  }

  return pal
```

Algoritma 13.7 Menguji palindrom (versi 2).

Contoh program yang memanggil fungsi `Palindrom` adalah seperti di bawah ini. Elemen-elemen larik dibaca terlebih dahulu dengan memanggil prosedur `BacaLarikKar`.

```
PROGRAM UjiPalindrom
{ Memeriksa apakah sebuah larik karakter membentuk kata yang palindrom }

DEKLARASI
   const Nmaks = 100
```

```
type LarikKar : array [1..Nmaks] of char

Huruf : LarikKar
n : integer

function Palindrom(input Huruf : LarikKar,
                   input N : integer)→boolean
{ Mengembalikan nilai true jika rangkaian huruf di dalam larik
  Huruf membentuk kata yang palindrom, atau false jika tidak
  palindrom. }

procedure BacaLarikKar(output Huruf : LarikKar, input n : integer)
{ Membaca elemen-elemen larik Huruf. }

ALGORITMA:
  read(n)     { baca ukuran larik }
  BacaLarikKar(Huruf,n)  { isi elemen-elemen larik dengan pembacaan }
  if Palindrom(Huruf,n) then
    write('palindrom')
  else
    write('bukan palindrom')
  endif
```

Algoritma 13.8 Menguji palindrom (versi 2).

Contoh Masalah 13.7: Polinom

Polinom p berderajat n ditulis dalam bentuk

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

Koefisien-koefisien polinom adalah $a_0, a_1, ..., a_n$. Misalnya,

$$p(x) = 2.67 + 3.28x - 6.5x^2 + 17.0x^5$$

adalah polinom berderajat 5, yang dalam hal ini,

$a_0 = 2.67$, $a_1 = 3.28$, $a_2 = -6.5$, $a_3 = 0$,

$a_4 = 0$, dan $a_5 = 17.0$.

Polinom direpresentasikan dengan menyimpan koefisien-koefisiennya di dalam larik bertipe *real*. Jika suatu suku (*term*) bernilai 0, maka koefisien suku tersebut diisi 0 (pada contoh di atas, $a[3] = 0$ dan $a[4] = 0$. Misalkan struktur data untuk polinom adalah sebuah tipe terstruktur yang mempunyai dua buah *field*, yaitu n (derajat polinom), dan a (larik yang menyimpan koefisien-koefisien polinom):

```
DEKLARASI
  const Nmaks = 100
  type KoefPolinom : array [0..Nmaks] of real
  type Polinom : record
                   <
                     n : integer      { derajat polinom }
                     a : KoefPolinom
                   >
```

Algoritma 13.9 Deklarasi struktur data untuk polinom.

Operasi terhadap polinom antara lain mengevaluasi polinom untuk nilai x tertentu, misalnya (dengan menggunakan contoh polinom di atas):

$$p(0) = 2.67 + 3.28(0) - 6.5(0)^2 + 17.0(0)^5 = 2.67$$

Operasi lainnya adalah menjumlahkan dua buah polinom, misalnya

$$p_1(x) = 2.67 + 3.28x - 6.5x^2 + 17.0x^5$$

$$p_2(x) = 1.03 - 1.14x^2 - 4.1x^3$$

$$p_3(x) = p_1(x) + p_2(x) = 3.70 + 3.28x - 7.64x^2 - 4.1x^3 + 17.0x^5$$

Buatlah prosedur untuk: (i) membaca koefisen-koefisien polinom, (ii) mengevaluasi nilai polinom p untuk x yang diberikan, dan (iii) menjumlahkan dua buah polinom, yang dalam hal ini polinom pertama berderat n dan polinom kedua berderajat m.

Penyelesaian

(i) Membaca koefisien-koefisien polinom

Prosedur untuk membaca koefisien-koefisien polimom sebagai berikut:

```
procedure BacaKoefPolinom(output p : Polinom)
{ Membaca koefisien-koefisien polinom berderajat n. }
{ K.Awal: p.n sudah terdefinisi dengan derajat polinom }
{ K.Akhir: seluruh elemen larik a berisi koefisien-koefisien polinom }

DEKLARASI
   i : integer   { pencatat indeks larik }

ALGORITMA:
   for i ← 0 to p.n do
      read(p.a[i])
   endfor
```

Algoritma 13.10 Membaca koefisien-koefisien polinom.

(ii) Mengevaluasi polinom

Setelah polinom terdefinisi, kita dapat mengevaluasi nilai polinom untuk sembarang nilai x.

Di dalam Bab 11 kita sudah membuat fungsi untuk menghitung perpangkatan. Kita menggunakan fungsi Pangkat untuk menghitung x^m. Nilai polinom dihitung dengan pernyataan penjumlahan:

```
jumlah ← jumlah + p.a[i]*Pangkat(x,i)
```

Pernyataan penjumlahan di atas diulangi mulai dari $i = 0$ sampai $i = n$.
Fungsi untuk menghitung perpangkatan x^m:

```
function Pangkat(input x : real, input m : integer) → real
{ mengembalikan harga perpangkatan x^m }
```

```
DEKLARASI
   p : real
   i : integer

ALGORITMA:
   p ← 1
   for i ← 1 to m do
      p ← p * x
   endfor

   return hasil
```

Algoritma 13.11 Fungsi untuk menghitung perpangkatan, x^m.

Fungsi untuk menghitung evaluasi $p(x)$:

```
function poli(input p : Polinom, input x : real)→real
{ mengembalikan nilai p(x) }

DEKLARASI
  jumlah : real
     i   : integer

    function Pangkat(input x : real, input m : integer) → real
  { mengembalikan harga perpangkatan x^m }

ALGORITMA:
   jumlah←0
   for i←0 to n do
      jumlah ← jumlah + p.a[i]*Pangkat(x,i)
   endfor

   return jumlah
```

Algoritma 13.12 Fungsi untuk mengevaluasi polinom p(x) (versi 1).

Perhitungan evaluasi polinom $p(x)$ dapat pula kita lakukan tanpa menggunakan fungsi Pangkat. Dengan mengamati bahwa $a_i x^i = a_i * x * x * \ldots * x$ dihitung dengan i operasi perkalian dan mengingat bahwa x^i diperoleh dari $x * x^{i-1}$, maka perpangkatan x^i dapat dilakukan sekali "jalan" di dalam kalang *FOR*. Dengan demikian, algoritma yang lebih elegan untuk mengevaluasi nilai polinom $p(x)$ dapat kita tulis seperti di bawah ini:

```
function poli(input p : Polinom, input x : real) → real
{ mengembalikan nilai p(x) }

DEKLARASI
  jumlah : real
  i      : integer
  pkt    : real      { untuk menyimpan perkalian dengan x }

ALGORITMA:
```

```
pkt ← 1
jumlah ← p.a[0]
for i ← 1 to n do
   pkt ← pkt * x
   jumlah ← jumlah + p.a[i] * pkt
endfor
return jumlah
```

Algoritma 13.13 Fungsi untuk mengevaluasi polinom p(x) (versi 2).

(iii) Menjumlahkan dua buah polinom

Penjumlahan dua buah polinom dilakukan dengan menjumlahkan koefisien-koefisien dari suku-suku yang bersesuaian. Derajat polinom hasil penjumlahan adalah derajat polinom yang terbesar dari polinom-polinom *operand*. Cara penjumlahan dua buah polinom diilustrasikan sebagai berikut:

$p_1(x) = 2.67 + 3.28x - 6.5x^2 + \quad + 17.0x^5$

$p_2(x) = 1.03 + \quad - 1.14x^2 - 4.1x^3 \quad +$

$p_3(x) = 3.70 + 3.28x - 7.64x^2 - 4.1x^3 \quad + 17.0x^5$

Algoritma penjumlahan 2 polinom sebagai berikut:

```
procedure JumlahPolinom(input p1, p2 : polinom, output p3 : polinom)
{ Menjumlahkan polinom p1 dan p2.
{ K.Awal: polinom p1 sudah terdefinisi derajat dan koefisien-koefisienya}
{ K.Akhir: p3 adalah polinom hasil penjumlahan }

DEKLARASI
  i      : integer

ALGORITMA:
  { tentukan derajat polinom p3 }
  if p1.n > p2.n then
     p3.n ← p1.n
  else
     p3.n ← p2.n
  endif

  { jumlahkan koef. polinom p1 dan p2 dari suku-suku berderajat sama }
  i ← 0
  while (i ≤ p1.n) and (i ≤ p2.n) do
     p3.a[i] ← p1.a[i] + p2.a[i]
     i ← i + 1
  endwhile
  { i > p1.n or i > p2.n }
 { salin sisa koefisien dari p1, jika ada }
  while (i ≤ p1.n) do
     p3.a[i] ← p1.a[i]
     i ← i + 1
  endwhile

 { salin sisa koefisien dari p2, jika ada }
```

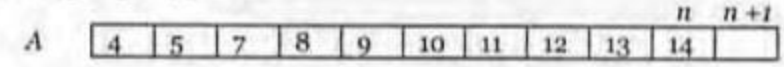
```
while (i ≤ p2.n) do
   p3.a[i] ← p2.a[i]
   i ← i + 1
endwhile
```

Algoritma 13.14 Menjumlahkan dua buah polinom.

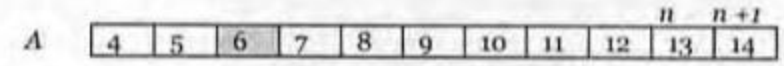
Contoh Masalah 13.8: Menyisipkan elemen ke larik terurut

Misalkan larik *integer A* berisi *n* buah elemen yang sudah terurut membesar (menaik). Tulislah prosedur untuk menyisipkan *x* ke dalam larik tersebut sehingga larik *A* tetap terurut menaik. Akibat penyisipan, elemen larik *A* bertambah 1 (asumsikan bahwa $n + 1 \leq Nmaks$). Sebagai contoh, misalkan kita menyisipkan 6 ke dalam larik *A* di bawah ini.

Sebelum penyisipan 6:



Setelah penyisipan 6:



Penyelesaian

Penyisipan elemen baru ke dalam larik yang sudah terurut mengharuskan terjadinya pergeseran elemen. Sebelum menyisipkan *x*, kita harus mencari posisi yang tepat untuk menempatkan *x*. Jika posisi tersebut sudah ditemukan, maka elemen-elemen pada posisi berikutnya digeser satu posisi ke "kanan".

```
procedure Sisip(input  A : LarikInt, input/output n : integer,
                input x : integer)
{ Menyisipkan x ke dalam larik A yang sudah terurut membesar
{ K.Awal: elemen-elemen larik A sudah terurut membesar; n sudah berisi
  ukuran larik. }
{ K.Akhir: x sudah berada pada posisi yang tepat sedemikian sehingga
  larik A tetap terurut membesar. Setelah penyisipan, ukuran larik A,
  yaitu n, bertambah 1 menjadi n + 1 }

DEKLARASI
   i      : integer   { indeks larik }
   ketemu : boolean

ALGORITMA:
   { cari dulu posisi untuk x dengan cara menemukan elemen yang lebih
     besar dari x }
   ketemu ← false
   i ← 1
   while (i ≤ n) and (not ketemu) do
     if x < A[i] then
       ketemu ← true   { posisi untuk x ditemukan }
     else
       i ← i+1
     endif
```

```
endwhile
{ i > N or ketemu }

if ketemu then

  { geser elemen A[i], A[i+1], ..., A[n] satu posisi ke kanan }
   for j ←n downto i do
      A[j+1]←A[j]
   endfor

   A[i]←x    { sisipkan x pada posisi ke-i }

else
    { berarti semua elemen A[1..n] lebih kecil dari x; sisipkan x
      sebagai elemen A[n+1] }
   A[n+1]←x
endif

n ← n + 1    { ukuran larik A bertambah 1 }
```

Algoritma 13.15 Menyisipkan elemen baru, x, ke dalam larik yang sudah terurut.

Contoh Masalah 13.9: Konversi *string* ke *integer*

Misalkan sebuah *string* telah berisi rangkaian karakter angka ('0'..'9') sehingga membentuk sebuah "bilangan". Tulislah program dalam bahasa *Pascal* untuk mengonversi rangkaian karakter angka tersebut ke dalam bilangan bulat. Sebagai contoh, *string* '3725' dikonversi menjadi *integer* sebagai berikut:

$(5 \times 1) + (2 \times 10) + (7 \times 100) + (3 \times 1000) = 3725$

Penyelesaian

Kesulitan yang timbul dalam pengonversian *string* bilangan menjadi nilai *integer* adalah menentukan faktor pengali sebuah karakter angka, apakah dikali dengan 10 atau dikali dengan 100, dan sebagainya.

Versi 1: menghitung mulai dari belakang

Dengan mengingat bahwa *string* sebenarnya adalah larik bertipe karakter, maka tahap pengonversian dimulai dari elemen larik terakhir menuju elemen pertama. Faktor pengali untuk elemen terakhir adalah 1, dan setiap kali pengulangan faktor pengali untuk elemen berikutnya adalah dikalikan dengan 10. Sebelum dikalikan dengan faktor pengali, setiap karakter angka harus dikonversi terlebih dahulu ke angka *integer*-nya dengan menggunakan fungsi *KarKeInt* yang sudah ditulis pada Contoh Masalah 11.6.

```
procedure KonversiStrKeInt(s : string; var n : integer);
{ Mengkonversi barisan karakter angka di dalam string s menjadi nilai
  integer. Versi 1 }
{ K.Awal: s sudah berisi berisi barisan karakter angka ('0'..'9') }
{ K.Akhir: n berisi representasi integer dari string s }

(*DEKLARASI*)
var
   i       : integer;  { indeks slarik }
```

```
   pengali : integer;  { faktor pengali }

   function KarKeInt(x : char):integer;
   { mengkonversi karakter digit ('0'..'9') ke integer (0..9) }

(* ALGORITMA: *)
begin
   pengali:=1;
   n:=0;
   for i:=Length(s) downto 1 do
    begin
      n := n + KarKeInt(s[i]) * pengali;
      pengali := pengali * 10;
    end;{for}
end;
```

Algoritma 13.16 Konversi *string* s menjadi *integer* (versi 1).

Versi 2: menghitung mulai dari depan

Tahap pengonversian dimulai dari elemen pertama, kedua, dan seterusnya. Setiap elemen selalu ditambah dengan hasil perkalian 10 dengan sebuah nilai konversi sementara (nilai konversi awal = 0). Nilai konversi sementara selalu dikali dengan 10 dan ditambah dengan elemen *string* yang sedang diacu. setiap karakter angka harus dikonversi terlebih dahulu ke angka *integer*-nya dengan menggunakan fungsi *KarKeInt* yang sudah ditulis pada Contoh Masalah 11.6. Versi 2 ini lebih sederhana daripada versi pertama.

```
procedure KonversiStrKeInt(s : string; var n : integer);
{ Mengkonversi barisan karakter angka di dalam string s menjadi nilai
  integer. Versi 1 }
{ K.Awal: s sudah berisi berisi barisan karakter angka ('0'..'9') }
{ K.Akhir: n berisi representasi integer dari string s }

(* DEKLARASI *)
var
   i       : integer;   { indeks slarik }
   pengali : integer;   { faktor pengali }

   function KarKeInt(x : char):integer;
   { mengkonversi karakter digit ('0'..'9') ke integer (0..9) }

(* ALGORITMA: *)
begin
   n := 0;
   for i := 1 to Length(s) do
      n := 10 * n + KarKeInt(s[i]);
   {endfor}
end;
```

Algoritma 13.17 Konversi *string* s menjadi *integer* (versi 2).

Fungsi KarKeInt:

```
function KarKeInt(c : char) : integer;
{ mengkonversi karakter digit ('0'..'9') ke integer (0..9) }

(* ALGORITMA: *)
begin
```

```
  case c of
    '0' : KarKeInt := 0;
    '1' : KarKeInt := 1;
    '2' : KarKeInt := 2;
    '3' : KarKeInt := 3;
    '4' : KarKeInt := 4;
    '5' : KarKeInt := 5;
    '6' : KarKeInt := 6;
    '7' : KarKeInt := 7;
    '8' : KarKeInt := 8;
    '9' : KarKeInt := 9;
  end;
end;
```

Algoritma 13.18 Fungsi konversi karakter ke *integer*.

Bahasa *Pascal* dan bahasa *C* menyediakan prosedur standar untuk konversi *string* ke representasi numeriknya, yaitu Val, sedangkan di dalam bahasa *C* berupa fungsi yaitu atoi.

Contoh Masalah 13.9: Membalikkan *string*

Misalkan sebuah *string s* telah berisi rangkaian karakter. Tulislah prosedur dalam bahasa *Pascal* dan bahasa *C* untuk membalikkan susunan karakter di dalam *s*. Misalnya jika *s* berisi *string* 'sate', maka hasil pembalikan *s* adalah *string* 'etas'.

Penyelesaian

PASCAL:

```
procedure Balikkan(var s : string);
{ Membalikkan susunan karakter di dalam string s }
{ K.Awal: string s sudah terdefinisi. }
{ K.Akhir: susunan karakter di dalam s kebalikan dari susunan semula. }

(* DEKLARASI *)
var
  i, j : integer;      { indeks larik }
  temp : char;

(* ALGORITMA: *)
begin
  i := 1;
  j := Length(s);     /* panjang string s */
  while (i <= j) do
   begin
    { pertukarkan s[i] dan s[j] }
    temp := s[i];
    s[i] := s[j];
    s[j] := temp;

    i := i + 1;
    j := j - 1;
   end;
end;
```

Algoritma 13.19 Membalikkan susunan karakter di dalam *string* (versi Pascal).

C:

```
void Balikkan(char s[])
/* Membalikkan susunan karakter di dalam string s */
/* K.Awal: string s sudah terdefinisi. */
/* K.Akhir: susunan karakter di dalam s kebalikan dari susunan semula. */

{

/* DEKLARASI */
  int i, j;      /* indeks larik */
  char temp;

/* ALGORITMA: */
  i = 0;
  j = strlen(s) - 1;     /* indeks elemen paling kanan */
  while (i <= j)
   {
    /* pertukarkan s[i] dan s[j] */
    temp = s[i];
    s[i] = s[j];
    s[j] = temp;

    i = i + 1;
    j = j - 1;
   }
}
```

Algoritma 13.20 Membalikkan susunan karakter di dalam string (versi C).

Contoh Masalah 13.10: Membaca data dari arsip

Misalkan N buah titik-titik (x, y) disimpan di dalam sebuah arsip teks. Susunan data di dalam arsip tersebut adalah sebagai berikut: baris pertama berisi nilai N, sedangkan baris-baris berikutnya berisi data setiap titik, yang dalam hal ini setiap baris terdiri atas nilai x dan nilai y, masing-masing nilai dipisahkan oleh spasi. Sebagai contoh, misalkan arsip data.txt sudah berisi data yang telah disusun sesuai dengan format yang ditetapkan di atas:

```
5
6 5
7 2
3 8
1 4
8 12
```

Baris pertama, 5, menyatakan banyaknya titik, sedangkan baris-baris berikutnya berisi 5 buah titik (x, y), di mana setiap baris terdiri atas nilai x dan y; misalnya baris pertama adalah titik (6, 5), baris kedua titik (7, 2), dan seterusnya. Tulislah dua buah prosedur, masing-masing dalam bahasa *Pascal* dan bahasa *C*; prosedur pertama membaca data dari arsip dan menyimpan titik-titik ke dalam larik P; sedangkan prosedur kedua mencerminkan setiap titik di dalam larik terhadap sumbu-y (dengan pencerminan ini maka titik (a, b) menjadi $(-a, b)$. Titik-titik hasil pencerminan disimpan di dalam larik Q.

Penyelesaian

PASCAL:

Deklarasi tipe Titik dan LarikTitik di dalam program utama:

```
const Nmaks = 100;     { ukuran maksimum larik }
type Titik = record
               y : integer;
               y : integer;
             end;

type LarikTitik = array[1..Nmaks] of Titik;
```

Prosedur membaca data dari arsip:

```
procedure BacaTitikDariArsip(NamaArsip : string[12]; var P : LarikTitik;
                             var N : integer)
{ Membaca data titik-titik dari arsip NamaArsip, dan menyimpannya ke
  dalam larik P yang bertipe LarikTitik. N akan berisi banyaknya titik.}

(* DEKLARASI *)
var
  i     : integer;  { pencacah banyak data }
  Fin   : text;     { pointer ke arsip masukan }

(* ALGORITMA: *)
begin
  { buka arsip masukan }
  assign(Fin, NamaArsip);
  reset(Fin);

  read(Fin, N);    {baca banyaknya titik }
  for i:=1 to N do
    read(Fin, P[i].x, P[i].y);
  end;{for}
end;
```

Prosedur pencerminan titik terhadap sumbu-*y*:

```
procedure PencerminanSumbu_Y(P : LarikTitik; N : integer;
                             var Q : LarikTitik)
{ Mencerminkan titik-titik di dalam larik P[1..N] terhadap sumbu-y.
Hasil pencerminan disimpan di dalam larik Q. }

(* DEKLARASI *)
var
  i : integer;

(* ALGORITMA: *)
begin
  for i:=1 to N do
   begin
     Q[i].x := -P[i].x;
     Q[i].y := P[i].y;
   end;{for}
end;
```

C:

Deklarasi tipe Titik dan LarikTitik di dalam program utama:

```
#define Nmaks 100      /* ukuran maksimum larik */
typedef struct {int x; int y;} Titik;
typedef Titik LarikTitik[Nmaks + 1];
```

Prosedur membaca data dari arsip:

```
void BacaTitikDariArsip(char NamaArsip[], LarikTitik P, int *N)
/* Membaca data titik-titik dari arsip NamaArsip, dan menyimpannya ke
   dalam larik P yang bertipe LarikTitik. N akan berisi banyaknya titik.*/

{
  /* DEKLARASI */
   int i;        /* pencacah banyak data */
   FILE *Fin;    /* pointer ke arsip masukan */

  /* ALGORITMA: */

  /* buka arsip masukan */
  Fin = fopen(NamaArsip, "r");

  fscanf(Fin, "%d", &(*N));  /* baca banyaknya data */
  for (i=1; i<=N; i++)
    fscanf(Fin, "%d %d", &P[i].x, &P[i].y);
  /*endfor*/
}
```

Prosedur pencerminan titik terhadap sumbu-*y*:

```
void PencerminanSumbu_Y(LarikInt P, LarikTitik Q, int N)
/* Mencerminkan titik-titik di dalam larik P[1..N] terhadap sumbu-y.
   Hasil pencerminan disimpan di dalam larik Q. */

{
 /* DEKLARASI */
 int i;

/* ALGORITMA: */
 for (i=1; i<=N; i++)
  {
    Q[i].x = -P[i].x;
    Q[i].y = P[i].y;
  }
}
```

Contoh-contoh program *Pascal/C* di atas dapat dimodifikasi sedemikian sehingga penanda akhir data adalah sebuah nilai khusus (seperti 9999), atau akhir data diidentifikasi dengan akhir arsip (menggunakan fungsi *EOF* atau *feof*). Lihat contoh hal ini di dalam upa-bab 7.7.

14
Matriks

Elemen dari tipe terstruktur seperti larik dapat distrukturkan lagi. Sebuah larik yang setiap elemennya adalah larik lagi disebut matriks (*matrix*) [WIR76]. Matriks sudah dikenal secara luas dalam berbagai bidang ilmu, terutama dalam bidang matematika. Matriks identitas adalah contoh matriks yang dikenal secara umum, karena semua elemen diagonalnya 1, yang lainnya 0,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sebuah sistem persamaan lanjar yang terdiri dari empat persamaan berikut,

$$\begin{aligned} 2x_1 - 6x_2 + 4x_3 + 12x_4 &= -10 \\ 3x_1 + 2x_2 - 8x_3 - 8x_4 &= 12 \\ x_1 + 10x_2 + 8x_3 - 7x_4 &= 8 \\ 4x_1 - 5x_2 - 9x_3 + 11x_4 &= 21 \end{aligned}$$

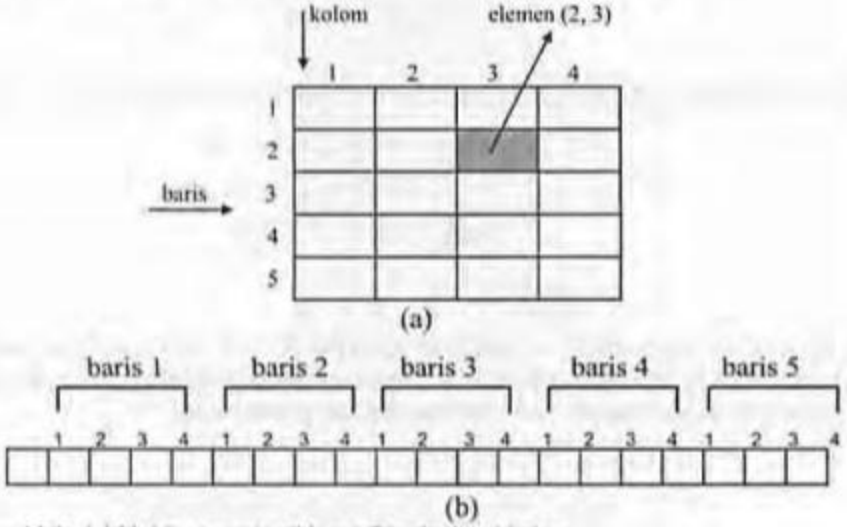
lazim ditulis dalam bentuk persamaan matriks $Ax = b$, yang dalam hal ini A adalah matriks koefisien dari ruas kiri persamaan, x adalah matriks kolom peubah dan b adalah matrik kolom ruas kanan persamaan:

$$\begin{bmatrix} 2 & -6 & 4 & 12 \\ 3 & 2 & -8 & -8 \\ 1 & 10 & 8 & -7 \\ 4 & -5 & -9 & 11 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} -10 \\ 12 \\ 8 \\ 21 \end{bmatrix}$$

Masih banyak matriks lain yang terdapat di dalam bidang matematika, seperti matriks transformasi, matriks permutasi, dan sebagainya. Matriks juga digunakan secara luas pada bidang seperti ekonomi, rekayasa, dan lain-lain. Di dalam Bab 14 ini kita akan mempelajari struktur matriks dan beberapa algoritma pemrosesan matriks.

14.1 Definisi Matriks

Matriks adalah struktur penyimpanan data di dalam memori utama yang setiap individu elemennya diacu dengan menggunakan dua buah indeks (yang biasanya dikonotasikan dengan indeks baris dan indeks kolom). Sebagaimana pada larik, di dalam buku ini kita menggambarkan matriks sebagai sekumpulan kotak yang tersusun berjajar pada setiap baris dan kolom. Susunan bujursangkar seperti itu sering dinamakan juga *grid*. Gambar 14.1(a) memperlihatkan matriks yang terdiri dari 5 buah baris dan 4 buah kolom. Kita katakan matriks tersebut berukuran 5 × 4. Baris matriks tersebut diacu dengan indeks 1, 2, 3, 4, dan 5, sedangkan kolom matriks diacu dengan indeks 1, 2, 3, dan 4. Elemen yang diarsir menyatakan elemen pada baris ke-2 dan kolom ke-3 . Karena adanya dua buah indeks tersebut, matriks disebut juga larik **dwimatra** (matra = *dimension*).



Gambar 14.1 (a) Matriks yang terdiri atas 5 baris dan 4 kolom
(b) Representasi matriks 5 × 4 di dalam memori sebagai larik.

Meskipun matriks merupakan larik dwimatra, namun representasinya di dalam memori tetaplah sebagai deretan sel berturutan. Gambar 14.1(b) memperlihatkan representasi matriks berukuran 5 × 4 di dalam memori. Angka 1, 2, 3, dan 4 menyatakan indeks elemen relatif pada setiap "baris" (apa yang dimaksud dengan baris di sini adalah kelompok 4 buah elemen).

Karena matriks sebenarnya adalah larik, maka konsep umum dari larik juga berlaku untuk matriks yaitu:

1. Kumpulan elemen yang bertipe sama. Tipe elemen matriks dapat berupa tipe dasar (*integer*, *real*, *boolean*, *char*, dan *string*), atau tipe tersrtuktur seperti *record*.
2. Setiap elemen data dapat diakses secara langsung jika indeksnya (baris dan kolom) diketahui, yang dalam hal ini indeks menyatakan posisi relatif di dalam kumpulannya.
3. Merupakan struktur data yang statis, artinya jumlah elemennya sudah dideklarasikan terlebih dahulu dan tidak bisa diubah selama pelaksanaan program.

Struktur matriks praktis untuk dipakai (pengaksesannya cepat karena langsung melalui indeksnya) tetapi memakan banyak tempat di memori. Misalnya matriks *integer* berukuran 100 × 100, membutuhkan 10000 × lebar *integer*. Jika satu representasi *integer* = 2 *byte*, maka dibutuhkan memoeri sebesar 200000 *byte*!

Karena matriks adalah struktur statis, maka ukuran matriks harus sudah diketahui sebelum pelaksanaan program (lihat upabab 14.3 Pendeklarasian Matriks). Sebagaimana pada larik biasa, kita menuliskan matra matriks sebagai penomoran indeks baris dan indeks kolom, mulai dari indeks terendah sampai indeks tertinggi. Sebagai contoh, misalkan matriks pada Gambar 14.1 bernama *M*, maka *M*[1..5, 1..4] menyatakan matriks berukuran 5 × 4, dengan indeks baris dari 1 sampai 5 dan indeks kolom dari 1 sampai 4. Indeks matriks tidak harus dimulai dari 1, tetapi juga boleh dari 0 atau dari bilangan negatif, seperti *M*[0..4, 1..3], *M*[-2..2, 0..3]. Bahkan, indeks juga dapat bertipe karakter, seperti *M*['a'..'e', 1..4] atau tipe lainnya asalkan mempunyai keterurutan.

Di atas sudah kita katakan bahwa elemen matriks diacu melalui indeks baris dan indeks kolomnya. Jika indeks baris dinyatakan dengan *i* dan indeks kolom dinyatakan dengan *j*, maka notasi algoritmik untuk mengacu elemen pada baris *i* dan kolom *j* adalah

nama_matriks[i, j]

Jika matriks pada Gambar 14.1 bernama *M*, maka elemen yang diarsir pada gambar itu adalah *M*[2, 3]. Cara mengacu dengan peubah indeks hanya benar apabila nilai dari peubah indeks tersebut sudah terdefinisi, misalnya

M[*p*, 2], *M*[1, *k*], *M*[*p*, *q*]

hanya benar jika p, k, dan q sudah terdefinisi nilainya.

Matriks pada Gambar 14.1 dapat digambarkan kembali dengan notasi setiap elemen-elemennya pada Gambar 14.2.

	1	2	3	4
1	$M[1,1]$	$M[1,2]$	$M[1,3]$	$M[1,4]$
2	$M[2,1]$	$M[2,2]$	$M[2,3]$	$M[2,4]$
3	$M[3,1]$	$M[3,2]$	$M[3,3]$	$M[3,4]$
4	$M[4,1]$	$M[4,2]$	$M[4,3]$	$M[4,4]$
5	$M[5,1]$	$M[5,2]$	$M[5,3]$	$M[5,4]$

Gambar 14.2 Matriks *M* dengan notasi setiap elemen-elemennya.

Di bawah ini kita berikan beberapa contoh matriks yang lain, misalnya:

1. *C* [1..300, 1..200]: matriks yang merepresentasikan derajat keabuan (*greyscale*) pada citra digital (*digital image*) *monochrome*. Citra atau gambar digital berbentuk persegi panjang. Setiap elemen matriks menyatakan satu *pixel* (*picture element*), jadi citra *C* tersebut berukuran 300 × 200 *pixel*. Nilai di dalam setiap elemen matriks menyatakan derajat keabuan *pixel*, yang berkisar dari 0 sampai 255. Representasi citra digital dengan matriks lazim digunakan pada bidang Pengolahan Citra (*Image Processing*) dan Pengenalan Pola (*Pattern Recognition*).

	1	2	3	4	5	6	...	200
1	12	0	15	6	128	34	...	0
2	45	35	200	12	0	17	...	122
3	6	47	203	53	20	18	...	201
4	7	60	67	87	0	90	...	127
5	89	89	89	64	0	76	...	0
6	23	39	87	93	0	32	...	178
...	...	...	...	...	...	...	...	...
300	120	129	101	0	0	0	...	12

2. *Hub*[1..4, 1..4]: matriks yang menyatakan hubungan diplomatik antara negara-negara. Elemen matriks bernilai *true* berarti negara *i* dan negara *j* mempunyai hubungan diplomatik, sebaliknya bernilai *false* jika tidak ada hubungan diplomatik.

	Indonesia	Portugal	Jepang	Rwanda
Indonesia	-	*false*	*true*	*false*
Portugal	*false*	-	*true*	*true*
Jepang	*true*	*true*	-	*true*
Rwanda	*false*	*true*	*true*	-

3. *Nilai*[1..6, 1..3]: matriks yang menyatakan nilai ujian 6 orang mahasiswa untuk tiga buah mata kuliah.

	Kimia	Fisika	Biologi
Andi Saptono	A	B	B
Hadi Setyono	A	D	B
Mas Rizal	A	B	A
Rizka Kurniawan	C	C	B
Henry	B	B	B
Wikan Danar	A	C	C

14.2 Pendeklarasian Matriks

Sebelum matriks digunakan untuk menyimpan data, terlebih dahulu matriks harus dideklarasikan. Mendeklarasikan matriks artinya menentukan nama matriks, tipe data, dan ukurannya. Pendeklarasian matriks di dalam teks algoritma ditulis di dalam bagian DEKLARASI. Kata kunci yang dipakai untuk matriks tetap *array*.

Ada beberapa cara pendeklarasian matriks:

1. Sebagai nama peubah.

```
DEKLARASI
    M : array [1..5, 1..4] of integer
```

Algoritma 14.1 Deklarasi sebuah matriks *integer*

2. Sebagai tipe.

```
DEKLARASI
    type Mat : array[1..5, 1..4] of integer
    M : Mat    { M adalah nama matriks yang bertipe Mat }
```

Algoritma 14.2 Deklarasi matriks sebagai tipe bentukan

3. Mendefinisikan ukuran maksimum matriks sebagai sebuah konstanta.

```
DEKLARASI
    const NbarisMaks = 20    { jumlah baris maksimum }
    const NkolomMaks = 20    { jumlah kolom maksimum }
    M : array [1..NbarisMaks, 1..NKolomMaks] of integer
```

Algoritma 14.3 Deklarasi matriks dengan ukuran maksimum sebagai konstanta

Dengan mendeklarasikan matriks seukuran maksimum di atas (Algoritma 14.3), ukuran matriks yang dibutuhkan dapat ditentukan pada saat pelaksanaan algoritma, asalkan jumlah baris dan jumlah kolom yang ditentukan tidak melebihi ukuran maksimum matriks (*NbarisMaks* dan *NkolomMaks*). Misalnya kita mendefinisikan ukuran matriks di dalam algoritma dengan instruksi berikut:

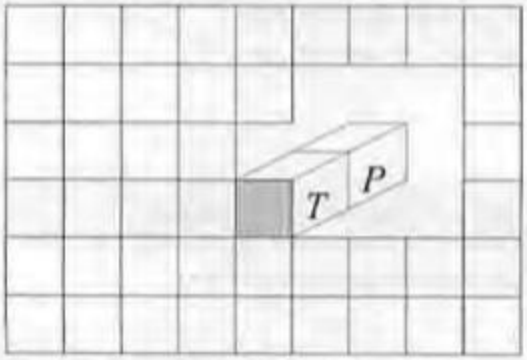
```
read(JumlahBaris)      { asalkan JumlahBaris ≤ NbarisMaks }
read(JumlahKolom)      { asalkan JumlahKolom ≤ NkolomMaks }
```

Seperti yang sudah disebutkan pada bagian terdahulu, elemen matriks dapat bertipe terstruktur. Misalkan kita memiliki sekumpulan data yang berupa pasangan temperatur (T) dan tekanan (P) pada titik-titik diskrit di sebuah permukaan logam. Seluruh data ini direpresentasikan di dalam matriks B dengan pendeklarasian sebagai berikut:

```
DEKLARASI
   { Matriks temperatur-tekanan }

   const NBarisMaks = ...   { jumlah baris matriks }
   const NKolomMaks = ...   { jumlah kolom matriks }

   type TP : record <T:real, P:real>   { tipe elemen matriks }

   B : array [1..NbarisMaks, 1..NKolomMaks] of TP
```

Algoritma 14.4 Deklarasi matriks terstruktur

Setiap elemen matriks B merupakan sebuah *record* yang terdiri atas *field* T dan *field* P. Matriks B dapat dibayangkan seperti pada Gambar 14.14. Elemen yang diarsir adalah sebuah *record* yang terdiri dari dua buah kotak (T dan P) yang disusun ke belakang.



Gambar 14.3 Matriks B dengan notasi setiap elemen-elemennya.

Berikut ini contoh mengacu elemen matriks *B* pada Algoritma 14.4 di atas:

B[1, 2] → mengacu satu elemen matriks
B[1, 2].*T* → mengacu temperatur pada koordinat (1,2)
B[1, 2].*P* → mengacu tekanan pada koordinat (1,2)

14.3 Pemrosesan Matriks

Algoritma pemrosesan matriks pada umumnya adalah memanipulasi elemen-elemen matriks. Pemrosesan matriks adalah proses beruntun (sekuensial). Setiap elemen matriks "dikunjungi" (*traversal*) lalu dilakukan aksi terhadap elemen tersebut.

Di dalam upabab ini diberikan beberapa contoh algoritma pemrosesan matriks. Untuk semua algoritma yang diberikan, kita sepakati menggunakan deklarasi matriks seperti berikut ini:

```
DEKLARASI
  const NbarisMaks = 20  { jumlah baris maksimum }
  const NkolomMaks = 20  { jumlah kolom maksimum }

  type MatriksInt : array[1..NbarisMaks, 1..NKolomMaks] of integer
```

Algoritma 14.5 Deklarasi matriks yang digunakan di dalam bab ini

Karena secara lojik elemen matriks disusun dalam susunan baris dan kolom, maka pemrosesan matriks dilakukan per baris dan per kolom. Karena ada dua buah indeks, maka kita harus memutuskan indeks mana yang "nilainya tetap" selama indeks yang lain "dijalankan". Cara pemrosesan yang umum adalah menelusuri matriks baris per baris, yang pada setiap baris melakukan proses terhadap elemen pada setiap kolomnya (kita namakan pemrosesan "per baris-per kolom"). Karena ada dua indeks, maka algoritma pemrosesan matriks umumnya kalang bersarang (*nested loop*), satu kalang untuk tiap indeks. Kalang terluar digunakan untuk memproses baris, sedangkan kalang terdalam untuk memproses kolom-kolom pada baris yang sedang diacu.

Contoh 14.1. Mengunjungi elemen-elemen matriks. Tinjaulah matriks *M*[1..5, 1..4] di bawah ini, dengan *i* adalah indeks baris dan *j* adalah indeks kolom.

	1	2	3	4
1	10	4	45	30
2	20	16	18	23
3	29	98	19	27
4	2	15	26	65
5	21	8	13	7

Algoritma pemrosesan "per baris-per kolom":

Untuk setiap baris i = 1, 2, 3, 4, 5 lakukan:
  untuk setiap kolom i = 1, 2, 3, 4 lakukan:
    proses $M[i, j]$

Hasil pemrosesan:
i = 1, elemen yang diproses berturut-turut: 10, 4, 45, 30
i = 2, elemen yang diproses berturut-turut: 20, 16, 18, 23
i = 3, elemen yang diproses berturut-turut: 29, 98, 19, 27
i = 4, elemen yang diproses berturut-turut: 2, 15, 26, 65
i = 5, elemen yang diproses berturut-turut: 21, 8, 13, 7

Skema umum pemrosesan matriks dengan gaya per baris-per kolom dinyatakan dengan *pseudo-code* algoritma berikut ini:

```
procedure ProsesMatriks1(input M : MatriksInt, input Nbar,
Nkol:integer)
{ Pemrosesan elemen matriks M[1..Nbar,1..Nkol] per baris per kolom. }
{ K.Awal : Matriks M sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Setiap elemen matriks M telah diproses. }

DEKLARASI
  i : integer  { indeks baris }
  j : integer  { indeks kolom }

ALGORITMA:
  for i ← 1 to Nbar do
     for j ← 1 to Nkol do
        Proses(M[i,j])
     endfor
  endfor
```

Algoritma 14.6 Skema umum pemrosesan matriks (per baris-per kolom)

Keterangan

Proses (M[i,j]) adalah operasi yang memanipulasi nilai $M[i, j]$, seperti operasi I/O (baca, tulis), komputasi, pengubahan, dan sebagainya.

Selain struktur *for*, struktur pengulangan *while* dan *repeat* dapat juga digunakan untuk pemrosesan matriks. Skema pemrosesan matriks dengan menggunakan kedua struktur pengulangan tersebut ditunjukkan pada Algoritma 14.7 dan Algoritma 14.8.

```
procedure ProsesMatriks1(input M : MatriksInt, input Nbar,
Nkol:integer)
{ Pemrosesan elemen matriks M[1..Nbar,1..Nkol] per baris per kolom. }
{ K.Awal : Matriks M sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Setiap elemen matriks M telah diproses. }

DEKLARASI
  i : integer  { indeks baris }
```

```
  j : integer  { indeks kolom }
ALGORITMA:
  i ← 1
  while i ≤ Nbar do
     j ← 1
     while j ≤ Nkol do
        Proses(M[i,j])
        j ← j + 1      { kolom berikutnya }
     endwhile
     { j > Nkol }
     i ← i + 1      { baris berikutnya }
  endwhile
  { i > Nbar }
```

Algoritma 14.7 Skema umum pemrosesan matriks (per baris-per kolom) dengan struktur *while*

```
procedure ProsesMatriks1(input M : MatriksInt, input Nbar,
Nkol:integer)
{ Pemrosesan elemen matriks M[1..Nbar,1..Nkol] per baris per kolom. }
{ K.Awal : Matriks M sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Setiap elemen matriks M telah diproses. }

DEKLARASI
  i : integer  { indeks baris }
  j : integer  { indeks kolom }

ALGORITMA:
  i ← 1
  repeat
     j ← 1
     repeat
        Proses(M[i,j])
        j ← j + 1      { kolom berikutnya }
     until j > Nkol
     i ← i + 1      { baris berikutnya }
  until i > Nbar
```

Algoritma 14.8 Skema umum pemrosesan matriks (per baris-per kolom) dengan struktur *repeat*

Adakalanya kita memerlukan pemrosesan matriks kolom demi kolom dan pada setiap kolom kita memproses setiap barisnya (kita namakan pemrosesan "per kolom-per baris"). Kalang terluar digunakan untuk memproses kolom, sedangkan kalang terdalam untuk memproses elemen-elemen pada baris yang sedang diacu.

Algoritma pemrosesan baris "per kolom-per baris":

Untuk setiap kolom $j = 1, 2, 3, 4$ lakukan:
  untuk setiap baris $i = 1, 2, 3, 4, 5$ lakukan:
    proses $M[i, j]$

Hasil penelusuran:
j = 1, elemen yang diproses berturut-turut: 10, 20, 29, 2, 21
j = 2, elemen yang diproses berturut-turut: 4, 16, 98, 15, 8
j = 3, elemen yang diproses berturut-turut: 45, 18, 19, 26, 13
j = 4, elemen yang diproses berturut-turut: 30, 23, 27, 65, 7

Skema umum pemrosesan matriks dengan gaya per kolom-per baris dinyatakan dengan *pseudo-code* Algoritma 14.9 berikut ini:

```
procedure ProsesMatriks2(input M : MatriksInt, input Nbar,
Nkol:integer)
{Pemrosesan elemen matriks M[1..Nbar, 1..Nkol] per kolom per baris. }
{ K.Awal : Matriks M sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Setiap elemen matriks M telah diproses  }

DEKLARASI
 i : integer  { indeks baris }
 j : integer  { indeks kolom }


ALGORITMA:
  for j ← 1 to Nkol do
     for i ← 1 to Nbar do
        Proses(M[i,j])
     endfor
 endfor
```

Algoritma 14.9 Skema umum pemrosesan matriks (per kolom-per baris)

Perhatikan Algoritam 14.9 di atas, bahwa untuk pemrosesan per kolom per baris ini, kita tidak menulis elemen matriks sebagai `M[j,i]`, tetapi tetap sebagai `M[i,j]`, sebab bila kita tuliskan `M[j,i]` hasilnya sama saja dengan penelusuran per baris-per kolom (coba periksa!).

Di dalam buku ini kita menggunakan skema penelusuran per baris-per kolom karena skema inilah yang lazim digunakan (lebih alami dibandingkan skema per kolom-per baris). Kita akan memberikan contoh-contoh algoritma pemrosesan matriks pada beberapa upa-bab di bawah ini.

1. Menginisialisasi Matriks

Menginisialisasi matriks artinya memberi nilai awal yang sama untuk seluruh (atau sebagian) elemen matriks. Proses inisialisasi biasanya dilakukan sebelum matriks digunakan untuk perhitungan. Misalnya dalam algoritma perkalian matriks $A \times B = C$, seluruh elemen matriks C harus diisi dengan nilai 0 karena C akan digunakan untuk menampung hasil penjumlahan yang berulang (*iteration*) -lihat pada upa-bab Latihan Soal mengenai perkalian matriks. Namun, tidak semua matriks perlu diinisialisasi, karena itu proses ini tidak selalu harus dilakukan. Misalnya jika elemen matriks dibaca dari piranti masukan, maka inisialisasi matriks tidak perlu dilakukan.

Algoritma inisialisasi matriks berikut ini mengisi setiap elemen matriks dengan 0 (Anda dapat menggunakan nilai yang lain selain 0).

```
procedure InisMatriks1(input/output M : MatriksInt,
                       input Nbar, Nkol : integer)
{ Menginisialisasi seluruh elemen matriks M[1..Nbar, 1..Nkol] dengan 0.}
{ K.Awal : Nbar dan Nkol sudah terdefinisi dengan banyaknya baris dan
  kolom matriks }
{ K.Akhir: M[I,J] = 0 untuk I = 1..Nbar, J = 1..Nkol }

DEKLARASI
 i : integer  { indeks baris }
 j : integer  { indeks kolom }

DESKRIPSI:
  for i ← 1 to Nbar do
     for j ← 1 to Nkol do
         M[i, j] ← 0
     endfor
  endfor
```

Algoritma 14.10 Menginisialisasi seluruh elemen matriks dengan 0

2. Membaca Elemen Matriks

Membaca elemen matriks artinya mengisi elemen-elemen matriks dengan data dari piranti masukan, misalnya papan ketik. Yang harus diperhatikan selama pembacaan dari papan ketik adalah urutan pembacaan data, karena data disimpan di dalam matriks sesuai dengan urutan pembacaan. Untuk menghindari kesalahan pengisian elemen, sebaiknya pada waktu pembacaan data diberikan keterangan/petunjuk yang menginformasikan indeks baris dan indeks kolom elemen yang akan diisi. Bila secara implisit sudah diketahui urutan pembacaan adalah per baris per kolom, maka petunjuk pembacaan boleh tidak diberikan.

Contoh 14.2. Membaca elemen-elemen matriks. Misalkan urutan data yang dibaca (per baris per kolom) sebagai berikut:

12 45 16 9 67 15 8 1 10 15 1 9 12 7 0 3 21 56 13 19

maka matriks *M* akan berisi nilai-nilai sebagai berikut:

	1	2	3	4
1	12	45	16	9
2	67	15	15	1
3	10	15	1	9
4	12	7	0	3
5	21	56	13	19

Algoritma pembacaan matriks dari papan ketik kita bedakan dua macam, yaitu pengisian yang menggunakan petunjuk pembacaan dan pengisian

tanpa menggunakan petunjuk pembacaan (secara implisit sudah diketahui pembacaannya per baris per kolom)

1. Pengisian matriks tanpa petunjuk pembacaan

```
procedure BacaMatriks1(input/output M : MatriksInt,
                       input Nbar, Nkol : integer)
{ Mengisi elemen matriks M[1..Nbar, 1..Nkol] dari papan ketik tanpa
menggunakan petunjuk pembacaan.
{ K.Awal : Nbar dan Nkol sudah terdefinisi dengan banyaknya baris dan
  kolom matriks }
{ K.Akhir: Seluruh elemen matriks M sudah berisi nilai yang dibaca
dari papan ketik. }
DEKLARASI
i : integer  { indeks baris }
j : integer  { indeks kolom }
ALGORITMA:
 for i ← 1 to Nbar do
    for j ← 1 to Nkol do
       read (M[i, j])
    endfor
 endfor
```

Algoritma 14.11 Membaca elemen-elemen matriks tanpa petunjuk pembacaan

2. Pengisian mariks dengan menggunakan petunjuk pembacaan

```
procedure BacaMatriks2(input/output M : MatriksInt,
                       input Nbar, Nkol : integer)
{Mengisi elemen matriks M[1..Nbar, 1..Nkol] dari papan ketik dengan
 menggunakan petunjuk pembacaan. }
{ K.Awal : Nbar dan Nkol sudah terdefinisi dengan banyaknya baris dan
  kolom matriks }
{ K.Akhir: Seluruh elemen matriks M sudah berisi nilai yang dibaca
dari piranti masukan }

DEKLARASI
i : integer  { indeks baris }
j : integer  { indeks kolom }

ALGORITMA:
 for i ← 1 to Nbar do
    for j ← 1 to Nkol do
      write('Ketikkan elemen M[',i,',',j,'] : ') { petunjuk pembacaan}
       read (M[i,j])
    endfor
 endfor
```

Algoritma 14.12 Membaca elemen-elemen matriks dengan petunjuk

Misalkan ketika *i* dan *j* masing-masing bernilai 2 dan 3, maka di layar peraga akan muncul tulisan

```
Ketikkan elemen M[2, 3]:
```

sehingga pengguna program paham bahwa elemen matriks yang akan diisi nilainya adalah elemen $M[2, 3]$.

3. Menulis Matriks

Menulis matriks artinya mencetak elemen-elemen matriks ke piranti keluaran (misalnya ke layar peraga, ke dalam arsip, atau ke *printer*) dengan asumsi bahwa elemen matriks sudah terdefinisi nilainya (misalnya sudah diisi melalui proses pembacaan atau hasil dari suatu manipulasi matriks lain).

```
procedure TulisMatriks(input M : MatriksInt,
                       input Nbar, Nkol : integer)
{ Mencetak elemen matriks M[1..Nbar, 1..Nkol] ke piranti keluaran. }
{ K.Awal : elemen-elemen matriks sudah terdefinisi harganya. }
{ K.Akhir: seluruh elemen matriks tertulis ke piranti keluaran. }

DEKLARASI
i : integer   { indeks baris }
 j : integer   { indeks kolom }

ALGORITMA:
 for i ← 1 to Nbar do
    for j ← 1 to Nkol do
       write(M[i,j])
    endfor
 endfor
```

Algoritma 14.13 Menulis elemen-elemen matriks

4. Penjumlahan Dua Buah Matriks

Penjumlahan dua buah matriks A dan B menghasilkan matriks C, atau $A + B = C$, hanya dapat dilakukan bila ukuran matriks A dan ukuran matriks B sama dan kedua matriks sudah terdefinisi nilai-nilainya. Matriks C akan berukuran sama dengan matriks A dan B.

Penjumlahan matriks A dan B didefinisikan sebagai berikut:

$$C[i,j] = A[i,j] + B[i,j] \text{ untuk semua } i \text{ dan } j.$$

Contoh 14.3. Penjumlahan dua buah matriks.

Matriks A + Matriks B = Matriks C

$$\begin{bmatrix} 1 & 3 & 5 \\ 18 & 31 & 1 \\ 2 & 4 & 15 \end{bmatrix} + \begin{bmatrix} -6 & 9 & 8 \\ 3 & 2 & 1 \\ 4 & 5 & -7 \end{bmatrix} = \begin{bmatrix} -5 & 12 & 13 \\ 21 & 33 & 2 \\ 6 & 7 & -7 \end{bmatrix}$$

Algoritma penjumlahan dua buah matriks:

```
procedure JumlahDuaMatriks(input A, B : MatriksInt, Nbar, Nkol:integer
                           output C : Matriks,)
{ Menjumlahkan matriks A dan B, yaitu A + B = C. }
{ K.Awal : Matriks A dan B sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Matriks C berisi hasil penjumlahan A dan B }

DEKLARASI
  i : integer  { indeks baris }
  j : integer  { indeks kolom }


ALGORITMA:
  for i ← 1 to Nbar do
     for j ← 1 to Nkol do
        C[i,j] ← A[i,j] + B[i,j]
     endfor
  endfor
```

Algoritma 14.14 Menjumlahkan dua buah matriks

14.4 Translasi Notasi Algoritmik Matriks ke dalam Bahasa PASCAL dan Bahasa C

ALGORITMIK

```
DEKLARASI
  const NbarisMaks = 20  { jumlah baris maksimum }
  const NkolomMaks = 20  { jumlah kolom maksimum }
  type MatriksInt : array[1..NbarisMaks, 1..NkolomMaks] of integer
  M : Matriks
```

PASCAL

```
const
  NbarisMaks = 20;  { jumlah baris maksimum }
  NkolomMaks = 20;  { jumlah kolom maksimum }
type
  MatriksInt = array[1..NbarisMaks, 1..NkolomMaks] of integer;
var
  M : MatriksInt;
```

C

```
#define NbarisMaks   20  /* jumlah baris maksimum */
#define NkolomMaks   20  /* jumlah kolom maksimum */
typedef int MatriksInt [NbarisMaks+1][NkolomMaks+1];
MatriksInt M;
```

Perhatikan bahwa dalam bahasa *C* indeks larik/matriks selalu dimulai dari 0. Oleh karena itu, agar batas maksimum indeks baris dan batas maksimum indeks kolom sama seperti pada notasi algoritmik dan bahasa *PASCAL* (dimulai dari 1), maka jumlah elemen baris dan jumlah elemen kolom dilebihkan satu.

Di bawah ini dituliskan beberapa contoh algoritma pemrosesan matriks yang sudah dijelaskan pada bagian sebelum ini dan sekarang diterjemahkan ke dalam bahasa *PASCAL* dan bahasa *C*.

1. Membaca Matriks

PASCAL

```
procedure BacaMatriks1(var M : MatriksInt; Nbar, Nkol : integer);
{Mengisi elemen matriks M[1..Nbar, 1..Nkol] dari papan ketik
(keyboard)  dengan menggunakan petunjuk pembacaan. }
{ K.Awal : Nbar dan Nkol sudah terdefinisi dengan banyaknya baris dan
  kolom matriks. }
{ K.Akhir: Seluruh elemen matriks M sudah berisi nilai yang dibaca
dari
  papan ketik. }

var
 i : integer;  { indeks baris }
  j : integer;  { indeks kolom }

begin
 for i := 1 to Nbar do
    for j := 1 to Nkol do
     begin
       write('Ketikkan elemen M[',i,',',j,'] : '); { petunjuk
                                                     pembacaan }
       readln(M[i,i]);
     end;{for}
 {endfor}
end;
```

Contoh pemanggilan prosedur di atas:

```
writeln('Baca ukuran matriks:');
write('Berapa jumlah baris (1 - 20)?'); readln(p);
write('Berapa jumlah kolom? (1 - 20)'); readln(q);
BacaMatriks1(M, p, q);
```

C

```
void BacaMatriks1(MatriksInt M, int Nbar, int Nkol)
/* Mengisi elemen matriks M[1..Nbar, 1..Nkol] dari papan ketik
(keyboard) dengan menggunakan petunjuk pembacaan. */
/* K.Awal : Nbar dan Nkol sudah terdefinisi dengan banyaknya baris dan
   kolom matriks. */
/* K.Akhir: Seluruh elemen matriks sudah berisi nilai yang dibaca dari
   papan ketik. */
```

```
{
int i;  /* indeks baris */
 int j;  /* indeks kolom */

 for (i = 1; i <= Nbar; i++)
   for (j = 1; j <= Nkol; j++)
    {
      printf("Ketikkan elemen M[%d][%d] : ", i, j); /* petunjuk baca */
      scanf(M[i][j]));
    }

 /* endfor*/
}
```

Contoh pemanggilan prosedur di atas:

```
printf("Baca ukuran matriks: \n");
printf("Berapa jumlah baris? (1 - 20) "); scanf(&p);
printf("Berapa jumlah kolom?(1 - 20) "); scanf(&q);
BacaMatriks1(M, p, q);
```

2. Menulis Matriks

PASCAL

```
procedure TulisMatriks(M : MatriksInt; Nbar, Nkol : integer);
{ Mencetak elemen matriks M[1..Nbar, 1..Nkol] ke layar. }
{ K.Awal : elemen matriks sudah terdefinisi harganya. }
{ K.Akhir: seluruh elemen matriks tertulis di layar.  }

var
i : integer;  { indeks baris }
 j : integer;  { indeks kolom }

begin
 for i := 1 to Nbar do
  begin
    for j := 1 to Nkol do
       write(M[i,j],'  ');
    {endfor}
    writeln; { pindahkan kursor ke awal baris berikutnya }
  end;{for}
end;
```

C

```
void TulisMatriks(MatriksInt M, int Nbar, int Nkol)
/* Mencetak elemen matriks M[1..Nbar, 1..Nkol] ke layar. */
/* K.Awal : elemen matriks sudah terdefinisi harganya. */
/* K.Akhir: seluruh elemen matriks tertulis di layar */

{
 int i;  /* indeks baris */
  int j;  /* indeks kolom */

  for (i = 1; i <= Nbar; i++)
   {
```

```
    for (j = 1; j <= Nkol; j++)
       printf("%d   ", M[i][j]);
    /*endfor*/
    printf("\n"); /* pindahkan kursor ke baris berikutnya */
   }
}
```

3. Penjumlahan Dua Buah Matriks

PASCAL

```
procedure JumlahDuaMatriks(A, B : MatriksInt; Nbar, Nkol : integer;
                           var C : MatriksInt);
{ Menjumlahkan matriks A dan B, yaitu A + B = C. }
{ K.Awal : Matriks A dan B sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Matriks C berisi hasil penjumlahan A dan B }

var
  i : integer;  { indeks baris }
  j : integer;  { indeks kolom }

begin
  for i := 1 to Nbar do
    for j := 1 to Nkol do
      C[i,j] := A[i,j] + B[i,j];
    {endfor}
 {endfor}
end;
```

C

```
void JumlahDuaMatriks(MatriksInt A, MatriksInt B, int Nbar, int Nkol,
                      MatriksInt C)
/* Menjumlahkan matriks A dan B, yaitu A + B = C. */
/* K.Awal : Matriks A dan B sudah terdefinisi elemen-elemennya. */
/* K.Akhir: Matriks C berisi hasil penjumlahan A dan B */

{
  int i;  /* indeks baris */
  int j;  /* indeks kolom */


 for (i = 1; i <= Nbar; i++)
     for (j = 1; j <= Nkol; j++)
       C[i][j] = A[i][j] + B[i][j];
     /*endfor*/
  /*endfor*/
}
```

14.5 Contoh-contoh Tambahan

Contoh 14.4. Membaca data matriks dari arsip masukan.
Adakalanya data untuk matriks dibaca dari arsip (*file*) karena ukuran matriks yang besar sehingga lebih mudah menyunting datanya di dalam arsip dengan program pengolah kata daripada dimasukkan satu per satu. Begitupun keluaran (*output*) program dari matriks juga sering ditulis ke arsip. Pembacaan data dari arsip bergantung pada format (cara meyimpan) data di dalam arsip. Misalkan kita asumsikan bahwa data masukan disimpan di dalam arsip dengan format teks (*text*). Antara satu data dengan data lainnya dipisahkan oleh separator spasi. Tipe peubah yang digunakan untuk menampung hasil pembacaan harus sama dengan tipe data yang dibaca; jika datanya bertipe *integer* maka peubah pembacaan juga harus bertipe *integer*; jika data riil maka peubahnya juga harus bertipe riil; begitu seterusnya. Untuk lebih jelasnya, silakan baca kembali upa-bab 5.3.

Misalkan seluruh data matriks yang berukuran $m \times n$ sudah disimpan di dalam arsip `data.txt`. Susunan data di dalam `data.txt` sebagai berikut: baris pertama berisi nilai m dan n, sedangkan baris-baris berikutnya berisi data *integer* sebanyak m baris dan n kolom, antara setiap data dipisahkan oleh spasi. Sebagai contoh, misalkan arsip `data.txt` sudah berisi data yang telah disusun sesuai dengan format yang ditetapkan di atas:

```
3  5
6  5  7  2  3
8  1  4  8  12
2  5  4  9  10
```

Baris pertama, 3 dan 5 menyatakan ukuran matriks (m dan n), sedangkan baris-baris berikutnya berisi elemen-elemen matriks. Prosedur membaca data matriks dari sebuah arsip masukan dan menyimpannya ke dalam matriks yang bernama A, masing-masing dalam bahasa *Pascal* dan bahasa *C*, adalah seperti di bawah ini.

Bahasa *Pascal*:

```
procedure BacaMatriksDariArsip(NamaArsip : string;
                               var A : MatriksInt,; var m, n :
integer)
{ Membaca data matriks dari arsip NamaArsip, dan menyimpannya ke
  dalam matriks A yang berukuran m x n.
  K. Awal : NamarAsip telah berisi nama arsip yang menyimpan data
            matriks.
  K. Akhir: A berisi data matriks; m dan n berisi ukuran matriks
}

(* DEKLARASI *)
var
  i, j : integer;   { pencacah banyak data }
  Fin  : text;      { pointer ke arsip masukan }
```

```
(* ALGORITMA: *)
begin
  { buka arsip masukan }
  assign(Fin, NamaArsip);
  reset(Fin);

  read(Fin, m, n);   {baca ukuran matriks }
  for i := 1 to m do
    for j := 1 to n do
      read(Fin, A[i,j]);
    {endfor}
  {endfor}
  close(fin);
end;
```

Algoritma 14.15 Membaca matriks dari arsip teks (Bahasa Pascal)

Bahasa C:

```
void BacaMatriksDariArsip(char NamaArsip[], MatriksInt A, int *m, int
*n)
/* Membaca data matriks dari arsip NamaArsip, dan menyimpannya ke
  dalam matriks A yang berukuran m x n.
  K. Awal : NamarAsip telah berisi nama arsip yang menyimpan data
            matriks.
  K. Akhir: A berisi data matriks; m dan n berisi ukuran matriks
*/
{
  /* DEKLARASI */
   int i, j;      /* pencacah banyak data */
   FILE *Fin;    /* pointer ke arsip masukan */

/* ALGORITMA: */

  /* buka arsip masukan */
  Fin = fopen(NamaArsip, "r");

  fscanf(Fin, "%d %d", &(*m), &(*n));  /* baca ukuran matriks */
  for (i = 1; i <= (*m); i++)
    for (j = 1; j <= (*n); j++)
      fscanf(Fin, "%d", A[i][j]);
    /*endfor*/
  /*endfor*/

  fclose(fin);
}
```

Algoritma 14.16 Membaca matriks dari arsip teks (Bahasa C)

Contoh 14.5. Menjumlahkan baris dan kolom matriks.
Asumsikan ukuran matriks tidak melebihi *NbarisMaks* –1 dan *NkolomMaks* –1. Jumlahkan elemen-elemen pada setiap baris dan setiap kolom. Hasil penjumlahan disimpan pada kolom tambahan dan baris tambahan. Jadi, hasil

penjumlahan suatu baris disimpan pada kolom paling kanan (*Nkol* + 1), dan hasil penjumlahan suatu kolom disimpan pada baris paling bawah (*Nbar* + 1).

Contoh di bawah ini memperlihatkan hasil penjumlahan setiap baris dan setiap kolom pada matriks di bawah ini disimpan pada elemen yang diarsir.

	1	2	3	4	5
1	12	45	16	9	82
2	17	15	15	1	48
3	10	15	1	9	35
4	12	7	0	3	22
5	21	16	13	19	69
6	72	98	45	41	

Algoritma penjumlahan seluruh elemen pada setiap baris dan kolom matriks sebagai berikut:

```
procedure PenjumlahanBarisdanKolom(input/output A : MatriksInt,
                                   input Nbar, Nkol : integer)
{       Menjumlahkan baris-baris matriks A[1..Nbar,1..Nkol] dan
menyimpannya pada kolom Nkol + 1. Juga menjumlahkan kolom-kolom
matriks A[1..Nbar, 1..Nkol] dan menyimpannya pada baris Nbar + 1. }
{ K.Awal : Matriks A sudah terdefinisi nilai setiap elemennya; Nbar <=
NbarisMaks - 1 dan Nkol <= NkolomMaks - 1. }
{ K.Akhir: baris Nbar + 1 dan kolom Nkol + 1 masing-masing berisi
hasil penjumlahan  elemen kolom dan penjumlahan elemen baris matriks
A. }

DEKLARASI
  i: integer  {indeks baris}
  j : integer  {indeks kolom}

ALGORITMA:
  { Penjumlahan baris }
  for i ← 1 to Nbar do
     A[i, Nkol + 1] ← 0
     for j ← 1 to Nkol do
        A[i, Nkol+1] ← A[i, Nkol+1] + A[i,j]
     endfor
  endfor

  { Penjumlahan kolom }
  for j ← 1 to Nkol do
     A[Nbar + 1,j] ← 0
     for i ← 1 to Nbar do
        A[Nbar+1, j] ← A[Nbar+1, j] + A[i, j]
     endfor
  endfor
```

Algoritma 14.17 Menjumlahkan baris dan kolom

Contoh 14.6. Kesamaan dua buah matriks

Dua buah matriks, A dan B, dikatakan sama jika ukurannya sama dan elemen pada posisi yang bersesuaian sama harganya ($A[i, j] = B[i, j]$ untuk setiap i dan j).

Sebagai contoh, dari keempat matriks di bawah ini (A, B, C, D),

(i) $A \neq B$, karena ukuran keduanya tidak sama.

(ii) $A = C$, karena ukuran keduanya sama dan elemen pada posisi yang bersesuaian sama.

(iii) $A \neq D$, karena elemen pada posisi yang bersesuaian tidak sama.

	1	2	3	4
1	1	2	3	4
2	5	6	7	8
3	9	10	11	12
4	12	13	14	15

A

	1	2	3
1	1	2	3
2	4	5	6
3	7	8	9
4	10	11	12

B

	1	2	3	4
1	1	2	3	4
2	5	6	7	8
3	9	10	11	12
4	12	13	14	15

C

	1	2	3	4
1	11	2	3	4
2	5	16	7	8
3	9	10	11	12
4	18	13	14	15

D

Algoritma untuk memeriksa kesamaan dua buah matriks, sebagai berikut (sebagai fungsi):

```
function MatriksSama(input A : MatriksInt, input NbarA, NkolA :
integer,
         input B : MatriksInt, input NbarB, NkolB : integer) →
boolean
{ Memeriksa kesamaan matriks A[1..NbarA, 1..NkolA] dan B[1..NbarB,
1..NkolB]. Fungsi mengembalikan nilai true jika A = B atau false jika
A ≠ B }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }
  sama : boolean   { true jika A = B, false jika tidak sama }

ALGORITMA:
  { periksa ukuran matriks A dan B }
  if (NbarA ≠ NbarB) and (NkolA ≠ NkolB) then
     sama ← false    { A ≠ B }

  else { (NbarA = NbarB) and (NkolA = NkolB) }
     { periksa kesamaan elemen pada posisi yang bersesuaian.
       Pemeriksaan kesamaan matriks dihentikan bila ditemukan sebuah
       A[i,j] ≠ B[i, j] untuk suatu i dan j. }

     sama ← true    { asumsikan di awal bahwa A = B }
     i ← 1
     while ( i ≤ NbarA) and (sama) do
       j ← 1
       while (j ≤ NkolA) and (sama) do
          if A[i,j] ≠ B[i,j] then
             sama ← false  { A dan B tidak sama. Stop! }
          else
             j ← j + 1
```

```
        endif
      endwhile
      { j > NkolA or not sama }
      if sama then { jika di baris i masih sama, periksa di baris i + 1}
        i ← i + 1
      endif
    endwhile
    { i > NbarA or not sama }

    return sama

  endif
```

Algoritma 14.18 Memeriksa kesamaan dua buah matriks (versi 1)

Algoritma 14.18 di atas dapat dibuat lebih elegan tanpa menggunakan peubah *boolean*. Ketika ditemukan sebeuah elemen matriks yang tidak sama, maka fungsi langsung keluar dan mengembalikan nilai *false*. Hasil perbaikan terhadap Algoritma 14.18 dinyatakan di dalam Algoritma 14.19 berikut ini:

```
function MatriksSama(input A : MatriksInt, input NbarA, NkolA : integer,
         input B : MatriksInt, input NbarB, NkolB : integer) → boolean
{ Memeriksa kesamaan matriks A[1..NbarA, 1..NkolA] dan B[1..NbarB,
1..NkolB]. Fungsi mengembalikan nilai true jika A = B atau false jika
A ≠ B }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }

ALGORITMA:
  { periksa ukuran matriks A dan B }
  if (NbarA ≠ NbarB) and (NkolA ≠ NkolB) then
    return false   { A ≠ B }

  else { (NbarA = NbarB) and (NkolA = NkolB) }
    { periksa kesamaan elemen pada posisi yang bersesuaian.
     Pemeriksaan kesamaan matriks dihentikan bila ditemukan sebuah
     A[i,j] ≠ B[i, j] untuk suatu i dan j. }
    for i ← 1 to NbarA do
      for j ← 1 to NkolA do
        if A[i,j] ≠ B[i,j] then
          return false  { A dan B tidak sama. Stop! }
        endif
      endfor
    endfor

    return true  { semua A[i,j] = B[i,j] }

  endif
```

Algoritma 14.19 Memeriksa kesamaan dua buah matriks (versi 2)

Bila algoritma menguji kesamaan dua buah matriks dinyatakan sebagai prosedur, maka Algoritma 14.18 yang lebih tepat dimodifikasi menjadi prosedur, karena peubah sama dinyatakan sebagai parameter keluaran di dalam *header* prosedur. Prosedur uji kesamaan dua buah matriks dinyatakan di dalam Algoritma 14.20 berikut ini:

```
procedure UjiKesamaanMatriks(input A : MatriksInt,
                             input NbarA, NkolA : integer,
                             input B : MatriksInt,
                             input NbarB, NkolB : integer,
                             output sama : boolean)
{ Memeriksa kesamaan matriks A[1..NbarA, 1..NkolA] dan B[1..NbarB,
1..NkolB]. }
{ K. Awal: Matriks A dan B sudah terdefinisi elemen-elemennya;
           NbarA, NkolA, NbarB, NkolB sudah berisi ukuran matriks. }
{ K. Akhir: sama bernila true jika A = B, atau false jika A ≠ B }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }

ALGORITMA:
  { periksa ukuran matriks A dan B }
  if (NbarA ≠ NbarB) and (NkolA ≠ NkolB) then
     sama ← false   { A ≠ B }

  else { (NbarA = NbarB) and (NkolA = NkolB) }
     { periksa kesamaan elemen pada posisi yang bersesuaian.
       Pemeriksaan kesamaan matriks dihentikan bila ditemukan sebuah
       A[i,j] ≠ B[i, j] untuk suatu i dan j. }

     sama ← true   { asumsikan di awal bahwa A = B }
     i ← 1
     while ( i ≤ NbarA) and (sama) do
       j ← 1
       while (j ≤ NkolA) and (sama) do
          if A[i,j] ≠ B[i,j] then
             sama ← false  { A dan B tidak sama, Stop! }
          else
             j ← j + 1
          endif
       endif
       { j > NkolA or not sama }
       if sama then  { jika di baris i masih sama, periksa di baris i + 1}
          i ← i + 1
       endif
     endwhile
     { i > NbarA or not sama }

  endif
```

Algoritma 14.20 Memeriksa kesamaan dua buah matriks (versi 3)

Contoh 14.7. Matriks simetri.
Matriks A dikatakan simetri jika A adalah matriks bujursangkar (jumlah baris = jumlah kolom) dan $A[i, j] = A[j, i]$ untuk setiap i dan j (dengan kata lain, elemen-elemen di bawah diagonal utama merupakan pencerminan elemen-elemen di atasnya, atau sebaliknya. Elemen diagonal utama pada matriks di bawah ini dinyatakan dengan arsiran).

Sebagai contoh, dari ketiga buah matriks A, B, dan C di bawah ini,
(i) A bukan matriks simetri, karena A bukan matriks bujursangkar.
(ii) B matriks simetri, karena jumlah baris dan jumlah kolom sama dan $A[i,j] = A[j, i]$ untuk setiap i dan j.
(iii) C bukan matriks simetri, karena ada elemen $C[i, j] \neq C[j, i]$, misalnya $C[2, 3] \neq C[3, 2]$.

	1	2	3
1	1	2	3
2	5	6	7
3	9	10	11
4	12	13	14

A

	1	2	3	4
1	1	2	3	4
2	2	6	7	8
3	3	7	11	12
4	4	8	12	15

B

	1	2	3	4
1	11	2	3	4
2	5	16	7	8
3	3	10	11	12
4	4	8	12	15

C

Algoritma untuk memeriksa kesimetrian sebuah matriks sebagai berikut:

```
function Simetri(input A : MatrikaInt, input Nbar, Nkol : integer)
                 → boolean
{Memeriksa apakah matriks A[1..Nbar, 1..Nkol] simetri.}
{Fungsi mengembalikan true jika A simetri, atau false jika A tidak simetri}

DEKLARASI
  i    : integer  { indeks baris }
  j    : integer  { indeks kolom }

ALGORITMA:
  { periksa jumlah baris dan jumlah kolom harus sama }
  if  Nbar ≠ Nkol then
     return false    { A tidak simetri }

  else { Nbar = Nkol }
       { Periksa kesamaan elemen yang bersesuaian di atas dan di bawah
      diagonal utama. Pemeriksaan dapat dihentikan bila ditemukan
suatu   A[i,j] ≠ A[j,i]. }

     for i ← 1 to NbarA do
       for j ← 1 to NkolA do
          if A[i,j] ≠ A[j,i] then
             return false
          endif
       endfor
     endfor

     return true

  endif
```

Algoritma 14.21 Memeriksa kesimetrian matriks

Contoh 14.8. Mencari elemen terbesar di dalam matriks
Elemen maksimum matriks adalah elemen matriks yang mempunyai nilai terbesar. Elemen maksimum matriks dicari dengan menelusuri semua elemen matriks, mulai dari elemen *A*[1, 1] sampai elemen *A*[*Nbar*, *Nkol*].

Algoritma mencari elemen terbesar sebagai berikut:

```
procedure Maksimum(input A : MatriksInt, input Nbar, Nkol : integer,
                   output Maks : integer)
{ Mencari elemen maksimum pada matriks A[1..Nbar, 1..Nkol]. }
{ K.Awal : Matriks A sudah terdefinisi harga elemen-elemennya. }
{ K.Akhir: Maks berisi elemen maksimum matriks A }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }

ALGORITMA:
  Maks ← -9999  { diasumsikan -9999 sebagai nilai maksimum sementara }
  for i ← 1 to Nbar do
    for j ← 1 to Nkol do
      if A[i,j] > Maks then
        Maks ← A[i,j]
      endif
    endfor
  endfor
```

Algoritma 14.22 Mencari elemen maksimum di dalam matriks

Contoh 14.9. Mencari elemen tertentu di dalam matriks
Mencari nilai tertentu (misalkan *x*) di dalam matriks. Bila *x* ditemukan, maka keluaran yang diinginkan adalah indeks baris dan indeks kolomnya. Bila *x* tidak ditemukan di dalam matriks, maka indeks baris dan indeks kolom yang dihasilkan masing-masing bernilai -1.

Algoritma pencarian beruntun pada larik kita terapkan untuk matriks:

```
procedure CariX(input A : MatriksInt, input Nbar, Nkol : integer,
                input x : integer,
                output IdxBaris, IdxKolom : integer)
{ Mencari x di dalam matriks A[1..Nbar, 1..Nkol]. }

{ K.Awal: x sudah terdefinisi nilainya; matriks A juga sudah
terdefinisi elemen-elemennya. }
{ K.Akhir: IdxBaris dan IdxKolom berisi indeks matriks sedemikian
sehingga A[IdxBaris, IdxKolom] = x. Jika x tidak ditemukan, maka
IdxBaris dan IdxKolom diisi nilai -1 }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }
  ketemu : boolean { true jika X ditemukan, false jika sebaliknya }
```

```
ALGORITMA:
  i ← 1
  ketemu ← false
  while (i ≤ Nbar) and (not ketemu) do
    j ← 1
    { telusuri elemen-elemen matriks pada baris ke-i, dimulai dari
     kolom ke-j }
    while (j ≤ Nkol) and (not ketemu) do
      if A[i,j] = X then
         ketemu ← true
      else
         j ← j + 1  { periksa pada kolom berikutnya }
      endif
    endwhile
    { j > Nkol or ketemu }

    if not ketemu then
       i ← i + 1   { akan diperiksa elemen-elemen baris berikutnya }
    endif
  endwhile
  { i > Nbar or ketemu }

  if ketemu then
     IdxBaris ← i
     IdxKolom ← j
  else
     IdxBaris ← -1
     IdxKolom ← -1
  endif
```

Algoritma 14.23 Mencari elemen tertentu di dalam matriks

Contoh 14.10. Membuat matriks *transpose*

Matriks *transpose* adalah matriks yang diperoleh dari matriks yang lain. Matriks *transpose* dihasilkan dengan mentransformasikan seluruh elemen pada baris i (dari matriks asal A) menjadi elemen-elemen kolom j (pada matriks *transpose* At), sedemikian sehingga

$At[j, i] = A[i, j]$ untuk setiap i dan j.

Contoh matriks A dan matriks *transpose*-nya:

	1	2	3
1	1	2	3
2	5	6	7
3	9	10	11
4	12	13	14

A

	1	2	3	4
1	1	5	9	12
2	2	6	10	13
3	3	7	11	14

At

Yang harus diperhatikan dalam membuat matriks *transpose* adalah ukuran matriks *transpose*, sebab ukuran matriks *transpose* akan berkebalikan dengan ukuran matriks asal (pada contoh di atas, A berukuran 4×3, sedangkan At 3×4). Kita mengasumsikan ukuran matriks *transpose* masih tetap lebih kecil dari ukuran maksimum matriks yang telah dideklarasikan.

Algoritma membuat matriks *transpose*:

```
procedure BuatTranspose(input A : MatriksInt, input Nbar, Nkol:
integer,
                        output At : MatriksInt, output NbarAt, NkolAt:
                        integer)
{ Membentuk transpose dari matriks A[1..Nbar, 1..Nkol]. }
{ K.Awal : Matriks A sudah terdefinisi nilai elemen-elemennya. }
{ K.Akhir: At adalah tranpose dari matriks A sedemikian sehingga
  At[j,i] = A[i,j] untuk semua i dan j; NbarAt dan NkolAt berisi
ukuran matriks hasil transpose (jumlah baris dan jumlah kolom) }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }

ALGORITMA:
  NbarAt ← Nkol    { jumlah baris matriks transpose }
  NkolAt ← Nbar    { jumlah kolom matriks transpose }
  for i ← 1 to Nbar do
    for j ← 1 to Nkol do
      At[j, i] ← A[i, j]
    endfor
  endfor
```

Algoritma 14.24 Membuat matriks *transpose*

Contoh 14.11. Mencetak elemen diagonal matriks
Misalkan kita ingin mencetak hanya elemen-elemen diagonal matriks. Elemen diagonal adalah elemen $A[i, i]$ untuk semua I. Sebagai contoh, bila matriks A sebagai berikut:

	1	2	3	4
1	1	2	3	4
2	2	6	7	8
3	3	7	11	12
4	4	8	12	15

maka yang dicetak adalah: 1 6 11 15.

Algoritma mencetak elemen-elemen diagonal matriks:

```
procedure CetakDiagonal(input A : Matriks, input Nbar, Nkol : integer)
{ Mencetak elemen-elemen diagonal matriks A[1..Nbar, 1..Nkol ke
piranti keluaran. }
{ K.Awal : Matriks A sudah terdefinisi nilai elemen-elemennya. }
```

```
{ K.Akhir: Elemen A[i,i] tercetak ke piranti keluaran. }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }

ALGORITMA:
  for i ← 1 to Nbar do
    for j ← 1 to Nkol do
      if i = j then
        write(A[i,i])
      endif
    endfor
  endfor
```

Algoritma 14.25 Mencetak elemen-elemen diagonal matriks

Algoritma `CetakDiagonal` di atas dapat dibuat lebih mangkus dengan hanya satu buah kalang *for* sebagai berikut:

```
procedure CetakDiagonal(input A : MatriksInt, input Nbar,
Nkol:integer)
{ Mencetak elemen-elemen diagonal matriks A[1..Nbar, 1..Nkol ke
piranti keluaran. }
{ K.Awal : Matriks A sudah terdefinisi nilai elemen-elemennya. }
  K.Akhir: Elemen A[i,i] tercetak ke piranti keluaran. }

DEKLARASI
  i : integer      { indeks baris }
  j : integer      { indeks kolom }

ALGORITMA:
  for i ← 1 to Nbar do
    write(A[i,i])
  endfor
```

Algoritma 14.26 Mencetak elemen-elemen diagonal matriks (versi 2)

Contoh 14.12. Matriks nol

Matriks nol adalah matriks dengan semua elemen adalah 0. Matriks ini banyak berguna dalam matematika, khususnyan untuk menyimpulkan apakah hasil operasi matriks mengahasilkan matriks nol.

Contoh sebuah matriks nol yang berukuran 4×4:

	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

Algoritma untuk memeriksa apakah sebuah matriks merupakan matriks nol adalah dengan menelusuri seluruh elemen matriks. Pemeriksaan dapat dihentikan bila ditemukan pertama kalinya sebuah $A[i, j] \neq 0$ untuk suatu i dan j.

```
function Nol(input A : MatriksInt, input Nbar, Nkol : integer) →
boolean
{ Memeriksa apakah matriks A[1..Nbar, 1..Nkol] merupakan matriks nol;
mengembalikan nilai true jika A adalah matriks nol atau false jika
bukan matriks nol }

DEKLARASI
  i : integer  { indeks baris }
  j : integer  { indeks kolom }

ALGORITMA:
  for i ← 1 to NbarA do
    for j ← 1 to NkolA do
      if A[i,j] ≠ 0 then
        return false  { bukan matriks nol, stop! }
      endif
    endfor

  endfor

  return true   { semua A[i,j] = 0 }
```

Algoritma 14.27 Memeriksa apakah matriks nol

Contoh 14.13. Matriks segitiga bawah.
Matriks segitiga bawah adalah matriks bujursangkar dengan semua elemen di atas diagonal utama adalah 0. Matriks segitiga bawah banyak digunakan dalam pembahasan solusi sistem persamaan lanjar (linear).

Sebuah matriks bujursangkar di bawah ini merupakan matriks segitiga bawah:

	1	2	3	4
1	21	0	0	0
2	8	9	0	0
3	45	0	8	0
4	12	5	67	38

Algoritma untuk memeriksa apakah sebuah matriks merupakan matriks segitiga bawah adalah dengan menelusuri seluruh elemen matriks. Pemeriksaan dapat dihentikan bila ditemukan pertama kali $A[i, j] = 0$ untuk $i = 1..Nbar$ dan $j = i+1..Nkol$.

```
function SegitigaBawah(input A : MatriksInt, input Nbar, Nkol: integer)
                       → boolean
{ Memeriksa apakah A[1..Nbar,1..Nkol] merupakan matriks segitiga bawah. }
{ Fungsi mengembalikan nilai true jika A adalah matriks segitiga bawah
atau false jika bukan. }

DEKLARASI
  i : integer  { indeks baris }
  j : integer  { indeks kolom }

ALGORITMA:
  if Nbar ≠ Nkol then  { A bukan matriks bujursangkar }
    return false
  else
    for i ← 1 to NbarA do
        {      periksa elemen-elemen di atas diagonal utama }
      for j ← i + 1 to Nkol do
        if A[i,j] ≠ 0 then
          return false  { A bukan matriks segitiga bawah, stop! }
        endif
      endfor
    endfor
    return true
  endif
```

Algoritma 14.28 Memeriksa apakah matriks segitiga bawah

Contoh 14.14. Perkalian dua buah matriks

Perkalian dua buah matriks A dan B menghasilkan matriks C, yaitu $C = A \times B$. Algoritma mengalikan dua buah matriks lebih rumit daripada menjumlahkan keduanya. Syarat perkalian matriks A dan matrik B: jumlah kolom matriks A harus sama dengan jumlah baris matriks B.

Misalkan $A[1..m, 1..n]$ dan $B[1..n, 1..p]$, yang dalam hal ini,
m adalah jumlah baris matriks A,
n adalah jumlah kolom matriks A dan juga jumlah baris matriks B.
p adalah jumlah kolom matriks B.
Hasil perkalian A dan B menghasilkan matriks $C[1..m, 1..p]$.

Algoritma perkalian matriks:

1. Inisialisasi $C[i, j]$ dengan 0, untuk $i = 1, 2, ..., m$ dan $j = 1, 2, ..., p$.
2. Untuk setiap baris $i = 1, 2, ..., m$ pada matriks A lakukan:
   Untuk setiap kolom $j = 1, 2, ..., p$ pada matriks B lakukan:
   Untuk setiap baris $k = 1, 2, ..., n$ pada matriks B lakukan:
   $C[i, j] = C[i, j] + A[i, k] * B[k, j]$

Contoh perkalian matriks:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 10 & 11 \\ 12 & 13 \\ 14 & 15 \end{bmatrix} = \begin{matrix} (1*10)+(2*12)+(3*14) & (1*11)+(2*13)+(2*15) \\ (4*10)+(5*12)+(6*14) & (4*11)+(5*13)+(6*15) \\ (7*10)+(8*12)+(9*14) & (7*11)+(8*13)+(9*15) \end{matrix}$$

$$= \begin{bmatrix} 76 & 67 \\ 184 & 199 \\ 292 & 326 \end{bmatrix}$$

Pseudo-code algoritma perkalian matriks:

```
procedure PerkalianMatriks(input A : MatriksInt, input m, n : integer,
                           input B : MatriksInt, input p : integer,
                           output C : MatriksInt, output NbarC, NkolC: integer)
{Mengalikan matriks A[1..m, 1..n] dan B[1..n, 1..p] menghasilkan
matriks C[1..m, 1..p]. }
{ K.Awal : Matriks A dan B sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Matriks C berisi hasil perkalian A dan B, NbarC dan NkolC
  berisi ukuran matriks C }

DEKLARASI
  i, j, k : integer

ALGORITMA:
   NbarC ← m    { jumlah baris matriks hasil perkalian }
   NkolC ← p    { jumlah kolom matriks hasil perkalian }

   { inisialisasi seluruh elemen matriks C dengan 0 }
   for i ← 1 to NbarC do
     for j ← 1 to NkolC do
       C[i,j] ← 0
     endfor
   endfor

   { lakukan operasi perkalian }
   for i ← 1 to m do
     for j ← 1 to p do
        for k ← 1 to n do
           C[i,j] ← C[i,j] + A[i,k] * B[k,j]
        endfor
     endfor
   endfor
```

Algoritma 14.29 Perkalian dua buah matriks (versi 1)

Inisialisasi matriks *C* dengan 0 dapat dilakukan sekali jalan dengan operasi perkalian. Perbaikan algoritma perkalian di atas sebagai berikut:

```
procedure PerkalianMatriks(input A : MatriksInt, input m, n : integer,
                           input B : MatriksInt, input p : integer,
                           output C : MatriksInt, output NbarC, NkolC:
                           integer)
```

```
{ Mengalikan matriks A[1..m, 1..n] dan B[1..n, 1..p] menghasilkan
matriks C[1..m, 1..p]. }
{ K.Awal : Matriks A dan B sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Matriks C berisi hasil perkalian A dan B, NbarC dan NkolC
  berisi ukuran matriks C }

DEKLARASI
  i : integer
  j : integer
  k : integer

ALGORITMA:
  NbarC ← m    { jumlah baris matriks hasil perkalian }
  NkolC ← p    { jumlah kolom matriks hasil perkalian }

  for i ← 1 to m do
    for j ← 1 to p do
      C[i,j] ← 0      { inisialisasi C[i,j] dengan 0 }
      for k ← 1 to n do
        C[i,j] ← C[i,j] + A[i,k] * B[k,j]
      endfor
    endfor
  endfor
```

Algoritma 14.30 Perkalian dua buah matriks (versi 2)

Jika A dan B keduanya berukuran bujursangkar ($n \times n$), maka algoritma perkalian matriks menjadi lebih sederhana. Hasil perkalian adalah matriks C yang juga berukuran $n \times n$. Ukuran matriks hasil perkalian tidak perlu disimpan. *Pseudo-code* algoritmanya sebagai berikut:

```
procedure PerkalianMatriks(input A, B : MatriksInt, input n : integer,
                           output C : MatriksInt)
{       Mengalikan matriks A[1..n, 1..n] dan B[1..n, 1..n] menghasilkan
matriks C[1..n, 1..n]. }
{ K.Awal : Matriks A dan B sudah terdefinisi elemen-elemennya. }
{ K.Akhir: Matriks C berisi hasil perkalian A dan B. }

DEKLARASI
  i : integer
  j : integer
  k : integer

ALGORITMA:
  for i ← 1 to n do
    for j ← 1 to n do
      C[i,j] ← 0      { inisialisasi C[i,j] dengan 0 }
      for k ← 1 to n do
        C[i,j] ← C[i,j] + A[i,k] * B[k,j]
      endfor
    endfor
  endfor
```

Algoritma 14.31 Perkalian dua buah matriks (versi 2)

Contoh 14.15. Matriks karakter

Sebuah matriks karakter W berukuran $m \times n$. Elemen-elemen pada setiap baris berisi karakter-karakter pembentuk kata (lihat contoh di bawah). Satu baris menyatakan sebuah kata. Akhir sebuah kata adalah karakter spasi (' '). Diasumsikan elemen pertama pada setiap baris bukan spasi. Elemen matriks yang kosong berisi spasi.

	1	2	3	4	5	6	...	n
1	'a'	'm'	'p'	'u'	'h'			
2	'b'	'a'	'n'	'd'	'u'	'n'	'g'	
3	's'	'e'	'n'					
4	't'	'u'	'm'	'p'	'a'	'h'		
5	'y'	'a'						
6	'i'	't'	'b'					
...								
m	'm'	'a'	'r'	'i'				

Tuliskan algoritma untuk:
(a) mencetak setiap kata
(b) lebar kata terpanjang

Penyelesaian:

(a) Mencetak setiap kata

Algoritmanya sebagai berikut: telusuri elemen-elemen matriks per baris per kolom. Selama elemen matriks bukan spasi, cetak karakter di dalam elemen matriks tersebut.

```
procedure CetakKata(input W : MatriksKar, input m, n : integer)
{ Mencetak "kata" di dalam matriks W }
{ K.Awal : Matriks W sudah terdefinisi elemen-elemennya. }
{ K.Akhir: "Kata" tercetak ke piranti keluaran. }

DEKLARASI
  i : integer      { pencatat indeks baris }
  j : integer      { pencatat indeks kolom }

ALGORITMA:
  for i ← 1 to m do
    { cetak elemen matriks pada baris i sampai ketemu spasi }
    for j ← 1 to n do
        if W[i,j] ≠ ' ' then
           write(W[i,j])
        endif
    endfor

  endfor
```

Algoritma 14.32 Mencetak 'kata' pada setiap baris matriks

(b) Menentukan lebar kata terpanjang

Algoritmanya sebagai berikut: asumsikan untuk sementara lebar kata terpanjang adalah 0. Telusuri elemen-elemen matriks per baris per kolom. Catat lebar setiap kata, lalu bandingkan dengan lebar kata terpanjang sementara. Jika lebar kata sekarang lebih besar daripada lebar kata terpanjang sementara, ubahlah lebar kata terpanjang sementara dengan lebar kata sekarang.

```
procedure HitungLebarKataTerpanjang(input W : MatriksKar,
                                    input m, n : integer, output p : integer)
{ Menentukan lebar kata terpanjang }
{ K.Awal : Matriks W sudah terdefinisi elemen-elemennya. }
{ K.Akhir: p berisi lebar kata terpanjang. }


DEKLARASI
  i : integer      { pencatat indeks baris }
  j : integer      { pencatat indeks kolom }
  panjang : integer  { lebar kata }

ALGORITMA:
  p ← 0   { lebar kata terpanjang sementara }
  for i ← 1 to m do
    panjang ← 0

    { hitung panjang kata pada baris i }
    for i ← 1 to n do
       if W[i,j] ≠ ' ' then
          panjang ← panjang + 1
       endif
    endfor

    { update nilai p }
    if panjang > p then
       p ← panjang
    endif

  endfor
```

Algoritma 14.33 Menentukan lebar kata terpanjang

Soal Latihan Bab 14

1. Apa perbedaan pada deklarasi 2 buah matriks temperatur-tekanan di bawah ini:

```
DEKLARASI
   { Matriks temperatur-tekanan }

   const NBarisMaks = ...  { jumlah baris matriks }
   const NKolomMaks = ...  { jumlah kolom matriks }
```

```
   type TP : record <T:real, P:real>   { tipe elemen matriks }

   B : array [1..NbarisMaks, 1..NKolomMaks] of TP
DEKLARASI
   { Matriks temperatur-tekanan }

   const NBarisMaks = ...   { jumlah baris matriks }
   const NKolomMaks = ...   { jumlah kolom matriks }

   type TP : record <T : array[1..NbarisMaks, 1..NKolomMaks] of real,
                     P : array[1..NbarisMaks, 1..NKolomMaks] of real>
   B : TP
```

2. Diberikan sebuah matriks bujursangkar yang berukuran $n \times n$. Tuliskan algoritma untuk menentukan apakah matriks tersebut merupakan matriks segitiga atas.

3. Diberikan sebuah matriks bujursangkar yang berukuran $n \times n$. Tuliskan algoritma untuk menentukan apakah matriks tersebut merupakan matriks identitas (I).

4. Misalkan matriks C yang berukuran $m \times n$ sudah berisi data karakter. Tuliskan algoritma untuk menghitung frekuensi kemunculan huruf 'A' di dalam matriks tersebut.

5. Diberikan sebuah matriks *integer* yang berukuran $m \times n$. Tuliskan algoritma untuk menentukan apakah di dalam matriks tersebut ada baris yang semua elemennya 0.

6. M orang mahasiswa mengambil n mata kuliah. Setelah seluruh ujian akhir selesai, nilai rata-rata (NR) mahasiswa tersebut segera dihitung. NR dihitung dengan rumus:

$$NR = \frac{\sum_{j=1}^{n} MK_j \times SKS_j}{\sum_{j=1}^{n} SKS_j}$$

yang dalam hal ini, MK_j adalah mata kuliah yang ke-j dan SKS_j adalah bobot SKS dari mata kuliah MK_j. Nilai mahasiswa (berupa indeks nilai A, B, C, D, E) disimpan di dalam matriks *NilaiMhs* yang berukuran $m \times n$. Baris i pada matriks menyatakan mahasiswa ke-i, dan kolom j menyatakan mata kuliah (MK) ke-j. Setiap elemen matriks adalah berupa *record* yang terdiri atas *field SKS* dan *field IndeksNilai* (jadi matriksnya bertipe terstruktur). *NilaiMhs*[i, j] menyatakan data nilai mahasiswa i yang mengambil mata kuliah j, *NilaiMhs*[i, j]. *IndeksNilai* menyatakan indeks nilai mata kuliah j yang diambil mahasiswa i.

Tulislah algoritma untuk menghitung dan menuliskan *NR* setiap mahasiswa.

7. Tinjau matriks *B* pada Gambar 14.3 dan pendeklarasian tipe matriksnya. Tuliskan algoritma untuk menentukan titik-titik pada permukaan logam yang temperaturnya di atas T_0 dan tekanannya di atas P_0. T_0 dan P_0 adalah parameter masukan.

8. Tinjau kembali Contoh 14.15. Tuliskan algoritma untuk menentukan kata terpanjang.

15
Algoritma Pencarian

Pencarian (*searching*) merupakan proses yang fundamental dalam pengolahan data. Proses pencarian adalah menemukan nilai (data) tertentu di dalam sekumpulan data yang bertipe sama (baik bertipe dasar atau bertipe bentukan). Sebagai contoh, untuk mengubah (*update*) data tertentu, langkah pertama yang harus dilakukan adalah mencari keberadaan data tersebut di dalam kumpulannya. Jika data yang dicari ditemukan, maka data tersebut dapat diubah nilainya dengan data yang baru. Aktivitas awal yang sama juga dilakukan pada proses penambahan (*insert*) data baru. Proses penambahan data dimulai dengan mencari apakah data yang akan ditambahkan sudah terdapat di dalam kumpulan. Jika sudah ada dan mengasumsikan tidak boleh ada duplikasi data maka data tersebut tidak perlu ditambakan, tetapi jika belum ada, maka tambahkan.

Sumber: http://www.epa.gov

Bab 15 ini membicarakan algoritma pencarian data di dalam larik. Algoritma pencarian yang akan dibicarakan dimulai dengan algoritma pencarian yang paling sederhana (yaitu pencarian beruntun atau *sequential search*) hingga algoritma pencarian yang lebih maju yaitu pencaian bagidua (*binary search*).

15.1 Spesifikasi Masalah Pencarian

Pertama-tama kita menspesifikasikan masalah pencarian di dalam larik. Di dalam Bab 15 ini kita mendefinisikan persoalan pencarian secara umum sebagai berikut: *Diberikan larik L yang sudah terdefinisi elemen-elemennya, dan x adalah elemen yang bertipe sama dengan elemen larik L. Carilah x di dalam larik L.*

Hasil atau keluaran dari persoalan pencarian dapat bermacam-macam, bergantung pada spesifikasi rinci dari persoalan tersebut, misalnya:

a. Pencarian hanya untuk memeriksa keberadaan x. Keluaran yang diinginkan misalnya pesan (*message*) bahwa x ditemukan atau tidak ditemukan di dalam larik.

Contoh:

```
write('ditemukan!') atau
write('tidak ditemukan!')
```

b. Hasil pencarian adalah indeks elemen larik. Jika *x* ditemukan, maka indeks elemen larik tempat *x* berada diisikan ke dalam *idx*. Jika *x* tidak terdapat di dalam larik *L*, maka *idx* diisi dengan harga khusus, misalnya -1.
Contoh: Perhatikan larik di bawah ini:

L	21	36	8	7	10	36	68	32	12	10	36
	1	2	3	4	5	6	7	8	9	10	11

Misalkan $x = 68$, maka $idx = 7$, dan bila $x = 100$, maka $idx = -1$.

c. Hasil pencarian adalah sebuah nilai *boolean* yang menyatakan status hasil pencarian. Jika *s* ditemukan, maka sebuah peubah bertipe *boolean*, misalnya "ketemu", diisi dengan nilai true, sebaliknya "ketemu" diisi dengan nilai false. Hasil pencarian ini selanjutnya disimpulkan pada bagian pemanggilan prosedur.

Contoh: Perhatikan larik *L* di atas:
Misalkan $x = 68$, maka ketemu = true, dan bila $x = 100$, maka ketemu = false.

Untuk kedua macam keluaran b dan c di atas, kita mengonsultasi hasil pencarian setelah proses pencarian selesai dilakukan, bergantung pada kebutuhan. Misalnya menampilkan pesan bahwa *x* ditemukan (atau tidak ditemukan) atau memanipulasi nilai *x*.

Contoh:

```
(1)  if idx ≠ -1 then  { x ditemukan }
       L[idx] ← L[idx] + 1   { manipulasi nilai x }
     endif
```

```
(2)  if ketemu then  {yaitu, ketemu = true}
        write(x, 'tidak ditemukan')
     else
        write(x, 'ditemukan')
     endif
```

Hal lain yang harus diperjelas dalam masalah pencarian adalah mengenai duplikasi data. Apabila X yang dicari terdapat lebih dari satu banyaknya di dalam larik L, maka hanya X yang pertama kali ditemukan yang diacu algoritma pencarian selesai. Sebagai contoh, perhatikan larik di bawah ini:

L

21	36	8	7	10	36	68	32	12	10	36
1	2	3	4	5	6	7	8	9	10	11

Larik L memiliki tiga buah nilai 36. Bila $x = 36$, maka algoritma pencarian selesai ketika X ditemukan pada elemen ke-2 dan menghasilkan $idx = 2$ (atau menghasilkan ketemu = true jika mengacu pada keluaran pencarian jenis c). Elemen 36 lainnya tidak dipertimbangkan lagi dalam pencarian.

Algoritma pencarian yang akan dibahas di dalam Bab 15 ini adalah:

1. Algoritma pencarian beruntun (*sequential search*).
2. Algoritma pencarian bagidua (*binary search*).

Untuk masing-masing algoritma, tipe larik yang digunakan didefinisikan di bagian deklarasi global seperti di bawah ini. Larik L bertipe *LarikInt*.

```
{ kamus data global }

DEKLARASI
  const Nmaks = 100     {jumlah maksimum elemen larik }
  type LarikInt = array [1..Nmaks] of integer
```

Algoritma 15.1 Deklarasi larik integer yang digunakan di dalam bab ini.

15.2 Algoritma Pencarian Beruntun

Algoritma pencarian yang yang paling sederhana, yaitu metode **pencarian beruntun** (*sequential search*). Nama lain algoritma pencarian beruntun adalah **pencarian lurus** (*linear search*).

Pada dasarnya, algoritma pencarian beruntun adalah proses membandingkan setiap elemen larik satu per satu secara beruntun, mulai dari elemen pertama, sampai elemen yang dicari ditemukan, atau seluruh elemen sudah diperiksa.

Contoh 15.1. Perhatikan larik L di bawah ini dengan $n = 6$ elemen:

13	16	14	21	76	15
1	2	3	4	5	6

Misalkan nilai yang dicari adalah: x = 21
Elemen yang dibandingkan (berturut-turut): 13, 16, 14, 21 (ditemukan!)
Indeks larik yang dikembalikan: *idx* = 4

Misalkan nilai yang dicari adalah: x = 13
Elemen yang dibandingkan (berturut-turut): 13 (ditemukan!)
Indeks larik yang dikembalikan: *idx* = 1

Misalkan nilai yang dicari adalah: x = 15
Elemen yang dibandingkan (berturut-turut): 13, 16, 14, 21, 76, 21 (tidak ditemukan!)
Indeks larik yang dikembalikan: *idx* = -1

Terdapat dua versi algoritma pencarian beruntun. Pada algoritma versi pertama, aksi pembandingan dilakukan sebagai kondisi pengulangan, sedangkan algoritma versi kedua, aksi pembandingan dilakukan di dalam badan pengulangan. Versi pertama tidak menggunakan peubah *boolean* dalam proses pencarian, sedangkan versi kedua menggunakan peubah *boolean*. Untuk masing-masing versi kita tuliskan dua macam algoritmanya berdasarkan hasil yang diinginkan: indeks larik atau nilai *boolean*. Selain itu, kita asumsikan jumlah elemen di dalam larik adalah *n* buah.

(a) Versi 1 (Pembandingan elemen dilakukan sebagai kondisi pengulangan)

(1) Hasil pencarian: sebuah peubah *boolean* yang bernilai *true* bila *x* ditemukan atau bernilai *false* bila *x* tidak ditemukan.

Setiap elemen larik *L* dibandingkan dengan *x* mulai dari elemen pertama, *L*[1]. Aksi pembandingan dilakukan selama indeks larik *i* belum melebihi *n* dan *L*[*i*] tidak sama dengan *x*. Aksi pembandingan dihentikan bila *L*[*i*] = *x* atau *i* = *n*. Elemen terakhir, *L*[*n*], diperiksa secara khusus. Keluaran yang dihasilkan oleh prosedur pencarian adalah sebuah peubah *boolean* (misal nama peubahnya *ketemu*) yang bernilai *true* jika *x* ditemukan, atau bernilai *false* jika *x* tidak ditemukan.

Algoritma pencarian beruntun versi 1 untuk kategori hasil berupa nilai *boolean* dapat kita tulis sebagai sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch1(input L : LarikInt, input n : integer,
                     input x : integer, output ketemu: boolean)

{ Mencari keberadaan nilai x di dalam larik L[1..n]. }
{ K.Awal: x dan larik L[1..n] sudah terdefinisi nilainya. }
{ K.Akhir: ketemu bernilai true jika x ditemukan. Jika x tidak
ditemukan, ketemu bernilai false. }
```

```
DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   i ← 1
   while (i < n ) and (L[i] ≠ x) do
      i ← i + 1
   endwhile
   { i = n or L[i] = x }
   if L[i] = x then        { x ditemukan }
     ketemu ← true
   else
     ketemu ← false        { x tidak ada di dalam larik L }
   endif
```

Algoritma 15.2 Prosedur pencarian beruntun (versi 1, hasil pencarian: *boolean*)

(ii) Fungsi pencarian beruntun:

```
function SeqSearch1(input L : LarikInt, input n : integer,
                    input x : integer) → boolean
{ Mengembalikan nilai true jika x ditemukan di dalam larik L[1..n],
atau nilai false jika x tidak ditemukan. }

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   i ← 1
   while (i < n ) and (L[i] ≠ x) do
      i ← i + 1
   endwhile
   { i = n or L[i] = x }
   if L[i] = x then        { x ditemukan }
     return true
   else
     return false          { x tidak ada di dalam larik L }
   endif
```

Algoritma 15.3 Fungsi pencarian beruntun (versi 1, hasil pencarian: *boolean*)

Perhatikanlah bahwa pada algoritma `SeqSearch1` di atas, pembandingan x dengan elemen larik dilakukan pada kondisi pengulangan. Apabila elemen larik yang ke-i tidak sama dengan x dan i belum sama dengan n, aktivitas pembandingan diteruskan ke elemen berikutnya (`i ← i+1`).

Pembandingan dihentikan apabila $L[i] = x$ atau indeks i sudah mencapai akhir larik ($i = n$). Perhatikan juga bahwa jika i sudah mencapai akhir larik, elemen terakhir ini belum dibandingkan dengan x. Pembandingan elemen terakhir dilakukan bersama-sama dengan penyimpulan hasil pencarian. Hasil pencarian

disimpulkan di luar kalang *while-do* dengan pernyataan `if (L[i] = x) then ...`

Pernyataan *if-then* ini juga sekaligus memeriksa apakah elemen terakhir, $L[n]$, sama dengan x. Jadi, pada algoritma `SeqSearch1` di atas, elemen terakhir diperiksa secara khusus.

Prosedur `SeqSearch1` dapat dipanggil dari program utama atau dari prosedur lain. Misal kita asumsikan prosedur `SeqSearch1` dipanggil dari program utama. Misalkan program utama bertujuan untuk memeriksa keberadaan x di dalam larik. Jika x terdapat di dalam larik maka ditampilkan pesan "ditemukan!", sebaliknya jika x tidak terdapat di dalam larik maka ditampilkan pesan "tidak ditemukan!".

Contoh program utama yang memanggil prosedur `SeqSearch1`:

```
PROGRAM Pencarian
{ Program untuk mencari nilai tertentu di dalam larik }

DEKLARASI
   const Nmaks = 100  { jumlah maksimum elemen larik }
   type LarikInt : array[1..Nmaks] of integer

   L : LarikInt
   x : integer       { elemen yang dicari }
   found : boolean   { true jika x ditemukan, false jika tidak }
   n : integer       { ukuran larik }

   procedure BacaLarik(output L : LarikInt, input n : integer )
   { Mengisi elemen larik L[1..n] dengan nilai yang dibaca dari
piranti masukan }

   procedure SeqSearch1(input L : LarikInt, input n : integer,
                        input x :integer, output ketemu : boolean)
   { Mencari keberadaan nilai x di dalam larik L[1..n]. }

ALGORITMA:
   read(n)          { tentukan banyaknya elemen larik }
   BacaLarik(L, n)  { baca elemen-elemen larik L}
   read(x)          { baca nilai yang dicari }
   SeqSearch1(L, n, x, found)    { cari }
   if found then  { found = true }
      write(x, ' ditemukan!')
   else
      write(x, ' tidak ditemukan!' )
   endif
```

Algoritma 15.4 Contoh program utama pemanggilan prosedur Pencarian beruntun

Prosedur `BacaLarik` yang disebutkan di dalam program utama di atas algoritmanya seperti di bawah ini:

```
procedure BacaLarik(output L : LarikInt, input n : integer )
{ Mengisi elemen larik L[1..n] dengan nilai yang dibaca dari piranti
  masukan.}
{ K.Awal: larik L belum terdefinisi elemen-elemennya. n sudah berisi
  jumlah elemen efektif. n diasumsikan tidak lebih besar dari
  ukuran maksimum larik (Nmaks). }
{ K.Akhir: setelah pembacaan, sebanyak n buah elemen larik L berisi
  nilai-nilai yang dibaca dari piranti masukan }
DEKLARASI
   i : integer { pencatat indeks larik }

ALGORITMA:
   for i ← 1 to n do
     read(L[i])
   endfor
```

Algoritma 15.5 Prosedur pembacaan elemen-elemen larik

Untuk pencarian beruntun yang berupa fungsi, contoh cara pemanggilan fungsi `SeqSearch1`:

```
read(x)
if not SeqSearch1(L,n,x) then
   write(x, ' tidak ditemukan!')
else
  { proses terhadap x }
  ...
endif
```

2. Hasil pencarian: indeks elemen larik yang mengandung nilai *x*.

Setiap elemen larik L dibandingkan dengan x mulai dari elemen pertama, $L[1]$. Aksi pembandingan dilakukan selama indeks larik i belum melebihi n dan $L[i]$ tidak sama dengan x. Aksi pembandingan dihentikan bila $L[i] = x$ atau $i = N$. Elemen terakhir, $L[n]$, diperiksa secara khusus. Keluaran yang dihasilkan oleh prosedur pencarian adalah peubah *idx* yang berisi indeks larik tempat x ditemukan. Jika x tidak ditemukan, *idx* diisi dengan nilai -1.

Algoritma pencarian beruntun versi 1 untuk kategori hasil berupa indeks elemen larik dapat kita tulis yakni sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch2(input L : larik, input n : integer, input x : integer,
                     output idx : integer)

{ Mencari keberadaan nilai x di dalam larik L[1..n]. }
{ K.Awal:  x dan elemen-elemen larik L[1..n] sudah terdefinisi.}
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x. Jika x tidak
  ditemukan, maka idx diisi dengan nilai -15. }

DEKLARASI
   i : integer   { pencatat indeks larik }
```

```
ALGORITMA:
   i ← 1
   while (i < n ) and (L[i] ≠ x) do
      i ← i + 1
   endwhile
   { i = n or L[i] = x }

   if L[i] = x then        { x ditemukan }
     idx ← i
   else
     idx ← -1
   endif
```

Algoritma 15.6 Prosedur pencarian beruntun (versi 1, hasil pencarian: indeks elemen)

(ii) Fungsi pencarian beruntun:

```
function SeqSearch2(input L : larik, input n : integer,
                    input x : integer) → integer

{ Mengembalikan indeks larik L[1..n] yang berisi x. Jika x tidak
 ditemukan, maka indeks yang dikembalikan adalah -1. }

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   i ← 1
   while (i < n ) and (L[i] ≠ x) do
      i ← i + 1
   endwhile
   { i = n or L[i] = x }

   if L[i] = x then        { x ditemukan }
     return i
   else
     return -1
   endif
```

Algoritma 15.7 Fungsi pencarian beruntun (versi 1, hasil pencarian: indeks elemen)

Misalkan program yang memanggil prosedur `SeqSearch2` bertujuan untuk menambahkan (*append*) nilai x ke dalam larik, namun sebelum penambahan itu harus ditentukan apakah x sudah terdapat di dalam larik. Jika x belum terdapat di dalam larik, maka x ditambahkan pada elemen ke-n+1. Karena itu kita harus memastikan bahwa penambahan satu elemen baru tidak melampaui ukuran maksimum larik (*Nmaks*). Setelah penambahan elemen baru ukuran larik efektif menjadi $n + 1$.

```
PROGRAM TambahElemenLarik
{ Program untuk menambahkan elemen baru pada ke dalam larik. Elemen
  baru dibaca dari piranti masukan, lalu dicari apakah sudah
  terdapat di dalam larik. Jika belum ada, tambahkan elemen baru
  setelah elemen terakhir. }
```

```
DEKLARASI
   const Nmaks = 100      { jumlah maksimum elemen larik }
   type LarikInt : array[1..Nmaks] of integer

   L : LarikInt
   n : integer        { ukuran larik L }
   x : integer        { elemen yang akan dicari }

   idx : integer      { mencatat indeks elemen larik yang berisi X }

   procedure BacaLarik(output L : LarikInt, input n : integer )
{      Mengisi elemen larik L[1..n] dengan data integer }

   procedure SeqSearch2(input L : LarikInt, input n : integer,
                        input x : integer, output idx : integer)
   { Mencari keberadaan nilai x di dalam larik L[1..n]. }

ALGORITMA:
   read(n)           { tentukan banyaknya elemen larik }
   BacaLarik(L,n)    { baca elemen-elemen larik L }

   read(x)
   SeqSearch2(L,n,x,idx)   { cari x sebelum ditambahkan ke dalam L }
   if idx ≠ -1 then
      write(x, ' sudah terdapat di dalam larik')

   else { x belum terdapat di dalam larik L, tambahkan x pada posisi
          ke-n+1 }
      n ← n + 1       { naikkan ukuran larik }
      L[n] ← x        { sisipkan x }
   endif
```

Algoritma 15.8 Program penambahan elemen larik

Untuk pencarian beruntun yang berupa fungsi, contoh cara pemanggilan fungsi `SeqSearch2`:

```
read(x)
idx ← SeqSearch2(L,n,x)
if idx = -1 then
   write(x, ' tidak ditemukan!')
else
  { instruksi manipulasi terhadap L[idx] }
   ...
endif
```

(a) Versi 2 (Pembandingan elemen dilakukan di dalam badan pengulangan)

Pada versi yang kedua ini, aksi pembandingan dilakukan di dalam badan pengulangan (jadi bukan di awal pengulangan seperti pada varian pertama). Untuk itu, kita memerlukan sebuah peubah *boolean* yang akan berfungsi untuk menyatakan apakah *x* sudah ditemukan. Misalkan peubah tersebut bernama

ketemu yang pada mulanya diisi nilai *false*. Bila x ditemukan pada elemen ke-i, yaitu $L[i] = x$, maka *ketemu* diisi dengan nilai *true*. Pengulangan dihentikan bila *ketemu* = *true*. Hasil pencarian disimpulkan di luar badan pengulangan.

Algoritma pencarian beruntun versi 2 lebih elegan dibandingkan dengan algoritma versi 1 karena semua elemen dibandingkan dengan cara yang sama. Algoritma pencariannya kita buat dua macam, yang pertama untuk hasil pencarian berupa berupa peubah *boolean*, dan algoritma kedua untuk hasil pencarian berupa indeks elemen larik.

2. Hasil pencarian: sebuah peubah *boolean* yang bernilai *true* bila x ditemukan atau bernilai *false* bila x tidak ditemukan.

Pada versi ini, peubah *boolean ketemu* diinisialisasi dengan nilai *false* dan indeks larrik i diisi dengan 1 (karena pembandingan dimulai dari elemen pertama). Setiap elemen L dibandingkan dengan x mulai dari elemen pertama. Jika $L[i]$ sama dengan x, peubah *ketemu* diisi nilai *true* dan pengulangan dihentikan. Sebaliknya, jika $L[i]$ tidak sama dengan x, pembandingan dilanjutkan untuk elemen berikutnya ($i \leftarrow i + 1$). Pada versi 2 ini, setiap elemen larik, termasuk elemen terakhir, diperiksa dengan cara yang sama. Keluaran yang dihasilkan adalah nilai yang disimpan di dalam peubah *ketemu*.

Algoritma pencarian beruntun versi 2 untuk kategori hasil berupa nilai *boolean* dapat kita tulis yakni sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch3(input L : LarikInt, input n : integer,
                     input x : integer, output ketemu : boolean)
{ Mencari keberadaan nilai x di dalam larik L[1..n]. }
{ K.Awal: nilai x, n, dan elemen larik L[1..n] sudah terdefinisi. }
{ K.Akhir: ketemu bernilai true jika x ditemukan. Jika x tidak
  ditemukan, ketemu bernilai false. }

DEKLARASI
   i : integer     { pencatat indeks larik }

ALGORITMA:
   i ← 1
   ketemu ← false
   while (i ≤ n) and (not ketemu) do
     if L[i] = x then
        ketemu ← true
     else
        i ← i + 1
     endif
   endwhile
   { i > n or ketemu }
```

Algoritma 15.9 Prosedur pencarian beruntun (versi 2, hasil pencarian: *boolean*)

(ii) Fungsi pencarian beruntun:

```
function SeqSearch3(input L : LarikInt, input n : integer,
                    input x : integer) → boolean
{ Mengembalikan nilai true jika x ditemukan di dalam larik L[1..n].
Jika x tidak ditemukan, nilai yang dikembalikan adalah false. }

DEKLARASI
   i : integer     { pencatat indeks larik }

ALGORITMA:
   i ← 1
   ketemu ← false
   while (i ≤ n ) and (not ketemu) do
     if L[i] = x then
        ketemu ← true
     else
        i ← i + 1
     endif
   endwhile
   { i > n or ketemu }
   return ketemu
```

Algoritma 15.10 Fungsi pencarian beruntun (versi 2, hasil pencarian: *boolean*)

2. Hasil pencarian: indeks elemen larik yang mengandung nilai *x*

Algoritmanya sama seperti `SeqSearch3` di atas, hanya saja setelah kalang *while-do* ditambahkan pernyataan *if-then* untuk memberikan hasil pencarian berupa indeks elemen larik (*idx*) yang berisi nilai *x*. Jika *x* tidak ditemukan maka *idx* diisi nilai -1.

Algoritma pencarian beruntun versi 2 untuk kategori hasil berupa indeks elemen larik dapat kita tulis yakni sebagai prosedur atau sebagai fungsi.

(i) Prosedur pencarian beruntun:

```
procedure SeqSearch4(input L : LarikInt, input n : integer,
                     input x : integer, output idx : integer)

{ Mencari keberadaan nilai X di dalam larik L[1..n]. }
{ K.Awal: x dan elemen-elemen larik L[1..N] sudah terdefinisi. }
{       K.Akhir: idx berisi indeks larik L tempat x berada. Jika x tidak
  ditemukan, maka idx diisi nilai -1. }

DEKLARASI
   i : integer          { pencatat indeks larik }
   ketemu : boolean     { true bila x ditemukan, false bila tidak }

ALGORITMA:
   i ← 1
   ketemu ← false
   while (i ≤ n ) and (not ketemu) do
     if L[i] = x then
```

```
      ketemu ← true
    else
      i ← i+1
    endif
  endwhile
  { i > n or ketemu }

  if ketemu then      { x ditemukan }
    idx ← i
  else                { x tidak ditemukan }
    idx ← -1
  endif
```

Algoritma 15.11 Prosedur pencarian beruntun (versi 2, hasil pencarian: indeks elemen)

(i) Fungsi pencarian beruntun:

```
function SeqSearch4(input L : LarikInt, input n : integer,
                    input x : integer) → integer

{ Mengembalikan indeks larik L[1..n] yang berisi x. Jika x tidak
  ditemukan, maka indeks yang dikembalikan adalah -1. }

DEKLARASI
  i : integer          { pencatat indeks larik }
  ketemu : boolean     { true bila x ditemukan, false bila tidak }

ALGORITMA:
  i ← 1
  ketemu ← false
  while (i ≤ n ) and (not ketemu) do
    if L[i] = x then
      ketemu ← true
    else
      i ← i+1
    endif
  endwhile
  { i > n or ketemu }

  if ketemu then      { x ditemukan }
    return i
  else                { x tidak ditemukan }
    return -1
  endif
```

Algoritma 15.12 Fungsi pencarian beruntun (versi 2, hasil pencarian: indeks elemen)

Kinerja Algoritma Pencarian Beruntun

Secara umum, algoritma pencarian beruntun berjalan lambat. Waktu pencarian sebanding dengan jumlah elemen larik. Misalkan larik berukuran n elemen. Maka, pada kasus di mana x tidak terdapat di dalam larik atau x ditemukan pada elemen yang terakhir, kita harus melakukan perbandingan dengan seluruh elemen larik, yang berarti jumlah perbandingan yang terjadi

sebanyak n kali. Kita katakan bahwa waktu pencarian dengan algoritma pencarian beruntun sebanding dengan n. Bayangkan bila larik berukuran 100.000 buah elemen, maka kita harus melakukan perbandingan sebanyak 100.000 buah elemen. Andaikan satu operasi perbandingan elemen larik membutuhkan waktu 0.01 detik, maka untuk 100.000 buah perbandingan diperlukan waktu sebesar 1000 detik atau 16,7 menit. Semakin banyak elemen larik, semakin lama pula waktu pencariannya. Karena itulah algoritma pencarian beruntun tidak bagus untuk volume data yang besar.

Algoritma Pencarian Beruntun pada Larik Terurut

Larik yang elemen-elemennya sudah terurut dapat meningkatkan kinerja algoritma pencarian beruntun. Jika pada larik tidak terurut jumlah perbandingan elemen larik maksimum n kali, maka pada larik terurut (dengan asumsi distribusi elemen-elemen larik adalah seragam atau *uniform*) hanya dibutuhkan rata-rata $n/2$ kali perbandingan. Hal ini karena pada larik yang terurut kita dapat segera menyimpulkan bahwa x tidak terdapat di dalam larik bila ditemukan elemen larik yang lebih besar dari x (pada larik yang terurut menaik) [TEN86].

Contoh 15.2. pencarian pada larik terurut.
(a) Diberikan larik L tidak terurut:

13	16	14	21	76	15
1	2	3	4	5	6

maka, untuk mencari 15, dibutuhkan perbandingan sebanyak 6 kali.

(b) Misalkan larik L di atas sudah diurut menaik:

13	14	15	16	21	76
1	2	3	4	5	6

maka, untuk mencari 15, dibutuhkan perbandingan hanya 3 kali (secara rata-rata).

Prosedur `SeqSearch7` berikut adalah algoritma pencarian beruntun pada larik yang terurut menaik, yang merupakan modifikasi dari algoritma `SeqSearch1` dengan perubahan pada eskpresi kondisi $L[i] \neq x$ menjadi $L[i] < x$.

```
procedure SeqSearch7(input L : LarikInt, input n : integer,
                     input x : integer, output idx : integer)

{ Mencari keberadaan nilai X di dalam larik L[1..n] yang elemen-
elemennya sudah terurut menaik. }
{ K.Awal: nilai x dan elemen-elemen larik L[1..n] sudah terdefinisi.
  Elemen-elemen larik L sudah terurut menaik. }
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x. Jika x tidak
  ditemukan, maka idx diisi dengan nilai -1. }
```

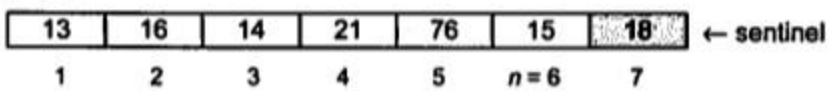
```
DEKLARASI
   i : integer          { pencatat indeks larik }
ALGORITMA:
   i ← 1
   while (i < n) and (L[i] < x) do
      i ← i + 1
   endwhile
   { i = N or L[i] = x }
   if L[i] = x then     { x ditemukan }
      idx ← i
   else
      idx ← -1
   endif
```

Algoritma 15.13 Pencarian beruntun pada larik terurut

Metode Pencarian Beruntun dengan Sentinel

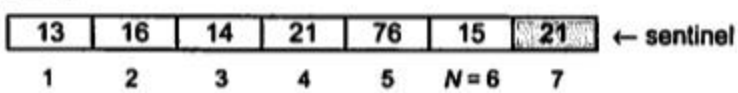
Jika pencarian bertujuan untuk menambahkan elemen baru setelah elemen terakhir larik, maka terdapat sebuah varian dari metode pencarian beruntun yang mangkus. Nilai x yang akan dicari sengaja ditambahkan terlebih dahulu pada elemen ke-n+15. Data yang ditambahkan setelah elemen terakhir larik ini disebut **sentinel**. Selanjutnya, pencarian beruntun dilakukan di dalam larik L[1..n+1]. Akibatnya, proses pencarian selalu menjamin bahwa x pasti berhasil ditemukan. Untuk menyimpulkan apakah x ditemukan pada elemen sentinel atau bukan, kita dapat mengetahuinya dengan melihat nilai *idx*. Jika *idx* = n+1 (yang berarti x ditemukan pada elemen sentinel), maka hal itu berarti bahwa x tidak terdapat di dalam larik L semula (sebelum penambahan sentinel). Keuntungannya, elemen sentinel otomatis sudah menjadi elemen yang ditambahkan ke dalam larik. Sebaliknya, jika *idx* < n +1 (yang berarti x ditemukan sebelum sentinel), maka hal itu berarti bahwa x sudah ada di dalam larik L semula.

Contoh 15.3. Pencarian beruntun dengan menggunakan sentinel.
(a) $x = 18$



18 ditemukan pada elemen ke-n+1. Sentinel otomatis sudah ditambahkan ke dalam larik. Ukuran larik sekarang = 7.

(b) $x = 21$



21 ditemukan pada elemen ke-5. Sentinel batal menjadi elemen yang ditambahkan ke dalam larik. Ukuran larik tetap 6.

Algoritma pencarian beruntun dengan sentinel kita tuliskan sebagai berikut:

```
procedure SeqSearchWithSentinel(input L : LarikInt, input n : integer,
                                input x : integer, output idx : integer)

{ Mencari x di dalam larik L[1..n] dengan menggunakan sentinel}
{ K.Awal: x dan elemen-elemen larik L[1..N] sudah terdefinisi
nilainya.}
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x.
  Jika x tidak ditemukan, maka idx diisi dengan nilai -15. }

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   L[n + 1] ← x     { sentinel }
   i ← 1
   while (L[i] ≠ x) do
      i ← i + 1
   endwhile
   { L[i] = x }    { Pencarian selalu berhasil menemukan x }

   { Kita harus menyimpulkan apakah x ditemukan pada elemen
     sentinel atau bukan }

   if idx = n + 1 then  { x ditemukan pada elemen sentinel }
      idx ← -1    { berarti x belum ada pada larik L semula }
   else     { x ditemukan pada indeks < n + 1 }
      idx ← i
   endif
```

Algoritma 15.14 Pencarian beruntun dengan sentinel

Program utama yang memanggil prosedur `SeqSearchWithSentinel` kira-kira algoritmanya sebagai berikut:

```
ALGORITMA:
   ...          { instruksi-instruksi sebelumnya }
   read(x)
   SeqSearchWithSentinel(L, n, x, idx)

   if idx ≠ -1 then
      write(x, ' sudah terdapat di dalam larik')

   else {x belum terdapat di dalam larik L semula, sentinel
         otomatis menjadi elemen yang ke-n+15. }
      n ← n + 1    { Naikkan ukuran efektif larik }
   endif
```

Algoritma 15.15 Program utama yang memanggil pencarian beruntun dengan sentinel

15.3 Algoritma Pencarian Bagidua

Pencarian pada data yang terurut menunjukkan kinerja yang lebih baik daripada pencarian pada data yang belum terurut. Hal ini, sudah kita bicarakan pada metode pencarian beruntun untuk data yang sudah terurut. Data yang terurut banyak ditemukan di dalam kehidupan sehari-hari. Data nomor telepon di dalam buku telepon misalnya, sudah terurut berdasarkan nama/instansi pelanggan telepon dari A sampai Z. Data karyawan diurut berdasarkan nomor induknya dari nomor kecil ke nomor besar. Data mahasiswa diurut berdasarkan NIM (Nomor Induk Mahasiswa), kata-kata (*entry*) di dalam kamus bahasa Inggris/Indonesia telah diurut dari A sampai Z, dan sebagainya.

Terdapat algoritma pencarian pada data terurut yang paling mangkus (*efficient*), yaitu algoritma **pencarian bagidua** atau **pencarian biner** (*binary search*). Algoritma ini digunakan untuk kebutuhan pencarian dengan waktu yang cepat. Sebenarnya, dalam kehidupan sehari-hari kita sering menerapkan pencarian bagidua. Untuk mencari arti kata tertentu di dalam kamus (misalnya kamus bahasa Inggris), kita tidak membuka kamus itu dari halaman awal sampai halaman akhir satu per satu, namun kita mencarinya dengan cara membelah atau membagi dua buku itu. Jika kata yang dicari tidak terletak di halaman pertengahan itu, kita mencari lagi di belahan bagian kiri atau belahan bagian kanan dengan cara membagi dua belahan yang dimaksud. Begitu seterusnya sampai kata yang dicari ditemukan. Hal ini hanya bisa dilakukan jika kata-kata di dalam kamus sudah terurut.

Prinsip pencarian dengan membagi data atas dua bagian mengilhami algoritma pencarian bagidua. Data yang disimpan di dalam larik harus sudah terurut. Untuk memudahkan pembahasan, selanjutnya kita misalkan elemen larik sudah terurut menurun. Dalam proses pencarian, kita memerlukan dua buah indeks larik, yaitu indeks terkecil dan indeks terbesar. Kita menyebut indeks terkecil sebagai indek ujung kiri larik dan indeks terbesar sebagai indeks ujung kanan larik. Istilah "kiri" dan "kanan" dinyatakan dengan membayangkan elemen larik terentang horizontal.

Misalkan indeks kiri adalah i dan indeks kanan adalah j. Pada mulanya, kita inisialisasi i dengan 1 dan j dengan n.

Langkah 1 : Bagi dua elemen larik pada elemen tengah. Elemen tengah adalah elemen dengan indeks $k = (i + j)$ div 2.
(Elemen tengah, $L[k]$, membagi larik menjadi dua bagian, yaitu bagian kiri $L[i..j]$ dan bagian kanan $L[k+1..j]$)

Langkah 2 : Periksa apakah $L[k] = x$. Jika $L[k] = x$, pencarian selesai sebab x sudah ditemukan. Tetapi, jika $L[k] \neq x$, harus ditentukan

apakah pencarian akan dilakukan di larik bagian kiri atau di bagian kanan. Jika $L[k] < x$, maka pencarian dilakukan lagi pada larik bagian kiri. Sebaliknya, jika $L[k] > x$, pencarian dilakukan lagi pada larik bagian kanan.

Langkah 3 : Ulangi Langkah 1 hingga x ditemukan atau $i > j$ (yaitu, ukuran larik sudah nol!)

Contoh 15.4 Ilustrasi pencarian bagidua: Misalkan diberikan larik L dengan delapan buah elemen yang sudah terurut menurun seperti di bawah ini:

81	76	21	18	16	13	10	7
i=1	2	3	4	5	6	7	8=j

(i) Misalkan elemen yang dicari adalah x = 18.

Langkah 1:
$i = 1$ dan $j = 8$
Indeks elemen tengah $k = (1 + 8)$ div $2 = 4$ (elemen yang diarsir)

81	76	21	18	16	13	10	7
1	2	3	4	5	6	7	8

kiri kanan

Langkah 2:
Pembandingan: $L[4] = 18$? Ya! (x ditemukan, proses pencarian selesai)

(ii) Misalkan elemen yang dicari adalah x = 16.

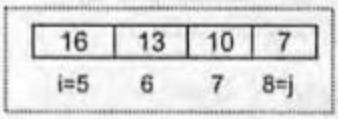
Langkah 1:
$i = 1$ dan $j = 8$
Indeks elemen tengah $k = (1 + 8)$ div $2 = 4$ (elemen yang diarsir)

81	76	21	18	16	13	10	7
1	2	3	4	5	6	7	8

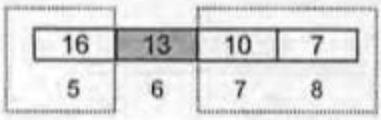
kiri kanan

Langkah 2:
Pembandingan: $L[4] = 16$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: $L[4] > 16$? Ya! Lakukan pencarian pada larik bagian kanan dengan $i = k + 1 = 5$ dan $j = 8$ (tetap)



Langkah 1':
$i = 5$ dan $j = 8$
Indeks elemen tengah $k = (5 + 8)$ div $2 = 6$ (elemen yang diarsir)

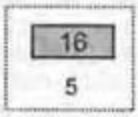


kiri' kanan'

Langkah 2':
Pembandingan: $L[6] = 16$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: $L[6] > 16$? Tidak! Lakukan pencarian pada larik bagian kiri dengan $i = 5$ (tetap) dan $j = k - 1 = 5$

Langkah 1":
$i = 5$ dan $j = 5$
Indeks elemen tengah $k = (5 + 5)$ div $2 = 5$ (elemen yang diarsir)

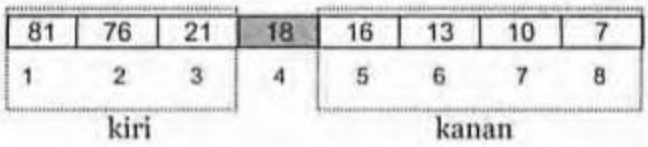


Langkah 2":
$L[5] = 16$? Ya! (x ditemukan, proses pencarian selesai)

(iii) Misalkan elemen yang dicari adalah $x = 100$

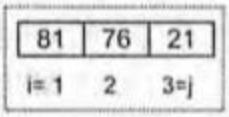
Langkah 1:
$i = 1$ dan $j = 8$

Indeks elemen tengah $k = (1 + 8)$ div $2 = 4$ (elemen yang diarsir)



Langkah 2:
Pembandingan: $L[4] = 100$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

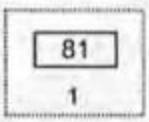
Pembandingan: $L[4] > 100$? Tidak! Lakukan pencarian pada larik bagian kiri dengan $i = 1$ (tetap) dan $j = k - 1 = 3$



Langkah 1':
$i = 1$ dan $j = 3$
Indeks elemen tengah $k = (1 + 3)$ div $2 = 2$ (elemen yang diarsir)

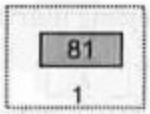
Langkah 2':
Pembandingan: $L[2] = 100$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: $L[2] > 100$? Tidak! Lakukan pencarian pada larik bagian kiri dengan $i = 1$ dan $j = k - 1 = 1$



Langkah 1":
$i = 1$ dan $j = 1$

Indeks elemen tengah $k = (1 + 1)$ div $2 = 1$ (elemen yang diarsir)



Langkah 2":
Pembandingan: $L[1] = 100$? Tidak! Harus diputuskan apakah pencarian akan dilakukan di bagian kiri atau di bagian kanan dengan pemeriksaan sebagai berikut:

Pembandingan: $L[1] > 100$? Tidak! Lakukan pencarian pada larik bagian kiri dengan $i = 1$ dan $j = k - 1 = 0$

Karena $i > j$, maka tidak ada lagi bagian larik yang tersisa. Dengan demikian, x tidak ditemukan di dalam larik. Proses pencarian dihentikan.

Algoritma pencarian bagidua pada larik *integer* yang sudah terurut menurun kita tuliskan di bawah ini, masing-masing sebagai prosedur dan sebagai fungsi.

(i) Prosedur pencarian bagidua:

```
procedure BinarySearch1(input L : LarikInt, input n : integer,
                        input x : integer, output idx : integer)

{ Mencari x di dalam larik L[1..n] yang sudah terurut menurun dengan
  metode pencarian bagidua. Keluaran prosedur ini adalah indeks elemen
larik yang mengandung nilai x; idx diisi 0 jika x tidak ditemukan. }
{ K.Awal : L[1..n] sudah berisi data yang sudah terurut menurun, dan x
adalah nilai yang akan dicari.}
{ K.Akhir: idx berisi indeks elemen larik yang mengandung nilai x;
tetapi bila x tidak ditemukan, maka idx diisi dengan -1 }

DEKLARASI
  i, j : integer       { indeks kiri dan indek kanan larik }
  k : integer          { indeks elemen tengah}
  ketemu : boolean     { flag untuk menentukan apakah X ditemukan }

ALGORITMA:
  i ← 1          { ujung kiri larik }
  j ← n          { ujung kanan larik }
  ketemu ← false    { asumsikan x belum ditemukan }
  while (not ketemu) and (i ≤ j) do
    k ←(i + j) div 2  {bagidua larik L pada posisi k}
    if (L[k] = x) then
      ketemu ← true
    else  { L[k] ≠ x }
      if (L[k] > x) then
        { Lakukan pencarian pada larik bagian kanan, set indeks ujung
          kiri larik yang baru }
        i ← k + 1
      else
        { Lakukan pencarian pada larik bagian kiri, set indeks ujung
          kanan larik yang baru }
```

```
        j ← k - 1
      endif
   endif
endwhile
{ ketemu = true or i > j}

if ketemu then  { x ditemukan }
   idx ← k
else            { x tidak ditemukan di dalam larik  }
   idx ← -1
endif
```

Algoritma 15.16 Prosedur pencarian bagidua pada larik yang terurut menurun

(ii) Fungsi pencarian bagidua:

```
function BinarySearch1(input L : LarikInt, input n : integer,
                       input x : integer) → integer

{ Mengembalikan indeks elemen larik L[1..n] yang mengandung nilai x;
bila x tidak ditemukan, maka indek yang dikembalikan adalah -1 }

DEKLARASI
  i, j : integer      { indeks kiri dan indek kanan larik }
  k : integer         { indeks elemen tengah}
  ketemu : boolean    { flag untuk menentukan apakah X ditemukan }

ALGORITMA:
  i ← 1       { ujung kiri larik }
  j ← n       { ujung kanan larik }
  ketemu ← false    { asumsikan x belum ditemukan }
  while (not ketemu) and (i ≤ j) do
    k ←(i + j) div 2  {bagidua larik L pada posisi k}
    if (L[k] = x) then
      ketemu ← true
    else  { L[k] ≠ x }
      if (L[k] > x) then
        { Lakukan pencarian pada larik bagian kanan, set indeks ujung
          kiri larik yang baru }
        i ← k + 1
      else
        { Lakukan pencarian pada larik bagian kiri, set indeks ujung
          kanan larik yang baru }
        j ← k - 1
      endif
   endif
endwhile
{ ketemu = true or i > j}

if ketemu then  { x ditemukan }
   return k
else            { x tidak ditemukan di dalam larik  }
   return -1
   endif
```

Algoritma 15.17 Prosedur pencarian bagidua pada larik yang terurut menurun

Untuk algoritma pencarian bagidua pada larik yang sudah terurut menaik, kita hanya perlu mengganti pernyataan `if (L[k] > x)` dengan `if L[k] < x)`.

Kinerja Algoritma Pencarian Bagidua

Kita dapat melihat bahwa pada setiap kali pencarian, larik dibagidua menjadi dua bagian yang berukuran hampir sama. Pada setiap pembagian, elemen tengah dibandingkan apakah sama dengan x (`if (L[k] = x)`. Pada kasus terburuk, yaitu pada kasus x tidak terdapat di dalam larik atau x ditemukan setelah ukuran larik tinggal 1 elemen, larik akan dibagi sebanyak $^2\log(n)$ kali, sehingga jumlah pembandingan yang dibutuhkan adalah sebanyak $^2\log(n)$ kali. Kita katakan bahwa waktu pencarian sebanding dengan $^2\log(n)$ [WIR76]. Untuk $n = 256$ elemen misalnya, kasus terburuk menghasilkan pembagian larik sebanyak $^2\log(256) = 8$ kali, yang berarti jumlah pembandingan elemen adalah 8 kali (bandingkan dengan metode pencarian beruntun yang pada kasus terburuk melakukan pembandingan sebanyak 256 kali). Jadi, untuk larik yang terurut, algoritma pencarian bagidua jauh lebih cepat daripada algoritma pencarian beruntun.

15.4 Pencarian pada Larik Terstruktur

Algoritma pencarian yang dibahas sebelum ini menggunakan larik dengan elemen-elemen bertipe sederhana. Pada sebagian besar kasus, elemen larik sering bertipe terstruktur. Contoh, misalkan *M* adalah sebuah larik yang elemennya menyatakan nilai ujian seorang mahasiswa untuk suatu mata kuliah (*MK*) yang ia ambil. Data setiap mahasiswa adalah *NIM* (Nomor Induk Mahasiswa), nama mahasiswa, mata kuliah yang ia ambil, dan nilai mata kuliah tersebut. Deklarasi nama dan tipe sebagai berikut:

```
DEKLARASI
  const Nmaks = 100
  type Mahasiswa : record <NIM : integer,   { Nomor Induk Mahasiswa }
                           NamaMhs : string,  { nama mahasiswa }
                           KodeMK  : string,  { kode mata kuliah }
                           Nilai: char  { indeks nilai MK (A/B/C/D/E) }
                          >

  type TabMhs : array[1..Nmaks] of Mahasiswa

  M : TabMhs
```

Algoritma 15.18 Pendeklarasian larik terstruktur

Struktur lojik larik *M* ditunjukkan pada Gambar 15.2.

TabMhs

	NIM	NamaMhs	KodeMK	Nilai
1	29801	Heru Satrio	MA211	A
2	29804	Amirullah Satya	FI351	B
3				
.				
.				
100	29887	Yanti Siregar	TL321	C

Gambar 15.2 Larik *M* dengan 100 elemen. Setiap elemen larik bertipe terstruktur (*record*). Tiap *record* terdiri atas *field NIM, field Nama, field MK,* dan *field Nilai.*

Misalkan pencarian data didasarkan pada *NIM*, maka proses pembandingan dilakukan terhadap *field NIM* saja. Algoritma pencarian beruntun dan algoritma pencarian bagidua untuk larik tertstruktur diberikan di bawah ini (untuk algoritma pencarian beruntun, kita gunakan algoritma Versi 2 dengan hasil pencarian adalah indeks larik, sedangkan untuk algoritma pencarian bagidua kita gunakan pencarian pada larik yang terurut menaik).

(a) Algoritma Pencarian Beruntun

```
procedure CariNIM_1(input M : TabMhs, input n : integer,
                    input NIMmhs : integer, output idx : integer)

{ Mencari keberadaan NIMmhs di dalam larik M[1..n] dengan metode
pencarian beruntun. }
{ K.Awal : nilai NIMmhs, n, dan elemen larik M[1..n] sudah terdefinisi.}
{ K.Akhir: idx berisi indeks larik tempat NIMmhs ditemukan,
  idx = -1 jika NIMmhs tidak ditemukan }

DEKLARASI
   i : integer     { pencatat indeks larik }
   ketemu : boolean

ALGORITMA:
   i ← 1
   ketemu ← false
   while (i ≤ n ) and (not ketemu) do
     if M[i].NIM = NIMmhs then
       ketemu ← true
     else
       i ← i + 1
     endif
   endwhile
    { i > n or ketemu }
        if ketemu then { NIMmhs ditemukan }
     idx ← i
    else    { NIMmhs tidak ditemukan }
      idx ← -1
    endif
```

Algoritma 15.19 Pencarian beruntun pada larik terstruktur

(b) Algoritma Pencarian Bagidua

```
procedure CariNIM_2(input M : TabMhs, input n : integer,
                    input NIMmhs : integer, output IDX : integer)

{ Mencari NIMmhs di dalam larik M[1..n] yang sudah terurut menaik
berdasarkan NIM dengan metode pencarian bagidua. }
{ K.Awal : Larik M[1..n] sudah berisi data yang sudah terurut menaik,
dan NIMmhs adalah nilai yang akan dicari }
{ K.Akhir: idx berisi indeks larik tempat NIMmhs ditemukan, idx = -1
jika NIMmhs tidak ditemukan }

DEKLARASI
  i, j : integer          { indeks kiri dan indeks kanan }
  k : integer             { indeks elemen tengah }
  ketemu : boolean        { flag untuk menentukan ketemu atau tidak }

ALGORITMA:
i ← 1       { ujung kiri larik }
j ← n       { ujung kanan larik }
ketemu ← false   { asumsikan x belum ditemukan }

while (not ketemu) and (i ≤ j) do

   k ← (i + j) div 2  { bagidua larik L pada posisi k }
   if (M[k].NIM = NIMmhs) then
     ketemu ← true
   else  { M[k].NIM ≠ NIMmhs }

      if  (M[k].NIM < NIMmhs) then
        { Lakukan pencarian pada larik bagian kanan, set indeks
          ujung kiri larik yang baru }
        i ← k + 1
      else
        { Lakukan pencarian pada larik bagian kiri, set indeks
          ujung kanan larik yang baru }
        j ← k - 1
      endif
   endif
endwhile
{ ketemu = true or i > j}

if (ketemu) then  { NIMmhs ditemukan }
   idx ← k
else              { NIMmhs tidak ditemukan di dalam larik  }
   idx ← -1
  endif
```

Algoritma 15.20 Pencarian bagidua pada larik terstruktur

15.5 Pakai yang Mana? Algoritma Pencarian Beruntun atau Pencarian Bagidua?

Kedua algoritma pencarian ini mempunyai kelebihan dan kekurangan masing-masing. Algoritma pencarian beruntun dapat digunakan baik untuk data yang belum terurut maupun untuk data yang sudah terurut. Sebaliknya, algoritma pencarian bagidua hanya dapat digunakan untuk data yang sudah terurut saja.

Ditinjau dari kinerja pencarian, kita sudah mengetahui bahwa untuk kasus terburuk (yaitu jika pencarian gagal menemukan x), algoritma pencarian beruntun memerlukan waktu yang sebanding dengan n (banyaknya data), sedangkan algoritma pencarian membutuhkan waktu yang sebanding dengan $^2\log(n)$. Karena $^2\log(n) < n$ untuk $n > 1$, maka jika n semakin besar waktu pencarian dengan algoritma bagidua jauh lebih sedikit daripada waktu pencarian dengan algoritma beruntun. Karena itulah, algoritma pencarian bagidua lebih baik untuk mencari data pada sekumpulan nilai yang sudah terurut ketimbang algoritma pencarian beruntun.

Sebagai perbandingan antara algoritma pencarian beruntun dengan pencarian bagidua, tinjaulah kasus di mana x tidak ditemukan di dalam larik yang sudah terurut. Misalkan,

(a) untuk larik yang berukuran $n = 256$ elemen
- algoritma pencarian beruntun melakukan pembandingan elemen larik sebanyak 256 kali,
- algoritma pencarian bagidua melakukan pembandingan sebanyak $^2\log(256) = 8$ kali,

(b) untuk larik yang berukuran $n = 1024$ elemen
- algoritma pencarian beruntun melakukan pembandingan elemen larik sebanyak 1024 kali,
- algoritma pencarian bagidua melakukan pembandingan sebanyak $^2\log(1024) = 10$ kali.

15.6 Pencarian pada Larik yang Tidak Bertipe Numerik

Meskipun algoritma-algoritma pencarian yang sudah dikemukakan di atas diterapkan pada larik *integer*, mereka tetap benar untuk larik data yang bertipe bukan numerik misalnya data bertipe karakter, maupun *string*. Aculah kembali dari Buku 1 bahwa operasi perbandingan (dengan operator $<$, $>$, $\neq$)

juga berlaku pada tipe data karakter maupun *string*. Jadi, pembandingan karakter seperti 'a' < 'b' atau pembandingan *string* seperti 'budi' < 'iwan' adalah ekspresi relasional yang sah.

15.7 Algoritma Pencarian Beruntun dan Pencarian Bagidua dalam Bahasa PASCAL dan Bahasa C

ALGORITMIK

```
DEKLARASI
  const Nmaks = 100      {jumlah maksimum elemen larik }
  type LarikInt : array [1..Nmaks] of integer
```

PASCAL

```
(* DEKLARASI *)
   const Nmaks = 100;      {jumlah maksimum elemen larik }
   type LarikInt = array [1..Nmaks] of integer;
```

C

```
/* DEKLARASI */
    #define Nmaks  100                  /* jumlah elemen larik */
    typedef int LarikInt [Nmaks+1];    /* indeks larik dimulai dari 0
            sampai Nmaks, tetapi elemen ke-0 tidak akan digunakan*/
```

1. Algoritma Pencarian Beruntun

ALGORITMIK

```
procedure SeqSearch2(input L : LarikInt, input n : integer,
                     input x : integer, output idx : integer)

{ Mencari keberadaan nilai x di dalam larik L[1..n]. }
{ K.Awal:  x dan elemen-elemen larik L[1..N] sudah terdefinisi.}
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x. Jika x tidak
 ditemukan, maka idx diisi dengan nilai -15. }

DEKLARASI
   i : integer    { pencatat indeks larik }

ALGORITMA:
   i ← 1
   while (i < n ) and (L[i] ≠ x) do
      i ← i + 1
   endwhile
   { i = n or L[i] = x }

   if L[i] = x then       { x ditemukan }
```

```
    idx ← i
  else
    idx ← -1
  endif
```

PASCAL

```
procedure SeqSearch2(L : LarikInt; n : integer; x : integer;
                     var idx : integer);
{ Mencari keberadaan nilai x di dalam larik L[1..N]. }
{ K.Awal:  x dan elemen-elemen larik L[1..N] sudah terdefinisi.}
{ K.Akhir: idx berisi indeks larik L yang berisi nilai x. Jika x tidak
  ditemukan, maka idx diisi dengan nilai -15. }

var
   i : integer;     { pencatat indeks larik }

begin
   i := 1;
   while (i < n ) and (L[i] <> x) do
      i := i + 1;
   {endwhile}
   {i = n or L[i] = x }

    if L[i] = x then    { x ditemukan }
      idx := i
    else
      idx := -1;        { x tidak ditemukan }
   {endif}
end;
```

C

```
void SeqSearch2(LarikInt L, int n, int x, int *idx)
/* Mencari keberadaan nilai X di dalam larik L[1..N]. */
/* K.Awal: nilai x dan elemen-elemen larik L[1..N] sudah terdefinisi. */
/* K.Akhir:     idx berisi indeks larik L yang berisi x. Jika X tidak
   ditemukan, idx diisi dengan nilai -1. */
{
   int i;       /* pencatat indeks larik */

   i = 1;
   while (i < n && L[i] != x)
      i++;
   /* endwhile */
   /* i == n or L[i] == x */

   if (L[i] == x)     /* x ditemukan */
     *idx = i;
   else               /* x tidak ditemukan */
     *idx = -1;
   /*endif*/

}
```

2. Algoritma Pencarian Bagidua

ALGORITMIK

```
procedure BinarySearch2(input L : LarikInt, input n : integer,
                        input x : integer, output idx : integer)

{ Mencari  x di dalam larik L[1..n] yang sudah terurut menurun dengan
metode pencarian bagidua. Keluaran prosedur ini adalah indeks elemen
berisi nilai x.  }
{ K.Awal : Larik L[1..n] sudah berisi data yang sudah terurut menurun,
  dan x adalah harga yang akan dicari. }
{ K.Akhir: idx berisi indeks larik tempat x ditemukan, idx = -1 jika x
  tidak ditemukan }

DEKLARASI
  i,j : integer          { indeks kiri dan indek kanan larik }
  k : integer            { indeks elemen tengah}
  ketemu : boolean       { pertanda untuk menentukan apakah x ditemukan }

ALGORITMA:
  i ← 1     { ujung kiri larik }
  j ← n     { ujung kanan larik }
  ketemu ← false    { asumsikan x belum ditemukan }
  while (not ketemu) and (i ≤ j) do
    k ← (i + j) div 2  { bagidua larik L pada posisi k }
    if (L[k] = x) then
      ketemu ← true
    else  { L[k] ≠ x }
      if (L[k] > x) then
        { Lakukan pencarian pada larik bagian kanan, set indeks
          ujung kiri larik yang baru }
        i ← k + 1
      else
        { Lakukan pencarian pada larik bagian kiri, set indeks
          ujung kanan larik yang baru }
        j ← k - 1
      endif
    endif
endwhile
{ ketemu = true or i > j}

if (ketemu) then  { x ditemukan }
   idx ← k
else              { X tidak ditemukan di dalam larik  }
   idx ← -1
endif
```

PASCAL

```
procedure BinarySearch2(L : LarikInt; n : integer; x : integer;
                        var idx : integer);

{ Mencari x di dalam larik L[1..n] yang sudah terurut menurun dengan
metode pencarian bagidua. Keluaran prosedur ini adalah indeks elemen
berisi nilai x. }
{ K.Awal : Larik L[1..N] sudah berisi data yang sudah terurut menurun,
  dan x adalah harga yang akan dicari. }
{ K.Akhir: idx berisi indeks larik tempat x ditemukan; idx = -1 jika x
  tidak ditemukan. }

var
  i,j : integer;        { indeks kiri dan indek kanan larik }
  k : integer;          { indeks elemen tengah}
  ketemu : boolean;     { flag untuk menentukan apakah X ditemukan }

begin
  i := 1;    { ujung kiri larik }
  j := n;    { ujung kanan larik }
  ketemu := false;  { asumsikan x belum ditemkan }
  while (not ketemu) and (i <= j) do
   begin
    k:=(i + j) div 2;  { bagidua larik L pada posisi k }
    if (L[k] = x) then
      ketemu:= true
    else  { L[k] <> x }
      if  (L[k] > x) then
        { Lakukan pencarian pada larik bagian kanan, set indeks
          ujung kiri larik yang baru }
         i := k + 1
      else
        { Lakukan pencarian pada larik bagian kiri, set indeks
          ujung kanan larik yang baru }
         j := k - 1;
      {endif}
   {endif}
  end{while};
  { ketemu = true or i > j}

  if (ketemu) then  { x ditemukan }
    idx := k
  else              { x tidak ditemukan di dalam larik  }
   idx := -1;
 {endif}
end;
```

C

```
void BinarySearch2(LarikInt L, int n, int X, int *idx)

/* Mencari x di dalam larik L[1..n] yang sudah terurut menurun dengan
   metode pencarian bagidua. Keluaran prosedur ini adalah indeks
elemen berisi nilai x.  */
```

```
/* K.Awal : larik L[1..n] sudah berisi data yang sudah terurut
menurun,dan x adalah harga yang akan dicari. */
/* K.Akhir: idx berisi indeks larik tempat X ditemukan; idx = -1 jika
x tidak ditemukan */

{
  typedef enum{true = 1, false = 0} boolean; /* deklarasi tipe boolean
*/

  int i,j;            /* indeks kiri dan indeks kanan larik */
  int k;              /* indeks elemen tengah */
  boolean ketemu;     /* flag untuk menentukan apakah X ditemukan */

  i = 1;             /* ujung kiri larik */
  j = n;             /* ujung kanan larik */
  ketemu = false;    /* asumsikan x belum ditemukan */
  while (!ketemu && i <= j)
  {
   k = (i + j)/2;  /* bagidua larik L pada posisi k */
   if (L[k] == x)
     ketemu = true;
   else  /* L[k] != x */

      if  (L[k] > x)
        /* akan dilakukan pencarian pada larik bagian kanan, set
        indeks ujung kiri larik yang baru */
         i = k + 1;
      else
        /* akan dilakukan pencarian pada larik bagian kiri,
           set indeks ujung kanan larik yang baru */
         j = k - 1;
      /*endif*/
 /*endif*/
}
/* endwhile*/
/* ketemu = true || i > j */

if (ketemu)    /* x ditemukan */
   *idx = k;
else          /* x tidak ditemukan di dalam larik  */
   *idx = -1;
/*endif*/
}
```

Soal Latihan Bab 15

1. Tulislah kembali algoritma pencarian beruntun yang memberikan hasil indeks elemen larik yang mengandung *x*, tetapi pencarian dimulai dari elemen terakhir.

2. [TEN86] Algoritma *SeqSearch8* berikut hanya membutuhkan satu ekspresi perbandingan pada awal kalang *while-do*. Realisasikan algoritma tersebut ke dalam bahasa *Pascal* dan bahasa *C*.

```
procedure SeqSearch8(input L : LarikInt, input n : integer,
                     input x : integer, output idx : integer)

{ Mencari keberadaan nilai X di dalam larik L[1..N]. }
{ K.Awal: x dan elemen-elemen larik L[1..N] sudah terdefinisi
  nilainya.}
{ K.Akhir: idx berisi indeks larik L yang berisi x; jika x tidak
  ditemukan, idx diisi dengan nilai -1. }

DEKLARASI
   i   : integer   { pencatat indeks larik }
   pos : integer

ALGORITMA:
   i ← 1
   pos ← n + 1
   while (i ≠ pos) do
     if x = L[i] then
       pos ← i
     else
       i ← i + 1
     endif
   endwhile
   { i = pos }
   if i > n then      { x tidak ditemukan }
     idx ← -1
   else               { x ditemukan }
     idx ← pos
   endif
```

3. Masalah apa yang terjadi jika algoritma pencarian beruntun kita tulis seperti berikut ini:

```
i ← 1
while (i ≤ n ) and (L[i] ≠ x) do
  i ← i + 1
endwhile
{ i > n or L[i] = x }

if L[i] = x then   { x ditemukan }
  idx ← i
else               { x tidak ditemukan }
  idx ← -1
endif
```

4. Nyatakan algoritma pencarian bagidua berupa fungsi yang mengembalikan indeks elemen larik yang berisi nilai *x*.

5. [PAR95] [TEN86] Metode **pencarian interpolasi** (*interpolation search*) merupakan varian dari metode pencarian bagidua, di mana elemen berikutnya yang akan dibandingkan didasarkan pada nilai indek *i* dan *j*. Misalnya jika *x* "diperkirakan" terletak pada posisi bagian antara

L[*i*] dan *L*[*j*], maka kita set *k* dengan nilai bagian antara *i* dan *j*. Memperkirakan posisi *x* di dalam larik disebut menginterpolasi. Rumus untuk menghitung *k* kita tulis sebagai berikut:

```
k ← i + (j - i)*(x - L[i]) div (L[j] - L[i])
```

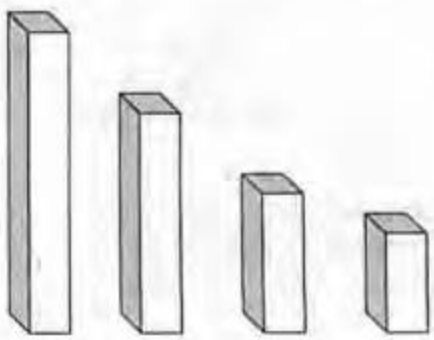
(Metode pencarian interpolasi mengasumsikan data di dalam larik terdistrubusi *uniform*. Metode ini pada dasarnya sama dengan metode pencarian bagidua, kecuali dalam penentuan elemen yang diperiksa berikutnya). Tulislah algoritma pencarian interpolasi.

16
Algoritma Pengurutan

Mengurutkan kartu (Sumber: www.infodesign.com.au)

Di dalam Bab 15 kita sudah membicarakan salah satu algoritma pencarian, yaitu algoritma pencarian bagidua. Algoritma pencarian bagidua hanya dapat diterapkan jika elemen-elemen larik sudah terurut lebih dahulu. Di dalam Bab 16 ini akan dikemukakan beberapa algoritma pengurutan data.

Masalah pengurutan merupakan persoalan yang menarik, karena terdapat puluhan algoritma pengurutan yang pernah dikemukakan orang. Tidak semua algoritma pengurutan akan dibahas di dalam Bab ini, hanya beberapa pengurutan sederhana yang akan dijelaskan. Kelak algoritma pengurutan yang lebih "maju" akan diberikan pada pembahasan algoritma tingkat lanjut pada seri buku algoritma yang lain.



16.1 Masalah Pengurutan

Pengurutan (*sorting*) adalah proses mengatur sekumpulan objek menurut urutan atau susunan tertentu [WIR76]. Masalah pengurutan dapat ditulis sebagai berikut:

Diberikan larik L dengan n elemen yang sudah terdefinisi elemen-elemennya. Urutan larik tersebut sehingga tersusun secara menaik (ascending):
$L[1] \leq L[2] \leq L[3] \leq ... \leq L[n]$

atau secara menurun *(descending):*
$L[1] \geq L[2] \geq L[3] \geq ... \geq L[n]$

Data yang diurut dapat berupa data bertipe dasar atau tipe terstruktur (*record*). Jika data bertipe terstruktur, maka harus dispesifikasikan berdasarkan *field* apa data tersebut diurutkan. *Field* yang dijadikan dasar pengurutan dikenal sebagai *field* kunci.

Berikut ini diberikan beberapa contoh data yang terurut:

i. 23, 27, 45, 67, 100, 130, 501
(data bertipe *integer* terurut menaik)

ii. 50.27, 31.009, 20.3, 19.0, -5.2, -10.9
(data bertipe riil terurut menurun)

iii. 'Amir', 'Badu', 'Budi', 'Dudi', 'Eno', 'Rudi', 'Zamzami'
(data bertipe *string* terurut menaik)

iv. 'd', 'e', 'g', 'i', 'x'
(data bertipe karakter terurut menaik)

v. <13596001, 'Eko', 'A'>, <13596006, 'Rizka', 'B'>,
<13596007, 'Hamdi', 'D'>, <13596010, 'Rizal', 'C'>,
<13596012, 'Ratna', 'B'>
(data mahasiswa bertipe tertsruktur terurut menaik berdasarkan *field NIM*)

Dalam kehidupan sehari-hari, *entry* di dalam buku telepon, kata di dalam kamus bahasa/istilah, dan *entry* di dalam ensiklopedi selalu terurut secara alfabetik. Kita juga sering melakukan pengurutan seperti mencatat alamat teman berdasarkan nama, menyusun tumpukan koran berdasarkan tanggal, mengantri di pasar swalayan berdasarkan urutan waktu kedatangan di kasir, dan sebagainya.

Data yang sudah terurut memiliki beberapa keuntungan. Selain mempercepat waktu pencarian, dari data yang terurut kita dapat langsung memperoleh nilai maksimum dan nilai minimum. Untuk data numerik yang terurut menurun, nilai maksimum adalah elemen pertama larik, dan nilai minimum adalah elemen terakhir larik. Hal ini bermanfaat untuk mengetahui juara kelas misalnya, atau mengetahui peserta ujian Seleksi Penerimaan Mahasiswa Baru (SPMB) yang memperoleh skor nilai tertinggi.

Pengurutan dikatakan **stabil** jika dua atau lebih data yang sama (atau identik) tetap pada urutan yang sama setelah pengurutan. Misalnya di dalam sekelompok data *integer* berikut terdapat 3 buah nilai 12 (diberi tanda petik ', ", dan '" untuk mengidentifikasi urutannya.

70, 12', 45, 10, 12", 60, 12'", 33, 50

Jika suatu metode pengurutan menghasilkan susunan data terurut seperti berikut:

10, 12', 12", 12'", 33, 45, 50, 60, 70

maka pengurutannya dikatakan stabil dan metode pengurutannya disebut metode stabil. Tetapi jika suatu metode pengurutan menghasilkan susunan data terurut seperti contoh berikut:

10, 12", 12', 12'", 33, 45, 50, 60, 70

maka pengurutan dan metodenya kita katakan **tidak stabil**.

Kestabilan pengurutan mungkin penting atau tidak penting [PAR 95]. Misalnya kita ingin menyusun data buku yang terurut secara alfabetik berdasarkan nama pengarang sekaligus terurut berdasarkan judul buku oleh pengarang tersebut (pengarang bisa menulis lebih dari satu buku). Jika kita menggunakan metode pengurutan yang stabil, maka kita urut berdasarkan judul buku lebih dahulu baru kemudian kita urut berdasarkan nama pengarang. Jika kita menggunakan metode pengurutan yang tidak stabil, maka kita tidak memperoleh hasil pengurutan yang kita inginkan.

16.2 Algoritma Pengurutan

Adanya kebutuhan terhadap proses pengurutan memunculkan bermacam-macam algoritma pengurutan. Banyak algoritma pengurutan yang telah ditemukan. Hal ini menunjukkan bahwa persoalan pengurutan adalah persoalan yang kaya dengan solusi algoritmik. Algoritma pengurutan yang sering ditemukan di dalam literatur-literatur komputer antara lain:

1. *Bubble Sort*
2. *Selection Sort* (*Maximum Sort* dan *Minimum Sort*)

3. *Insertion Sort*
4. *Heap Sort*
5. *Shell Sort*
6. *Quick Sort*
7. *Merge Sort*
8. *Radix Sort*
9. *Tree Sort*

Di dalam Bab 16 ini kita tidak akan membahas semua algoritma pengurutan tersebut, tetapi hanya empat buah algoritma pengurutan yang sederhana saja, yaitu:

1. Metode Pengurutan Apung (*Bubble Sort*)
2. Metode Pengurutan Seleksi (*Selection Sort*)
3. Metode Pengurutan Sisip (*Insertion Sort*)
4. Metode Pengurutan *Shell* (*Shell Sort*)

Dua algoritma pertama (*bubble* dan *selection sort*) melakukan prinsip pertukaran elemen dalam proses pengurutan sehingga keduanya dinamakan pengurutan dengan pertukaran (*exchange sorts*), sedangkan dua algoritma terakhir melakukan prinsip geser dan sisip elemen dalam proses pengurutan (*shift and insert sorts*). Semua algoritma pengurutan selalu melakukan operasi perbandingan elemen larik untuk menemukan posisi urutan yang tepat.

Algoritma pengurutan yang lain membutuhkan beberapa konsep pengetahuan pendahuluan yang tidak dicakup di dalam buku ini. Misalnya *Heap Sort* dan *Tree Sort* memerlukan pengetahuan konsep pohon (*tree*), *Quick Sort* dan *Merge Sort* membutuhkan pengetahuan konsep *divide and conquer* dan algoritma rekursif. Untuk pemahaman konsep pengurutan, empat buah algoritma sederhana di atas sudah cukup untuk diketahui.

Seperti halnya pada pencarian, algoritma pengurutan juga dapat diklasifikasikan sebagai algoritma pengurutan internal dan algoritma pengurutan eksternal.

1. Algoritma pengurutan internal, yaitu algoritma pengurutan untuk data yang disimpan di dalam memori komputer. Umumnya struktur internal yang dipakai untuk pengurutan internal adalah larik, sehingga pengurutan internal disebut juga pengurutan larik.
2. Algoritma pengurutan eksternal, yaitu metode pengurutan untuk data yang disimpan di dalam *disk storage*, disebut juga pengurutan arsip (*file*), karena struktur eskternal yang dipakai adalah arsip.

Di dalam Bab 16 ini kita hanya membicarakan algoritma pengurutan internal saja. Algoritma pengurutan eksternal merupakan bahasan tersendiri yang tidak dicakup di dalam buku ini.

Untuk semua algoritma pengurutan yang akan dijelaskan di dalam Bab 16 ini, kita menggunakan tipe data larik yang didefinisikan di bagian deklarasi sebagai berikut:

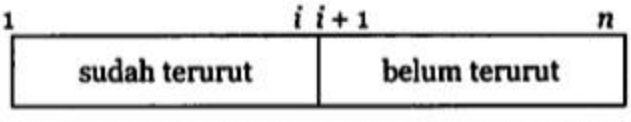
```
DEKLARASI
  const Nmaks = 1000   { jumlah maksimum elemen larik }
  type LarikInt = array [1..Nmaks] of integer
```

Algoritma 16.1 Deklarasi larik *integer* yang digunakan di dalam Bab 16 ini

16.3 Algoritma Pengurutan Apung

Algoritma pengurutan apung (*bubble sort*) diinspirasi oleh gelembung sabun yang berada di atas permukaan air. Karena berat jenis gelembung sabun lebih ringan daripada berat jenis air, maka gelembung sabun selalu terapung ke atas permukaan. Secara umum, benda-benda yang berat akan terbenam dan benda-benda yang ringan akan terapung ke atas permukaan.

Prinsip pengapungan di atas juga digunakan pada pengurutan apung. Apabila kita menginkan larik terurut menaik, maka elemen larik yang berharga paling kecil "diapungkan", artinya diangkat ke "atas" (atau ke ujung kiri larik) melalui proses pertukaran. Proses pengapungan ini dilakukan sebanyak $n - 1$ langkah (satu langkah disebut juga satu kali *pass*) dengan n adalah ukuran larik. Pada akhir setiap langkah ke-i, larik $L[1..n]$ akan terdiri atas dua bagian yaitu bagian yang sudah terurut, yaitu $L[1..i]$, dan bagian yang belum terurut, $L[i+1..n]$ (Gambar 16.1). Setelah langkah terakhir, diperoleh larik $L[1..n]$ yang terurut menaik.



Gambar 16.1 Bagian larik yang terurut dan belum terurut pada metode pengurutan apung.

Algoritma Pengurutan Apung

Untuk mendapatkan larik yang terurut menaik, algoritma pengurutan apung secara global sebagai berikut:

Untuk setiap *pass* $i = 1, 2, ..., n - 1$, lakukan:
Mulai dari elemen $k = n, n - 1, ..., i + 1$, lakukan:
1.1 Bandingkan $L[k]$ dengan $L[k - 1]$.
1.2 Pertukarkan $L[k]$ dengan $L[k - 1]$ jika $L[k] < L[k - 1]$.

Rincian setiap *pass* sebagai berikut:

***Pass* 1** : Mulai dari elemen ke-$k = n, n\text{-}1, ..., 2$, bandingkan $L[k]$ dengan $L[k-1]$. Jika $L[k] < L[k-1]$, pertukarkan $L[k]$ dengan $L[k-1]$.
Pada akhir langkah 1, elemen $L[1]$ berisi harga minimum pertama.

***Pass* 2** : Mulai dari elemen ke-$k = n, n-1, ..., 3$, bandingkan $L[k]$ dengan $L[k-1]$. Jika $L[k] < L[k-1]$, pertukarkan $L[k]$ dengan $L[k-1]$.
Pada akhir langkah 2, elemen $L[2]$ berisi harga minimum kedua, larik $L[1..2]$ terurut, sedangkan $L[3..n]$ belum terurut.

***Pass* 3** : Mulai dari elemen ke-$k = n, n-1, ..., 4$, bandingkan $L[k]$ dengan $L[k-1]$. Jika $L[k] < L[k-1]$, pertukarkan $L[k]$ dengan $L[k-1]$.
Pada akhir langkah 3, elemen $L[3]$ berisi harga minimum ketiga, larik $L[1..3]$ terurut, sedangkan $L[4..n]$ belum terurut.

⋮

***Pass* $n-1$** : Mulai dari elemen ke-$k = n$, bandingkan $L[k]$ dengan $L[k-1]$. Jika $L[k] < L[k-1]$, pertukarkan $L[k]$ dengan $L[k-1]$.

Pada akhir *pass* $n-1$, elemen $L[n-1]$ berisi nilai minimum ke-$(n-1)$ dan larik $L[1..n-1]$ terurut menaik (elemen yang tersisa adalah $L[n]$, tidak perlu diurut karena hanya satu-satunya).

Tinjau larik L dengan $n = 6$ buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menaik:

25	27	10	8	76	21
1	2	3	4	5	6

← arah pembandingan

Pass 1:

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
$k = 6$	$L[6] < L[5]$? (21 < 76?)	Ya	25, 27, 10, 8, 21, 76
$k = 5$	$L[5] < L[4]$? (21 < 8?)	Tidak	25, 27, 10, 8, 21, 76
$k = 4$	$L[4] < L[3]$? (8 < 10?)	Ya	25, 27, 8, 10, 21, 76
$k = 3$	$L[3] < L[2]$? (8 < 27?)	Ya	25, 8, 27, 10, 21, 76
$k = 2$	$L[2] < L[1]$? (8 < 25?)	Ya	8, 25, 27, 10, 21, 76

Hasil akhir *pass* 1:

8	25	27	10	21	76
1	2	3	4	5	6

Pass 2: (berdasarkan hasil akhir pass 1)

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
$k = 6$	$L[6] < L[5]$? (76 < 21?)	Tidak	**8**, 25, 27, 10, 21, 76
$k = 5$	$L[5] < L[4]$? (21 < 10?)	Tidak	**8**, 25, 27, 10, 21, 76
$k = 4$	$L[4] < L[3]$? (10 < 27?)	Ya	**8**, 25, 10, 27, 21, 76
$k = 3$	$L[3] < L[2]$? (10 < 25?)	Ya	**8**, 10, 25, 27, 21, 76

Hasil akhir *pass* 2:

8	10	25	27	21	76
1	2	3	4	5	6

Pass 3: (berdasarkan hasil akhir *pass* 2)

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
k = 6	$L[6] < L[5]$? (76 < 21?)	Tidak	**8, 10**, 25, 27, 21, 76
k = 5	$L[5] < L[4]$? (21 < 27?)	Ya	**8, 10**, 25, 21, 27, 76
k = 4	$L[4] < L[3]$? (21 < 25?)	Ya	**8, 10**, 21, 25, 27, 76

Hasil akhir pass 3:

8	10	21	25	27	76
1	2	3	4	5	6

Pass 4: (berdasarkan hasil akhir *pass* 3)

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
$k = 6$	$L[6] < L[5]$? (76 < 27?)	Tidak	**8, 10, 21**, 25, 27, 76
$k = 5$	$L[5] < L[4]$? (27 < 25?)	Tidak	**8, 10, 21**, 25, 27, 76

Hasil akhir *pass* 4:

8	10	21	25	27	76
1	2	3	4	5	6

Pass 5: (berdasarkan hasil akhir *pass* 4)

k	Elemen yang dibandingkan	Pertukarkan?	Hasil Sementara
K = 6	$L[6] < L[5]$? (76 < 27?)	Tidak	**8, 10, 21, 25**, 27, 76

Hasil akhir pass 5:

8	10	21	25	27	76
1	2	3	4	5	6

Hasil akhir *pass* 5 menyisakan satu elemen (yaitu 76) yang tidak perlu diurutkan lagi, maka pengurutan selesai. Larik *L* sekarang sudah terurut menaik!

```
procedure BubbleSort1 (input/output L : LarikInt, input n : integer)
{ Mengurutkan larik L[1..n] sehingga terurut menaik dengan metode
pengurutan apung. }
{ K.Awal : Elemen larik L sudah terdefenisi nilai-nilainya. }
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
                L[1] ≤ L[2] ≤ ... ≤ L[n]. } }

DEKLARASI
  i    : integer    { pencacah untuk jumlah langkah }
  k    : integer    { pencacah,untuk pengapungan pada setiap langkah }
  temp : integer    { peubah bantu untuk pertukaran }

ALGORITMA:
   for i ← 1 to n - 1 do
     for k ← n downto i + 1 do
        if L[k] < L[k-1] then

          {pertukarkan L[k] dengan L[k-1]}
           temp ← L[k]
           L[k] ← L[k-1]
           L[k-1] ← temp
         endif
     endfor
   endfor
```

Algoritma 16.2 Pengurutan apung (menaik)

Kita dapat menuliskan proses pertukaran (*temp*← *L*[*k*], *L*[*k*]←*L*[*k* – 1], *L*[*k* – 1]←*temp*) sebagai sebuah prosedur `Tukar` sebagai berikut:

```
procedure Tukar (input/output a : integer, input/output b : integer)
{ Mempertukarkan nilai a dan b
{ K.Awal : a dan b sudah terdefinisi nilai-nilainya.
{ K.Akhir: b berisi nilai a sebelum pertukaran, dan b berisi nilai a
  sebelum pertukaran. }

DEKLARASI
   temp : integer        { peubah bantu untuk pertukaran }

ALGORITMA:
   temp ← a
   a ← b
  b ← temp
```

Algoritma 16.3 Prosedur mempertukarkan nilai a dan b

Dengan memanfaatkan prosedur `Tukar`, algoritma pengurutan apung dapat disederhanakan penulisannya sebagai berikut:

```
procedure BubbleSort1 (input/output L : LarikInt, input n : integer)
{ Mengurutkan larik L[1..n] sehingga terurut menaik dengan metode
 pengurutan apung. }
```

```
{ K.Awal : Elemen larik L sudah terdefenisi nilai-nilainya. }
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
          L[1] ≤ L[2] ≤ ... ≤ L[n]. } }

DEKLARASI
   i : integer      { pencacah untuk jumlah langkah }
   k : integer      { pencacah,untuk pengapungan pada setiap langkah }

   procedure Tukar (input/output a : integer, input/output b : integer)
   { Mempertukarkan nilai a dan b }

ALGORITMA:
   for i ← 1 to n - 1 do
     for k ← n downto i + 1 do
        if L[k] < L[k - 1] then
           {pertukarkan L[k] dengan L[k - 1]}
           Tukar(L[k], L[k - 1])
        endif
     endfor
   endfor
```

Algoritma 16.4 Pengurutan apung (menaik) dengan menggunakan prosedur Tukar

Untuk memperoleh larik yang terurut menurun, kita melakukan proses sebaliknya, yaitu "melemparkan" elemen yang berharga maksimum ke "atas" (atau ke ujung "kiri" larik). Perhatikan algoritmanya di bawah ini.

```
procedure BubbleSort2 (input/output L : LarikInt, input n : integer)
{ Mengurutkan larik L[1..n] sehingga terurut menurun dengan metode
pengurutan apung. }
{ K.Awal : Elemen larik L sudah terdefinisi nilai-nilainya. }
{ K.Akhir: Elemen larik L terurut menurun sedemikian sehingga
          L[1] ≥ L[2] ≥ ... ≥ L[n].}

DEKLARASI
   i : integer      { pencacah untuk jumlah langkah }
   k : integer      { pencacah,untuk pengapungan pada setiap langkah }

    procedure Tukar (input/output a : integer, input/output b : integer)
   { Mempertukarkan nilai a dan b }

ALGORITMA:
    for i ← 1 to n - 1 do
       for k ← n downto i + 1 do
          if L[k] > L[k-1] then
             {pertukarkan L[k] dengan L[k-1]}
             Tukar(L[k], L[k-1])
          endif
       endfor
    endfor
```

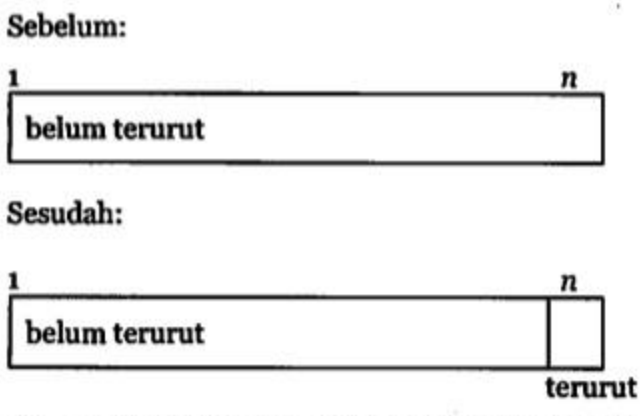
Algoritma 16.5 Pengurutan apung (menurun)

Komentar Mengenai Pengurutan Apung

Pengurutan apung merupakan algoritma pengurutan yang tidak mangkus (*efficient*). Hal ini, disebabkan oleh banyaknya operasi pertukaran yang dilakukan pada setiap langkah pengapungan. Untuk ukuran larik yang besar, pengurutan dengan algoritma ini membutuhkan waktu yang lama. Karena alasan itu, maka algoritma pengurutan apung jarang digunakan dalam praktek pemrograman. Namun, kelebihan algoritma ini adalah pada kesederhanaanya dan mudah dipahami.

16.4 Algoritma Pengurutan Seleksi

Algoritma pengurutan ini disebut pengurutan seleksi (*selection sort*) karena gagasan dasarnya adalah memilih elemen maksimum/minimum dari larik, lalu menempatkan elemen maksimum/minimum itu pada awal atau akhir larik (elemen terujung) (lihat Gambar 16.2). Selanjutnya elemen terujung tersebut "diisolasi" dan tidak disertakan pada proses selanjutnya. Proses yang sama diulang untuk elemen larik yang tersisa, yaitu memilih elemen maksimum/minimum berikutnya dan mempertukarkannya dengan elemen terujung larik sisa. Sebagaimana halnya pada algoritma pengurutan gelembung, proses memilih nilai maksimum/minimum dilakukan pada setiap *pass*. Jika larik berukuran n, maka jumlah *pass* adalah $n - 1$.



Gambar 16.2 Bagian larik yang terurut dan belum terurut pada algoritma Pengurutan Seleksi.

Ada dua varian algoritma pengurutan seleksi ditinjau dari pemilihan elemen maksimum/ minimum, yaitu:

1. Algoritma pengurutan seleksi-maksimum, yaitu memilih elemen maksimum sebagai basis pengurutan.
2. Algoritma pengurutan seleksi-minimum, yaitu memilih elemen minimum sebagai basis pengurutan.

Algoritma Pengurutan Seleksi-Maksimum

Untuk mendapatkan larik yang terurut menaik, algoritma pengurutan seleksi-maksimum ditulis secara garis besar sebagai berikut:

1. JumlahPass = n − 1
2. Untuk setiap pass i = 1, 2, ..., JumlahPass lakukan:
   2.1 cari elemen terbesar (*maks*) mulai dari elemen ke-1 sampai elemen ke-*n*;
   2.2 pertukarkan *maks* dengan elemen ke-*n*;
   2.3 kurangi *n* satu (karena elemen ke-*n* sudah terurut).

Rincian aksi pada setiap *pass*, seperti di bawah ini:

***Pass* 1** : Cari elemen maksimum di dalam $L[1..n]$.
Pertukarkan elemen maksimum dengan elemen $L[n]$.
Ukuran larik yang belum terurut = $n - 1$.

***Pass* 2** : Cari elemen maksimum di dalam $L[1..n-1]$.
Pertukarkan elemen maksimum dengan elemen $L[n-1]$.
Ukuran larik yang belum terurut = $n - 16$.

***Pass* 3** : Cari elemen maksimum di dalam $L[1..n-2]$.
Pertukarkan elemen maksimum dengan elemen $L[n-2]$.
Ukuran larik yang belum terurut = $n - 3$.

⋮

***Pass* $n - 1$** : Tentukan elemen maksimum di dalam $L[1..2]$
Pertukarkan elemen maksimum dengan elemen $L[2]$
Ukuran larik yang belum terurut = 1.

Setelah *pass* $n - 1$, elemen yang tersisa adalah $L[1]$, tidak perlu diurutkan lagi karena hanya satu-satunya.

Tinjau larik dengan $n = 6$ buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menaik dengan metode pengurutan seleksi-maksimum:

29	27	10	8	76	21
1	2	3	4	5	6

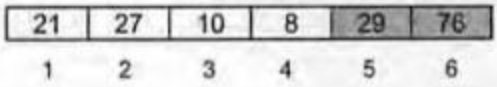
***Pass* 1:**

Cari elemen maksimum di dalam larik $L[1..6] \Rightarrow$ hasilnya: *maks* = $L[5] = 76$
Pertukarkan *maks* dengan $L[n]$, diperoleh:

29	27	10	8	21	76
1	2	3	4	5	6

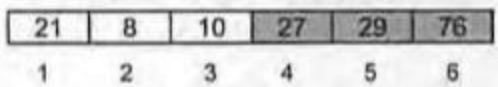
***Pass* 2:**

(berdasarkan susunan larik hasil *pass* 1)
Cari elemen maksimum di dalam larik $L[1..5] \Rightarrow$ hasilnya: $maks = L[1] = 29$
Pertukarkan *Maks* dengan $L[5]$, diperoleh:



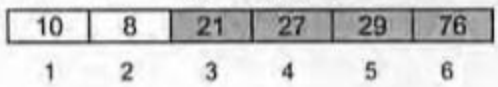
***Pass* 3:**

(berdasarkan susunan larik hasil *pass* 2)
Cari elemen maksimum di dalam larik $L[1..4] \Rightarrow$ hasilnya: $maks = L[2] = 27$
Pertukarkan *maks* dengan $L[4]$, diperoleh:



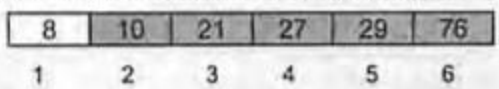
***Pass* 4:**

(berdasarkan susunan larik hasil *pass* 3)
Cari elemen maksimum di dalam larik $L[1..3] \Rightarrow$ hasilnya: $maks = L[1] = 21$
Pertukarkan *maks* dengan $L[3]$, diperoleh:



***Pass* 5:**

(berdasarkan susunan larik hasil *pass* 4)
Cari elemen maksimum di dalam larik $L[1..2] \Rightarrow$ hasilnya: $maks = L[1] = 10$
Pertukarkan *maks* dengan $L[2]$, diperoleh:



Tersisa satu elemen (yaitu 8), maka pengurutan selesai. Larik *L* sudah terurut menaik!

Jadi, pada setiap *pass* pengurutan terdapat proses mencari harga maksimum dan proses pertukaran dua buah elemen larik. Ingat, algoritma mencari elemen maksimum sudah pernah kita bicarakan di dalam Bab Larik.

Algoritma pengurutan seleksi-maksimum selengkapnya sebagai berikut:

```
procedure SelectionSort1(input/output L : LarikInt, input n:integer)
{ Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menaik
dengan metode pengurutan seleksi-maksimum.
```

```
{ K.Awal : Elemen larik L sudah terdefinisi harganya. }
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
           L[1] ≤ L[2] ≤ ... ≤ L[n] }

DEKLARASI
i     : integer    { pencacah pass }
j     : integer    { pencacah untuk mencari nilai maksimum }
imaks : integer    { indeks yang berisi nilai maksimum sementara }
maks  : integer    { elemen maksimum }
temp  : integer    { peubah bantu untuk pertukaran }

ALGORITMA:
for i ← n downto 2 do   { jumlah pass sebanyak n - 1 }

   { cari elemen maksimum pada elemen L[1..i] }
   imaks ← 1       { elemen pertama diasumsikan sebagai elemen
                     maksimum sementara }
   maks ← L[1]          { elemen maksimum }

   for j ← 2 to i do
     if L[j] > maks then
        imaks ← j
        maks ← L[j]
     endif
   endfor

   {pertukarkan maks dengan L[I]}
   temp ← L[i]
   L[I] ← maks
   L[Imaks] ← Temp

endfor
```

Algoritma 16.6 Pengurutan seleksi-maksimum (menaik)

Perhatikanlah bahwa pada algoritma `SelectionSort1` di atas kita dapat menghilangkan penggunaan peubah `maks`, karena yang kita perlukan sebenarnya adalah indeks elemen larik yang mengandung nilai maksimum tersebut. Jika indeks itu kita catat, maka elemen lariknya dapat diacu melalui indeks tersebut. Dengan demikian, algoritma pengurutan seleksi-maksimum dapat dibuat menjadi lebih elegan sebagai berikut:

```
procedure SelectionSort1(input/output L : LarikInt, input n:integer)
{ Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menaik
dengan metode pengurutan seleksi-maksimum. }
{ K.Awal : Elemen larik L sudah terdefinisi harganya. }
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
           L[1] ≤ L[2] ≤ ... ≤ L[n] }

DEKLARASI
i     : integer        { pencacah pass }
j     : integer        { pencacah untuk mencari nilai maksimum }
imaks : integer        { indeks yang berisi nilai maksimum sementara }
temp  : integer        { peubah bantu untuk pertukaran }
```

```
ALGORITMA:
for i ← n downto 2 do    { jumlah pass sebanyak n - 1 }

   { cari elemen maksimum pada elemen L[1..i] }
   imaks ← 1          {elemen pertama diasumsikan sebagai elemen
                      maksimum sementara }
   for j ← 2 to i do

     if L[j] > L[imaks] then
        imaks ← j
     endif
   endfor
   {pertukarkan L[imaks] dengan L[i]}
   temp ← L[i]
   L[i] ← L[imaks]
   L[imaks] ← temp

endfor
```

Algoritma 16.7 Pengurutan pilih maksimum tanpa peubah Maks

Untuk seterusnya, pada pencarian elemen maksimum/minimum yang kita catat hanya indeks elemen larik saja.

Seperti halnya pada pengurutan gelembung, prosedur `Tukar` dapat digunakan untuk mengganti runtunan aksi pertukaran pada prosedur `SelectionSort1` tadi.

```
procedure SelectionSort1(input/output L : LarikInt, input n:integer)
{ Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menaik
dengan metode pengurutan seleksi-maksimum. }
{ K.Awal : Elemen larik L sudah terdefinisi harganya. }
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
          L[1] ≤ L[2] ≤ ... ≤ L[n] }

DEKLARASI
i    : integer        { pencacah pass }
J    : integer        { pencacah untuk mencari nilai maksimum }
Imaks : integer       { indeks yang berisi nilai maksimum sementara }

   procedure Tukar (input/output a : integer, input/output b : integer)
  { Mempertukarkan nilai a dan b }

ALGORITMA:
for i ← n downto 2 do    { jumlah pass sebanyak n - 1 }

   { cari elemen maksimum pada elemen L[1..i] }
   imaks ← 1          {elemen pertama diasumsikan sebagai elemen
                       maksimum sementara }
   for j ← 2 to i do
     if L[j] > L[imaks] then
        imaks ← j
     endif
   endfor
```

```
    {pertukarkan L[imaks] dengan L[i]}
    Tukar(L[imaks], L[i])

  endfor
```

Algoritma 16.8 Pengurutan seleksi-maksimum dengan prosedur Tukar

Apabila diinginkan larik yang terurut menurun, maka algoritma pengurutan seleksi-maksimum sebagai berikut:

Untuk setiap *pass* $i = 1, 2, ..., n - 1$ lakukan:
1. cari elemen terbesar (*maks*) mulai dari elemen ke-i sampai elemen ke-n;
2. pertukarkan *maks* dengan elemen ke-i.

Rincian setiap *pass* sebagai berikut:

***Pass* ke-1** : Cari elemen maksimum di dalam $L[1..n]$.
Pertukarkan elemen maksimum dengan elemen $L[1]$.

***Pass* ke-2** : Cari elemen maksimum di dalam $L[16..n]$.
Pertukarkan elemen maksimum dengan elemen $L[2]$.

***Pass* ke-3** : Cari elemen maksimum di dalam $L[3..n]$.
Pertukarkan elemen maksimum dengan elemen $L[3]$.

⋮

***Pass* $n - 1$** : Cari elemen maksimum di dalam $L[n - 1.. n]$.
Pertukarkan elemen maksimum dengan elemen $L[n - 1]$.
Ukuran larik yang belum terurut = 1.

Setelah *pass* $n - 1$, elemen yang tersisa adalah $L[n]$, tidak perlu diurut karena hanya satu-satunya.

Tinjau larik dengan $n = 6$ buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menurun dengan metode pengurutan seleksi-maksimum

29	27	10	8	76	21
1	2	3	4	5	6

***Pass* 1:**

Cari elemen maksimum di dalam larik $L[1..6] \Rightarrow$ hasilnya: $imaks = 5$, $L[imaks] = 76$
Pertukarkan $L[imaks]$ dengan $L[1]$, diperoleh:

76	27	10	8	29	21
1	2	3	4	5	6

***Pass* 2:**

(berdasarkan susunan larik hasil *pass* 1)
Cari elemen maksimum di dalam larik $L[16..6] \Rightarrow$ hasilnya: *imaks* = 5, $L[imaks] = 29$
Pertukarkan $L[imaks]$ dengan $L[2]$, diperoleh:

76	29	10	8	27	21
1	2	3	4	5	6

***Pass* 3:**

(berdasarkan susunan larik hasil *pass* 2)
Cari elemen maksimum di dalam larik $L[3..6] \Rightarrow$ hasilnya: *imaks* = 5, $L[imaks] = 27$
Pertukarkan $L[imaks]$ dengan $L[3]$, diperoleh:

76	29	27	8	10	21
1	2	3	4	5	6

***Pass* 4:**

(berdasarkan susunan larik hasil *pass* 3)
Cari elemen maksimum di dalam larik $L[4..6] \Rightarrow$ hasilnya: *imaks* = 6, $L[imaks]=21$
Pertukarkan $L[imaks]$ dengan $L[4]$, diperoleh:

76	29	27	21	10	8
1	2	3	4	5	6

***Pass* 5:**

(berdasarkan susunan larik hasil *pass* 4)
Cari elemen maksimum di dalam larik $L[5..6] \Rightarrow$ hasilnya: *imaks* = 5, $L[imaks] = 10$
Perukarkan $L[imaks]$ dengan $L[5]$ (sebenarnya tidak perlu dilakukan sebab 10 sudah berada pada posisi yang tepat), diperoleh:

76	29	27	21	10	8
1	2	3	4	5	6

Tersisa satu elemen (yaitu 8), maka pengurutan selesai. Larik *L* sudah terurut menurun!

```
procedure SelectionSort2 (input/output L : LarikInt, input n:integer)
{       Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun
menurun dengan metode pengurutan seleksi-maksimum. }
{ K.Awal : Elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir: Elemen larik L terurut menurun sedemikian sehingga
           L[1] ≥ L[2] ≥ … ≥ L[n] }
```

```
DEKLARASI
i     : integer        { pencacah pass }
j     : integer        { pencacah untuk mencari nilai maksimum }
imaks : integer        { indeks yang berisi nilai maksimum sementara }

    procedure Tukar (input/output a : integer, input/output b : integer)
    { Mempertukarkan nilai a dan b }

ALGORITMA:
for i ← 1 to n - 1 do

  { cari indeks elemen maksimum di dalam L[I..n] }
   imaks ← i
   for j ← i + 1 to n do
     if L[j] > L[imaks] then
        imaks ← j
     endif
   endfor

   {pertukarkan L[imaks] dengan L[i]}
   Tukar(L[imaks], L[i])

endfor
```

Algoritma 16.9 Pengurutan seleksi-maksimum (menurun)

Algoritma Pengurutan Seleksi-Minimum

Berbeda dengan algoritma pengurutan seleksi-maksimum, maka pada algoritma pengurutan seleksi-minimum, basis pencarian adalah elemen minimum (terkecil). Elemen minimum ditempatkan di awal larik (agar larik terurut menaik) atau ditempatkan di akhir larik (agar larik terurut menurun).

Algoritma seleksi-minimum untuk memperoleh larik yang terurut menaik secara garis besar ditulis sebagai berikut:

Untuk setiap *pass* $i = 1, 2, ..., n - 1$ lakukan:
1. cari elemen terkecil (*min*) mulai dari elemen ke-*i* sampai elemen ke-*n;*
2. pertukarkan *min* dengan elemen ke-*i*.

Rincian setiap *pass* sebagai berikut:

***Pass* 1:** Cari elemen minimum di dalam $L[1..n]$
Pertukarkan elemen terkecil dengan elemen $L[1]$

***Pass* 2:** Cari elemen minimum di dalam L[16..*n*]
Pertukarkan elemen terkecil dengan elemen $L[2]$

Pass 3: Cari elemen minimum di dalam $L[3..n]$
Pertukarkan elemen terkecil dengan elemen $L[3]$

⋮

Pass *n*-1 : Cari elemen minimum di dalam $L[n\text{-}1..n]$
Pertukarkan elemen terkecil dengan elemen $L[N\text{-}1]$

(elemen yang tersisa adalah $L[n]$, tidak perlu diurut karena hanya satu-satunya)

Tinjau larik dengan $n = 6$ buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menaik dengan algoritma pengurutan seleksi-minimum:

29	27	10	8	76	21
1	2	3	4	5	6

***Pass* 1:**

Cari elemen terkecil di dalam larik $L[1..6] \Rightarrow$ hasilnya: $imin = 4$, $L[imin] = 8$
Pertukarkan $L[imin]$ dengan $L[1]$, diperoleh:

8	27	10	29	76	21
1	2	3	4	5	6

***Pass* 2:**

(berdasarkan susunan larik hasil *pass* 1)
Cari elemen terkecil di dalam larik $L[16..6] \Rightarrow$ hasilnya: $imin = 3$, $L[imin] = 10$
Pertukarkan $L[imin]$ dengan $L[2]$, diperoleh:

8	10	27	29	76	21
1	2	3	4	5	6

***Pass* 3:**

(berdasarkan susunan larik hasil *pass* 2)
Cari elemen terkecil di dalam larik L[3..6] $\Rightarrow$ hasilnya: imin = 6, L[imin] = 21
Tukar $L[imin]$ dengan $L[3]$, diperoleh:

8	10	21	29	76	27
1	2	3	4	5	6

***Pass* 4:**

(berdasarkan susunan larik hasil *pass* 3)
Cari elemen terkecil di dalam larik $L[4..6] \Rightarrow$ hasilnya: $imin = 6$, $L[imin] = 27$
Pertukarkan $L[imin]$ dengan $L[4]$, diperoleh:

8	10	21	27	76	29
1	2	3	4	5	6

***Pass* 5:**

(berdasarkan susunan larik hasil *pass* 4)
Cari elemen terkecil di dalam larik $L[5..6] \Rightarrow$ hasilnya: $imin = 6$, $L[imin] = 29$
Pertukarkan $L[imin]$ dengan $L[5]$, diperoleh:

8	10	21	27	29	76
1	2	3	4	5	6

Tersisa satu elemen (yaitu 76), maka pengurutan selesai. Larik *L* sudah terurut menaik!

```
procedure SelectionSort3(input/output L : LarikInt, input n : integer)
{ Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menaik
dengan metode pengurutan seleksi-minimum. }
{ K.Awal : Elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
           L[1] ≤ L[2] ≤ ... ≤ L[n] }

DEKLARASI
i    : integer        { pencacah pass }
j    : integer        { pencacah untuk mencari nilai maksimum }
imin : integer        { indeks yang berisi nilai maksimum sementara }

   procedure Tukar (input/output a : integer, input/output b : integer)
  { Mempertukarkan nilai a dan b }

ALGORITMA:
for i ← 1 to n - 1 do

   { cari indeks dari elemen minimum di dalam larik L[I..N] }
   imin ← i
   for j ← i + 1 to n do
     if L[j] < L[imin] then
        imin ← j
     endif
   endfor

   {pertukarkan L[imin] dengan L[i]}
   Tukar(L[imin], L[i])

endfor
```

Algoritma 16.10 Pengurutan seleksi-minimum (menaik)

Apabila diinginkan elemen-elemen larik terurut menurun, maka algoritma pengurutan seleksi-minimum sebagai berikut:

```
procedure SelectionSort4(input/output L : LarikInt, input n:integer)
{ Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menurun
dengan metode pengurutan seleksi-minimum. }
{ K.Awal : Elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir: Elemen larik L terurut menurun sedemikian sehingga
           L[1] ≥ L[2] ≥ ... ≥ L[n] }
```

```
DEKLARASI
i    : integer        { pencacah pass }
j    : integer        { pencacah untuk mencari nilai maksimum }
imin : integer        { indeks yang berisi nilai minimum sementara }

   procedure Tukar (input/output a : integer, input/output b : integer)
   { Mempertukarkan nilai a dan b }

ALGORITMA:
  for i ← n downto 2 do    { jumlah pass sebanyak n - 1 kali }

   { cari elemen terkecil di dalam L[1..i] }
   imin ← 1
   for j ← 2 to i do
     if L[j] < L[imin] then
        imin ← L[j]
     endif
   endfor

   {pertukarkan L[imin] dengan L[i]}
   Tukar(L[imin], L[i]);

 endfor
```

Algoritma 16.11 Pengurutan seleksi-minimum (menurun)

Tinjau larik dengan $n = 6$ buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menurun dengan metode pengurutan seleksi-minimum:

29	27	10	8	76	21
1	2	3	4	5	6

***Pass* 1:**

Cari elemen terkecil di dalam larik $L[1..6] \Rightarrow imin = 4$, $L[imin] = 8$
Pertukarkan $L[imin]$ dengan $L[n]$, diperoleh:

29	27	10	21	76	8
1	2	3	4	5	6

***Pass* 2:**

(berdasarkan susunan larik hasil *pass* 1)
Cari elemen terkecil di dalam larik $L[1..5] \Rightarrow imin = 3$, $L[imin] = 10$
Pertukarkan $L[imin]$ dengan $L[5]$, diperoleh:

29	27	76	21	10	8
1	2	3	4	5	6

***Pass* 3:**

(berdasarkan susunan larik hasil pass 2)
Cari elemen terkecil di dalam larik $L[1..4] \Rightarrow imin = 4$, $L[imin] = 21$
Pertukarkan $L[imin]$ dengan $L[4]$ (sebenarnya tidak perlu dilakukan, karena 21 sudah berada pada posisi yang tepat) diperoleh:

29	27	76	21	10	8
1	2	3	4	5	6

***Pass* 4:**

(berdasarkan susunan larik hasil *pass* 3)
Cari elemen minimum di dalam larik $L[1..3] \Rightarrow imin = 2$, $L[imin] = 27$
Pertukarkan $L[imin]$ dengan $L[3]$, diperoleh:

29	76	27	21	10	8
1	2	3	4	5	6

***Pass* 5:**

(berdasarkan susunan larik hasil *pass* 4)
Cari elemen minimum di dalam larik $L[1..2] \Rightarrow imin = 1$, $L[imin] = 29$
Pertukarkan $L[imin]$ dengan $L[2]$, diperoleh:

76	29	27	21	10	8
1	2	3	4	5	6

Tersisa satu elemen (yaitu 76), maka pengurutan selesai. Larik L sudah terurut menurun!

Komentar Mengenai Algoritma Pengurutan Seleksi

Dibanding dengan algoritma pengurutan apung, algoritma pengurutan seleksi memiliki kinerja yang lebih baik. Alasannya, operasi pertukaran elemen hanya dilakukan sekali saja pada setiap *pass*, dengan demikian lama pengurutannya berkurang dibandingkan dengan metode pengurutan gelembung.

16.5 Algoritma Pengurutan Sisip

Dari namanya, pengurutan sisip (*insertion sort*) adalah metode pengurutan dengan cara menyisipkan elemen larik pada posisi yang tepat. Pencarian posisi yang tepat dilakukan dengan menyisir larik. Selama penyisiran dilakukan pergeseran elemen larik. Metode pengurutan sisip cocok untuk persoalan menyisipkan elemen baru ke dalam sekumpulan elemen yang sudah terurut.

Algoritma Pengurutan Sisip untuk Pengurutan Menaik

Untuk mendapatkan larik yang terurut menaik, algoritma pengurutan sisip secara garis besar diitulis sebagai berikut:

Untuk setiap *pass* $i = 2, ..., n$ lakukan:
1. $y \leftarrow L[i]$
2. sisipkan y pada tempat yang sesuai di antara $L[1]$ ... $L[i]$

Rincian setiap *pass* sebagai berikut:

Asumsikan: $L[1]$ dianggap sudah pada tempatnya

Pass 2 : Elemen $y = L[2]$ harus dicari tempatnya yang tepat di dalam $L[1..2]$ dengan cara menggeser elemen $L[1..1]$ ke kanan (atau ke bawah, jika anda membayangkan larik terentang vertikal) bila $L[1..1]$ lebih besar daripada $L[2]$. Misalkan posisi yang tepat adalah k. Sisipkan $L[2]$ pada $L[k]$.

Pass 3 : Elemen $y = L[3]$ harus dicari tempatnya yang tepat di dalam $L[1..3]$ dengan cara menggeser elemen $L[1..2]$ ke kanan (atau ke bawah) bila $L[1..2]$ lebih besar daripada $L[3]$. Misalkan posisi yang tepat adalah k. Sisipkan $L[3]$ pada $L[k]$.

⋮

Pass n : Elemen $y = L[n]$ harus dicari tempatnya yang tepat di dalam $L[1..n]$ dengan cara menggeser elemen $L[1..n-1]$ ke kanan (atau ke bawah) bila $L[1..n-1]$ lebih besar daripada $L[n]$. Misalkan posisi yang tepat adalah k. Sisipkan $L[n]$ pada $L[k]$.

Hasil dari *pass n*: Larik $L[1..n]$ sudah terurut menaik, yaitu
$L[1] \leq L[2] \leq ... \leq L[n]$.

Tinjau larik dengan $n = 6$ buah elemen di bawah ini yang belum terurut. Larik ini akan diurut menaik dengan metode pengurutan sisip:

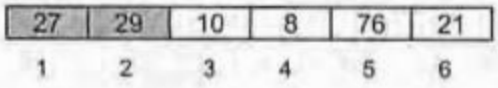
29	27	10	8	76	21
1	2	3	4	5	6

Asumsikan: Elemen $y = L[1] = 29$ dianggap sudah terurut.

29	27	10	8	76	21
1	2	3	4	5	6

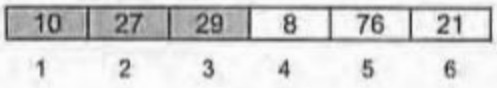
***Pass* 2:**

(berdasarkan susunan larik pada akhir *pass* 1)
Cari posisi yang tepat untuk $y = L[2] = 27$ di dalam $L[1..2]$, diperoleh:



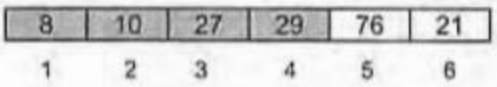
***Pass* 3:**

(berdasarkan susunan larik pada akhir *pass* 2)
Cari posisi yang tepat untuk $y = L[3] = 10$ di dalam $L[1..3]$, diperoleh:



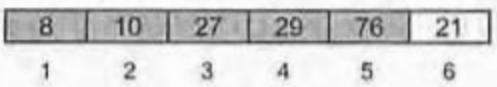
***Pass* 4:**

(berdasarkan susunan larik pada akhir *pass* 3)
Cari posisi yang tepat untuk $y = L[4] = 8$ di dalam $L[1..4]$, diperoleh:



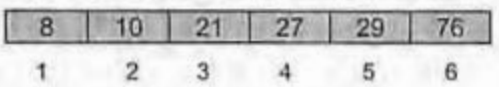
***Pass* 5:**

(berdasarkan susunan larik pada akhir *pass* 4)
Cari posisi yang tepat untuk $y = L[5] = 76$ di dalam $L[1..5]$, diperoleh:



***Pass* 6:**

(berdasarkan susunan larik pada akhir *pass* 5)
Cari posisi yang tepat untuk $y = L[6] = 21$ di dalam $L[1..6]$, diperoleh:



Algoritma pengurutan sisip selengkapnya sebagai berikut:

```
procedure InsertionSort1(input/output L : LarikInt, input n : integer)
{Mengurutkan elemen-elemen larik L[1..n] sehingga tersusun menaik
dengan  metode pengurutan sisip. }
{ K.Awal  :  Elemen-elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir :  Elemen-elemen larik L terurut menaik sedemikian sehingga
             L[1] ≤ L[2] ≤ ... ≤ L[n] }
```

```
DEKLARASI
i : integer       {    pencacah pass }
j : integer       {    pencacah untuk penelusuran larik }
y : integer       {    peubah bantu agar L[K] tidak ditimpa selama
pergeseran }
   ketemu : boolean { untuk menyatakan posisi penyisipan ditemukan }

ALGORITMA:  { elemen L[1] dianggap sudah terurut }
  for i ← 2 to n do  { mulai dari pass 2 sampai pass N }

     y ← L[i]
    { cari posisi yang tepat untuk y di dalam L[1..i-1] sambil
      menggeser}
    j ← i - 1
    ketemu ← false

    while (j ≥ 1) and (not ketemu) do
      if y < L[j] then
         L[j + 1] ← L[j]     { geser }
         j ← j-1
      else
         ketemu←true
      endif
    endwhile
    { j < 1 or ketemu }
    L[j+1] ← y  { sisipkan y pada tempat yang sesuai }

  endfor
```

Algoritma 16.12 Pengurutan sisip (menaik)

Algoritma Pengurutan Sisip untuk Pengurutan Menurun

Algoritma pengurutan sisip untuk memperoleh elemen larik yang terurut menurun, seperti berikut ini:

```
procedure InsertionSort2(input/output L : LarikInt, input n : integer)
{ Mengurutkan elemen larik L[1..n] sehingga tersusun menurun dengan
metode pengurutan sisip. }
{ K.Awal  : Elemen-elemen larik L sudah terdefinisi nilainya.
{ K.Akhir : Elemen-elemen larik L terurut menurun sedemikian sehingga
            L[1] ≥ L[2] ≥ ... ≥ L[n] }

DEKLARASI
i : integer          {      pencacah pass }
j : integer          {      pencacah untuk penelusuran larik }
y : integer          {      peubah bantu agar L[K] tidak ditimpa
selama pergeseran }
   ketemu : boolean  { untuk menyatakan posisi penyisipan ditemukan }

ALGORITMA:
  { elemen L[1] dianggap sudah terurut }
  for i ← 2 to n do  { mulai dari langkah 2 sampai langkah n }

     y ← L[i]
    { cari posisi yang tepat untuk y di dalam L[1..i-1] sambil menggeser }
```

```
    j ← i-1
    ketemu←false
    while (j ≥ 1) and (not ketemu) do
      if y > L[j] then
         L[j+1] ← L[j]        { geser }


         j ← j - 1
      else
         ketemu←true
      endif
    endwhile
    { J < 1 or ketemu }
    L[j+1] ← y   { sisipkan y pada tempat yang sesuai }

  endfor
```

Algoritma 16.13 Pengurutan sisip (menurun)

Komentar Mengenai Algoritma Pengurutan Sisip

Kelemahan algoritma pengurutan sisip terletak pada banyaknya operasi pergeseran yang diperlukan dalam mencari posisi yang tepat untuk elemen larik. Pada setiap *pass i*, operasi pergeseran yang diperlukan maksimum $i - 1$ kali. Untuk larik yang berukuran besar, jumlah operasi pergeseran meningkat secara kuadratik, sehingga pengurutan sisip kurang bagus untuk volume data yang besar.

16.6 Algoritma Pengurutan *Shell*

Algoritma pengurutan *Shell* diberi nama seuai nama penemunya (Donald Shell tahun 1959) [PAR95]. Algoritma ini merupakan perbaikan terhadap metode pengurutan sisip. Kelemahan metode pengurutan sisip sudah disebutkan pada bagian sebelum ini di atas. Jika data pada posisi ke-1000 ternyata posisi yang tepat adalah sebagai elemen kedua, maka dibutuhkan kira-kira 998 kali pergeseran elemen.

Untuk mengurangi pergeseran terlalu jauh, kita mengurutkan larik setiap k elemen dengan metode pengurutan sisip, misalkan kita urutkan setiap 5 elemen (k kita namakan juga *step* atau *increment*). Selanjutnya, kita gunakan nilai *step* yang lebih kecil, misalnya $k = 3$, lalu kita urut setiap 3 elemen. Begitu seterusnya sampai nilai $k = 1$. Karena nilai *step* selalu berkurang maka algoritma pengurutan *Shell* kadang-kadang dinamakan juga algoritma pengurutan kenaikan yang berkurang (*diminishing increment sort*)

Untuk memperjelas metode pengurutan *Shell*, tinjau pengurutan data *integer* berikut:

Data sebelum pengurutan:

81	94	11	96	12	35	17	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

***Pass* 1 (*step* = 5): Urutkan setiap lima elemen**

81	94	11	96	12	35	17	95	28	58	41	75	15
1	2	3	4	5	6	7	8	9	10	11	12	13

35 ..... 41 81
17 ....75 94
11 ....15..... ..95
28 96

Baris pertama menyatakan elemen yang sudah terurut pada posisi 1, 6, dan 11. Baris kedua menyatakan elemen yang terurut pada posisi 2, 7, dan 12 Baris ketiga menyatakan elemen yang sudah terurut pada posisi 3, 9, dan 13 Baris keempat menyatakan elemen yang sudah terurut pada posisi 4 dan 9. Hasil *pass* pertama: 35, 17, 11, 28, 12, 41, 75, 15, 96, 58, 81, 94, 95

***Pass* 1 (*step* = 3): Urutkan setiap tiga elemen**

35	17	11	28	12	41	75	15	96	58	81	94	95
1	2	3	4	5	6	7	8	9	10	11	12	13

28 ..... 35 ..... 58..... ..75 95
12 ..... 15 ....17..... ..81
11 ..... 41..... 94..... ..96

Hasil *pass* kedua: 28, 12, 11, 35, 15, 41, 58, 17, 94, 75, 81, 96, 95

***Pass* 3 (*step* = 1): Urutkan setiap satu elemen**

35	17	11	28	12	41	75	15	96	58	81	94	95
1	2	3	4	5	6	7	8	9	10	11	12	13
11	12	15	17	28	35	41	58	75	81	94	95	96

Hasil *pass* ketiga: 11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96

Perhatikanlah bahwa pada *pass* yang terakhir (*step* = 1), pengurutan *Shell* menjadi sama dengan pengurutan sisip biasa.

Nilai-nilai *step* seperti 5, 3, dan 1 bukanlah angka "sihir" (*magic*). Kita dapat memilih nilai-nilai *step* yang lain yang bukan kelipatan dari *step* yang lain. Pemilihan *step* yang merupakan perpangkatan dari dua (seperti 8, 4, 2, 1) dapat mengakibatkan perbandingan elemen yang sama pada suatu *pass*

akan terulang kembali pada *pass* berikutnya. Meskipun beberapa penelitian telah dibuat pada algoritma *Shell*, namun tidak seorang pun yang dapat membuktikan bahwa pemilihan *step* tertentu paling bagus di antara pemilihan *step* yang lain [KRU91].

Secara garis besar, algoritma pengurutan *Shell* dituliskan sebagai berikut:

1. *step* ← *n* { *n* = ukuran larik }
2. While *step* > 1 do
   a. *step* ← *step* div 3 + 1
   b. For *i* ← 1 to *step* do
      Insertion Sort setiap elemen ke-*step* mulai dari elemen ke-*i*

Algoritma pengurutan *Shell* dibuat dengan pertama-tama memodifikasi algoritma pengurutan sisip sedemikian sehingga kita dapat menspesifikasikan titik awal pengurutan dan ukuran *step* (pada algoritma pengurutan sisip yang asli, titik awal pengurutan adalah elemen pertama dan ukuran *step* = 1). Modifikasi algoritma pengurutan sisip ini kita beri nama *InsSort*. Algoritma *InsSort* dan *ShellSort* selengkapnya kita tulis berikut ini.

```
procedure InsSort(input/output L : LarikInt, input n, start,
                  step : integer)
{Mengurutkan elemen larik L[start..n] sehingga tersusun menaik dengan
metode pengurutan sisip yang dimodifikasi untuk Shell Sort. }
{ K.Awal : Elemen-elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir:Elemen-elemen larik pada kenaikan sebesar step terurut menaik}

DEKLARASI
I : integer       { pencacah step }
J : integer       { pencacah untuk penelusuran larik }
y : integer       { peubah bantu yang menyimpan nilai L[K] }
   ketemu : boolean { untuk menyatakan posisi penyisipan ditemukan }

ALGORITMA:
  { elemen L[start] dianggap sudah terurut }
  i ← start + step
  while i ≤ n do
      y ← L[i]
      { sisipkan L[i] ke dalam bagian yang sudah terurut }
      { cari posisi yang tepat untuk y di dalam L[start..i-1] sambil
        menggeser}
     j ← i - step
     ketemu←false

     while (j ≥ 1) and (not ketemu) do
       if y < L[j] then
          L[j + step] ← L[j]   geser }
          j ← j - step
       else
          ketemu ← true
       endif
     endwhile
     { j < 1 or ketemu }
```

```
    L[j + step] ← y  { sisipkan y pada tempat yang sesuai }
    i ← i + step
  endwhile
```

Algoritma 16.14 Modifikasi pengurutan sisip untuk pengurutan *Shell*

Perhatikan Algoritma 16.14 di atas, jika *start* = 1 dan *step* = 1, maka algoritma `InsSort` menjadi sama dengan algoritma pengurutan sisip biasa.

```
procedure ShellSort(input/output L : LarikInt, input n : integer)
{Mengurutkan elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan Shell. }
{ K.Awal :  Elemen-elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir: Elemen-elemen larik L terurut menaik }

DEKLARASI
   step, start : integer

DESKRIPSI
  step ← n
  while step > 1 do
     step ← step div 3 + 1
     for start ← 1 to step do
        InsSort(L, n, start, step)
     endfor
  endwhile
```

Algoritma 16.15 Pengurutan *Shell*

Komentar Mengenai Algoritma Pengurutan *Shell*

Sebagaimana sudah dijelaskan sebelumnya, algoritma pengurutan *Shell* merupakan perbaikan terhadap metode pengurutan sisip. Namun, tidak seorang pun yang pernah dapat menganalisis algoritma pengurutan *Shell* secara tepat, karena pemilihan ukuran *step* (seperti 5, 3, 1, atau pendefinisian ukuran *step* dengan pernyataan `step ← step div 3 + 1`) didasarkan pada pertimbangan sugesti. Tidak ada aturan yang diketahui untuk menemukan ukuran *step* yang optimum. Jika *step* dipilih yang berdekatan, semain banyak jumlah *pass* yang dihasilkan, tapi setiap *pass* mungkin lebih cepat. Jika ukuran *step* menurun dengan cepat, jumlah *pass* akan berkurang tetapi setiap *pass* menjadi lebih panjang (lama). Namun yang pasti, ukuran *step* terakhir adalah 1, sehingga pada akhir proses, larik diurutkan dengan pengurutan sisip biasa.

16.7 Penggabungan Dua Buah Larik Terurut

Misalkan kita memiliki dua buah larik, L_1 dan L_2, yang masing-masing sudah terurut menaik. Kita ingin membentuk sebuah larik baru, L_3, yang

merupakan gabungan dari dua buah larik tersebut sedemikian sehingga $L3$ juga terurut menaik.

Misalkan elemen-elemen larik $L1$ dan $L2$ masing-masing adalah

Larik $L1$:

1	13	24

Larik $L2$:

2	15	27	30

Penggabungan $L1$ dan $L2$ menghasilkan $L3$ yang tetap terurut menaik:

Larik $L3$:

1	2	13	15	24	27	30

Proses penggabungan dikerjakan dengan cara membandingkan satu elemen pada larik $L1$ dengan satu elemen pada larik $L16$. Jika elemen pada $L1$ lebih kecil dari elemen pada $L2$, maka salin elemen dari $L1$ ke $L3$. Elemen berikutnya pada $L1$ maju satu elemen, sedangkan elemen $L2$ tetap. Hal yang sama juga berlaku bila elemen dari $L2$ lebih kecil dari elemen $L1$, maka salin elemen dari $L2$ ke $L3$. Larik $L2$ maju satu elemen, larik $L1$ tetap. Dengan cara seperti ini, akan ada larik yang elemennya sudah duluan habis disalin, sedangkan larik yang lain masih tersisa. Salin seluruh elemen yang tersisa ke $L3$.

Hal lain yang harus diperhatikan adalah ukuran larik $L3$. Jumlah elemen larik $L3$ adalah banyaknya elemen larik $L1$ ditambah dengan banyaknya elemen larik $L16$. Jumlah elemen larik $L3$ ini harus lebih kecil dari jumlah maksimum elemen larik yang disediakan (*Nmaks*).

Penggabungan $L1$ dan $L2$ pada Contoh 16.9 dilaksanakan sebagai berikut:

$L1$	$L2$		$L3$
1 13 24	**2** 15 27 30	1 < 2 → 1	1
1 **13** 24	**2** 15 27 30	2 < 13 → 2	1 2
1 **13** 24	2 **15** 27 30	13 < 15→13	1 2 13
1 13 **24**	2 **15** 27 30	15 < 24→15	1 2 13 15
1 13 **24**	2 15 **27** 30	24 < 27→ 24	1 2 13 15 24
1 13 24	2 15 **27** 30	27 →	1 2 13 15 24 27
1 13 24	2 15 27 **30**	30 →	1 2 13 15 24 27 30

Algoritma penggabungan dua buah larik terurut selengkapnya sebagai berikut:

```
procedure  TableMerge (input L1 : LarikInt, input n1 : integer,
                       input L2 : LarikInt, input n2 : integer,
                       output L3 : LarikInt, output n3 : integer)
{ Penggabungan dua buah larik terurut, L1 dan L2, menghasilkan larik
baru L3 yang terurut menaik. }
{ K.Awal:Larik L1[1..n1] sudah terdefinisi elemen-elemennya dan
terurut menaik. Larik L2[1..n2] sudah terdefinisi elemen-elemennya dan
terurut menaik. }
{K.Akhir: Larik L3[1..n3] berisi hasil penggabungan larik L1 dan larik
L2 dan elemen-elemennya terurut menaik. n3 = n1 + n2 adalah ukuran
larik L3. }

DEKLARASI
    k1, k2, k3 : integer

ALGORITMA:
   n3 ← n1 + n2          { ukuran larik L3 }
   k1 ← 1
   k2 ← 1
   k3 ← 1
   while (k1 ≤ n1) and (k2 ≤ n2) do
      if L1[k1] ≤ L2[k2] then
        L3[k3] ← L1[k1]
        k1 ← k1 + 1    { L1 maju satu elemen}
      else
        L3[k3] ← L2[k2]
        k2 ← k2 + 1    { L2 maju satu elemen}
      endif

      k3 ← k3 + 1
   endwhile
   { k1 > n1 or k2 > n2 }

   { salin sisa L1, jika ada }
   while (k1 ≤ n1) do
      L3[k3] ← L1[k1]
      k1 ← k1 + 1
      k3 ← k3 + 1
   endwhile
   { k1 > n1 }

   { salin sisa L2, jika ada }
   while (k2 ≤ n2) do
     L3[k3] ← L2[k2]
     k2 ← k2 + 1
     k3 ← k3 + 1
  endwhile
  { k2 > n2 }
```

Algoritma 16.16 Penggabungan dua buah larik terurut

16.8 Pengurutan pada Larik Terstruktur

Algoritma pengurutan yang dibahas sebelum ini menggunakan larik dengan elemen-elemen bertipe sederhana. Pada sebagian besar kasus, elemen larik sering bertipe terstruktur. Contoh, misalkan *TabMhs* adalah sebuah larik yang elemennya menyatakan nilai ujian seorang mahasiswa untuk suatu mata kuliah (*MK*) yang ia ambil. Data setiap mahasiswa adalah *NIM* (Nomor Induk Mahasiswa), nama mahasiswa, mata kuliah yang ia ambil, dan nilai mata kuliah tersebut. Deklarasi nama dan tipe sebagai berikut:

```
DEKLARASI
  const Nmaks = 100
  type Mahasiswa : record <NIM   : integer,   { Nomor Induk Mahasiswa }
                           NamaMhs : string,  { nama mahasiswa }
                           KodeMK  : string,  { kode mata kuliah }
                           Nilai: char    { indeks nilai MK (A/B/C/D/E) }
                          >
  type TabMhs : array[1..Nmaks] of Mahasiswa
  M : TabMhs
```

Algoritma 16.17 Deklarasi larik terstuktur

Struktur lojik larik *M* ditunjukkan di dalam Bab 1. Pada proses *entry* data mahasiswa, data dimasukkan secara acak (tidak terurut berdasarkan NIM). Kita dapat mengurutkan sekumpulan data mahasiswa tersebut dengan algoritma pengurutan yang sudah dibicarakan di atas.

Prosedur berikut menggunakan metode pengurutan pilih maksimum untuk mengurutkan sekumpulan data mahasiswa berdasarkan kunci *NIM*.

```
procedure SortDataMhs(input/output M : TabMhs, input n : integer)
{ Mengurutkan elemen larik M[1..N] sehingga tersusun menaik dengan
metode pengurutan pilih maksimum. }
{ K.Awal : Elemen-elemen larik M sudah terdefinisi. }
{ K.Akhir: Elemen larik M terurut menaik sedemikian sehingga
           M[1] ≤ M[2] ≤ ... ≤ M[n] }

DEKLARASI
    i     : integer    { pencacah pass }
    j     : integer    { pencacah untuk mencari nilai maksimum }
    imaks : integer    { indeks yang berisi nilai maksimum sementara }

    procedure Tukar (input/output a : Mahasiswa,
                     input/output b : Mahasiswa)
    { Mempertukarkan elemen a dan b }

ALGORITMA:
  for i ← n downto 2 do     { jumlah pass sebanyak n - 1 }
    { cari NIM terbesar pada elemen M[1..i] }
    imaks ← 1  {elemen pertama diasumsikan sebagai elemen terbesar
               sementara }
    for j ← 2 to i do
     if M[j].NIM > M[imaks].NIM then
        imaks ← j
     endif
```

```
    endfor
    {pertukarkan M[imaks] dengan M[i]}
    Tukar(M[imaks], M[i])
endfor
```

Algoritma 16.18 Pengurutan larik terstruktur

Prosedur `Tukar`:

```
procedure Tukar (input/output a : Mahasiswa,
                 input/output b : Mahasiswa)
{ Mempertukarkan nilai a dan b
{ K.Awal : a dan b sudah terdefinisi nilai-nilainya.
{ K.Akhir: b berisi nilai a sebelum pertukaran, dan b berisi nilai a
  sebelum pertukaran. }

DEKLARASI
   temp : Mahasiswa        { peubah bantu untuk pertukaran }

ALGORITMA:
   temp ← a
   a ← b
   b ← temp
```

Algoritma 16.19 Prosedur Tukar

16.9 Algoritma Pengurutan dalam Bahasa PASCAL dan Bahasa C

Di bawah ini kita sajikan algoritma pengurutan yang telah dikemukakan di atas dalam kedua bahasa tersebut.

PASCAL

```
(* DEKLARASI nama *)
 const Nmaks = 1000;      {jumlah elemen larik }
 type LarikInt = array [1..Nmaks] of integer;
```

1. Algoritma Pengurutan Apung

```
procedure BubbleSort1(var L : LarikInt; n : integer)
{Mengurutkan larik L[1..n] sehingga terurut menaik dengan algoritma
pengurutan gelembung. }
{ K.Awal : Elemen larik L sudah terdefinisi. }
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
           L[1] <= L[2] < … <= L[n] }

var
   i    : integer;  { pencacah untuk jumlah langkah }
   k    : integer;  { pencacah,untuk pengapungan pada setiap langkah }
   temp : integer;  { peubah bantu untuk pertukaran }
```

```
begin
   for i := 1 to n - 1 do
      for k := n downto i + 1 do
         if L[k] < L[k-1] then
           begin
             { pertukarkan L[k] dengan L[k-1] }
             temp := L[k];
             L[k] := L[k-1];
             L[k-1] := temp;
           end;{if}
      { endfor }
  {endfor }
end;
```

2. Algoritma Pengurutan Seleksi-Maksimum

```
procedure SelectionSort1(var L : LarikInt; n : integer);
{ Mengurutkan elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan seleksi-maksimum. }
{ K.Awal : Elemen larik L sudah terdefinisi harganya.
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
           L[1] <= L[2] <= ... <= L[n] }

var
i     : integer;       { pencacah untuk jumlah langkah }
j     : integer;       { pencacah untuk mencari nilai maksimum }
imaks : integer;    { indeks yang berisi nilai maksimum sementara }
temp  : integer;       { peubah bantu untuk pertukaran }

begin
for i := n downto 2 do  { jumlah pass sebanyak n - 1 }
 begin
   { cari elemen maksimum pada elemen L[1..i] }
   imaks := 1;  { elemen pertama diasumsikan sebagai elemen maksimum
                  sementara }
   for j := 2 to i do
     if L[j] > L[Imaks] then
        Imaks := j;
     {endif}
   {endfor}

   {pertukarkan L[imaks] dengan L[i]}
   temp := L[imaks];
   L[imaks] := L[i];
   L[i] := temp;

 end;{for}
end;
```

3. Algoritma Pengurutan Seleksi-Minimum

```
procedure SelectionSort3(var L : LarikInt; n : integer);
{ Mengurutkan elemen larik L[1..N] sehingga tersusun menaik dengan
metode pengurutan seleksi-minimum. }
{ K.Awal : Elemen larik L sudah terdefinisi nilainya.
{ K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
           L[1] ≤ L[2] ≤ ... ≤ L[n] }
```

```
var
i    : integer;          { pencacah pass }
j    : integer;          { pencacah untuk mencari nilai maksimum }
imin : integer;          { indeks yang berisi nilai maksimum sementara }
temp : integer;          { peubah bantu untuk pertukaran }

begin
for i := 1 to n - 1 do
 begin
   { cari indeks dari elemen minimum di dalam larik L[I..n] }
   imin := i;
   for j := i + 1 to n do
     if L[j] < L[imin] then
        Imin := j;
     {endif}
   {endfor}

   {pertukarkan L[imin] dengan L[i]}
   temp := L[imin];
   L[imin] := L[i];
   L[i] := temp;

 end;
end;
```

4. Algoritma Pengurutan Sisip

```
procedure InsertionSort1(var L : LarikInt; n : integer);
{ Mengurutkan elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan sisip.
{ K.Awal : Elemen-elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir: Elemen-elemen larik L terurut menaik sedemikian sehingga
           L[1] ≤ L[2] ≤ ... ≤ L[n]} }

var
i : integer;     { pencacah pass }
j : integer;     { pencacah untuk penelusuran larik }
y : integer;     { peubah bantu agar L[K] tidak ditimpa selama pergeseran }
   ketemu : boolean;  { peubah Boolean untuk menyatakan posisi
                        penyisipan ditemukan }

begin
  { elemen L[1] dianggap sudah terurut }
  for i := 2 to n do  { mulai dari langkah 2 sampai langkah n }
    begin
     { sisipkan L[i] ke dalam bagian yang sudah terurut }
      y := L[i];
     { cari posisi yang tepat untuk y di dalam L[1..i-1] sambil menggeser}
     j := i - 1;
     ketemu:=false;
     while (j >= 1) and (not ketemu) do
      begin
        if y < L[j] then
         begin
           L[j+1] := L[j];
           j := j - 1;
         end
        else
          ketemu:=true;
```

```
      end;{while}
      { j < 1 or ketemu }
      L[j+1] := y;  { sisipkan y pada tempat yang sesuai }

  end; {for}
end;
```

5. Algoritma Pengurutan *Shell*

```
procedure ShellSort(var L : LarikInt; n : integer);
{ Mengurutkan elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan Shell. }
{ K.Awal :  Elemen-elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir: Elemen-elemen larik L terurut menaik sedemikian sehingga
           L[1] ≤ L[2] ≤ ... ≤ L[n] }
var
   step, start : integer;

begin
  step := n;
  while step > 1 do
  begin
     step := step div 3 + 1;
     for start := 1 to step do
        InsSort(L, N, start, step);
     {endfor}
   end; {while }
end;
```

```
procedure InsSort(var L : LarikInt; n, start, step : integer);
{ Mengurutkan elemen larik L[start..n] sehingga tersusun menaik dengan
metode pengurutan sisip yang dimodifikasi untuk Shell Sort. }
{ K.Awal : Elemen-elemen larik L sudah terdefinisi nilainya. }
{ K.Akhir:Elemen-elemen larik pada kenaikan sebesar step terurut
menaik}

var
i : integer;        { pencacah step }
j : integer;        { pencacah untuk penelusuran larik }
y : integer;        { peubah bantu yang menyimpan nilai L[i]}
ketemu : boolean;   { peubah Boolean untuk menyatakan posisi penyisipan
                      ditemukan }

begin
  { elemen L[start] dianggap sudah terurut }
  i := start + step;
  while i <= n do
   begin
      y := L[i];  { sisipkan L[i] ke dalam bagian yang sudah terurut }

     {cari posisi yang tepat untuk y di dalam L[start..I-1] sambil
       menggeser}
     j := i - step;
     ketemu := false;
     while (j >= 1) and (not ketemu) do
      begin
        if y < L[j] then
```

```
        begin
          L[j + step] := L[j];          { geser }
          j := j - step
        end
       else
         ketemu:=true;
      {endif}
     end;{while}
    { j < 1 or ketemu }

    L[j+step] := y;  { sisipkan y pada tempat yang sesuai }
    i := i + step;
  end; {while}
```

C

```
/* DEKLARASI nama */
#define Nmaks 1000;      /* jumlah elemen larik */
typedef int LarikInt[Nmaks+1];
```

1. Algoritma Pengurutan Apung

```
void BubbleSort1 (LarikInt L, int n)
/* Mengurutkan larik L[1..n] sehingga terurut menaik dengan metode
pengurutan apung. */
/* K.Awal : Elemen larik L sudah terdefinisi */
/* K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
            L[1] <= L[2] < … <= L[n] */

{
 int i;     /* pencacah untuk jumlah langkah */
 int k;     /* pencacah,untuk pengapungan pada setiap langkah */
 int temp; /* peubah bantu untuk pertukaran */

 for (i = 1; i <= n - 1; i++)
    for (k = n; k >= i+1; k--)
        if (L[k] < L[k-1])
          {
           /* pertukarkan L[k] dengan L[k-1] */
            temp = L[k];
            L[k] = L[k-1];
            L[k-1] = temp;
          }
    /* endfor */
 /* endfor */
}
```

2. Algoritma Pengurutan Seleksi-Maksimum

```
void SelectionSort1(LarikInt L, int n)
/* Mengurutkan elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan maksimum. */
/* K.Awal : Elemen larik L sudah terdefinisi harganya. */
/* K.Akhir: Elemen larik L terurut menaik sedemikian sehingga
             L[1] <= L[2] <= … <= L[n] */
{
```

```
int i;        /* pencacah untuk jumlah langkah */
int j;        /* pencacah untuk mencari nilai maksimum */
   int imaks;    /* indeks yang berisi nilai maksimum sementara */
   int temp;     /* peubah bantu untuk pertukaran */

for (i = n; i >= 2; i--)    /* jumlah pass sebanyak n - 1 */
 {
   /* cari elemen maksimum pada elemen L[1..i]  */
   imaks = 1;  /* elemen pertama diasumsikan sebagai elemen maksimum
sementara */
   for (j = 2; j >= i; j++)
     if (L[j] > L[imaks])
        imaks = j;
     /*endif*/
   /*endfor*/

   /*pertukarkan L[imaks] dengan L[i]*/

   temp = L[imaks];
   L[imaks]= L[i];
   L[i] = temp;

 }
}
```

3. Algoritma Pengurutan Seleksi-Minimum

```
void SelectionSort3(LarikInt L, int n)
/* Mengurutkan elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan pilih minimum. */
/* K.Awal  : Elemen larik L sudah terdefinisi nilainya. */
/* K.Akhir : Elemen larik L terurut menaik sedemikian sehingga
             L[1] ≤ L[2] ≤ ... ≤ L[n]  */

{  int i;          /* pencacah pass */
   int j;          /* pencacah untuk mencari nilai maksimum */
   int imin;       /* indeks yang berisi nilai maksimum sementara */
   int temp;       /* peubah bantu untuk pertukaran */

for (i = 1; i <= n-1; i++)
 {
   /* cari indeks dari elemen minimum di dalam larik L[I..n] */
   imin = i;
   for (j = i+1; j<= n; j++)
     if (L[j] < L[imin])
        imin = j;
     /*endif*/
   /*endfor*/

   /*pertukarkan L[imin] dengan L[i]*/
   temp = L[imin];
   L[imin] = L[i];
   L[i] = temp;
 }
}
```

4. Algoritma Pengurutan Sisip

```
void InsertionSort1(LarikInt L, int n)
/* Mengurutkan elemen larik L[1..n] sehingga tersusun menaik dengan
metode pengurutan sisip. */
/* K.Awal : Elemen-elemen larik L sudah terdefinisi nilainya. */
/* K.Akhir: Elemen-elemen larik L terurut menaik sedemikian sehingga
            L[1] ≤ L[2] ≤ … ≤ L[n] */
{
   typedef enum{true = 1, false = 0} boolean;  /* tipe boolean */

   int i;          /* pencacah pass */
   int j;          /* pencacah untuk penelusuran larik */
   int y;          /* peubah bantu agar L[i] tidak ditimpa selama
                      pergeseran */
   boolean ketemu; /* peubah Boolean untuk menyatakan posisi
                      penyisipan ditemukan */

  /* elemen L[1] dianggap sudah terurut */
  for (i = 2; i <= n; i++)  /* mulai dari langkah 2 sampai langkah n */
   {
     /* sisipkan L[i] ke dalam bagian yang sudah terurut */
      y = L[i];
     /* cari posisi yang tepat untuk x di dalam L[1..i-1] sambil
        menggeser */
     j = i - 1;
     ketemu = false;

     while (j >= 1) && (!ketemu)
      {
        if (y < L[j])
         {
           L[j+1] = L[j];
           j--;
         }
        else
          ketemu = true;
      }
     /* j < 1 or ketemu */
     L[j+1] = y;  /* sisipkan y pada tempat yang sesuai */

  }
}
```

5. Algoritma Pengurutan *Shell*

```
void ShellSort(LarikInt L, int n)
/* Mengurutkan elemen larik L[1..n] sehingga tersusun menaik
   dengan  metode pengurutan Shell. */
/* K.Awal :  Elemen-elemen larik L sudah terdefinisi nilainya. */
/* K.Akhir: Elemen-elemen larik L terurut menaik sedemikian sehingga
            L[1] ≤ L[2] ≤ … ≤ L[n] */

{
int step, start;

  step = n;
  while (step > 1)
   {
     step = step/3 + 1;
```

```
    for (start = 1; start <= step; start++)
       InsSort(L, n, start, step);
    /*endfor*/
  }/*while */
}
```

```
void InsSort(LarikInt L, int n, int start, int step)
/*Mengurutkan elemen larik L[start..n] sehingga tersusun menaik dengan
metode pengurutan sisip yang dimodifikasi untuk Shell Sort. */
/* K.Awal : Elemen-elemen larik L sudah terdefinisi nilainya. */
/* K.Akhir:Elemen-elemen larik pada kenaikan sebesar step
   terurut menaik*/
{
   typedef enum{true = 1, false = 0} boolean;  /* tipe boolean */
int i;             /* pencacah step */
int j;             /* pencacah untuk penelusuran larik */
int y;             /* peubah bantu yang menyimpan nilai L[K] */
boolean ketemu; /* peubah Boolean untuk menyatakan posisi penyisipan
ditemukan */

/* elemen L[start] dianggap sudah terurut */
  i = start + step;
  while (i <= n)
   {
     y = L[i];  /* sisipkan L[i] ke dalam bagian yang sudah terurut. */

    /* cari posisi yang tepat untuk y di dalam L[start..i-1] sambil
       menggeser. */
    j = i - step;
    ketemu = false;
    while (j >= 1 && !ketemu)
     {
       if (y < L[j])
        {
          L[J+step] = L[j];   /* geser */
          j = j - step;
        }
       else
         ketemu = true;
      /*endif*/
     } /*while*/
    /* j < 1 or ketemu */

    L[j + step] = y;  /* sisipkan y pada tempat yang sesuai */
    i = i + step;
  } /*while*/
}
```

Soal Latihan Bab 16

1. Lakukan pengurutan menaik untuk sekumpulan data *integer* berikut:

   54, 23, 12, 56, 78, 50, 12, 89, 10, 12

masing-masing dengan algoritma pengurutan apung, pengurutan seleksi - maksimum, pengurutan seleksi-minimum, pengurutan sisip, dan pengurutan *Shell* (khusus untuk *Shell*, gunakan *step* 5, 3, 1).

2. Berdasarkan contoh pengurutan pada soal 1 di atas, algoritma pengurutan mana yang termasuk stabil dan algoritma mana yang tidak stabil? (tandai posisi angka-angka 12 sebelum dan sesudah pengurutan. Untuk membedakan ketiga angka 12 tersebut, masing-masing ditandai menjadi 12', 12", dan 12"').

3. Tuliskan kembali algoritma pengurutan apung sedemikian sehingga elemen-elemen terurut "tumbuh" dari "kanan" ke "kiri" (atau dari "bawah" ke "atas").

4. Tuliskan kembali algoritma pengurutan *Shell* untuk memperoleh larik yang terurut menurun.

5. Modifikasilah algoritma pengurutan sisip untuk mengurutkan data yang di-*entry* dari papan ketik. Dalam hal ini, setiap kali data dibaca, data tersebut dicarikan posisinya yang tepat di dalam larik sehingga larik tetap terurut menaik.

6. Modifikasi algoritma pengurutan seleksi (baik maksimum maupun minumum) sedemikian sehingga jika elemen maksimum/minimum yang ditemukan sudah pada posisi yang seharusnya, maka tidak perlu dilakukan pertukaran.

7. Lakukan pengurutan menaik untuk sekumpulan data *integer* berikut dengan metode *Shell*. Gunakan (i) *step* 7, 3, dan (ii) *step* 8, 4, 2, 1:

   34, 56, 12, 8, 90, 32, 78, 65, 21, 87, 92, 8, 50, 23, 19, 68, 71, 69, 1, 10

8. Metode pengurutan dengan pencacahan (*count sort*) adalah sebagai berikut: data yang akan diurutkan bertipe *integer* dan harganya terletak di dalam selang *NilaiMin..NilaiMaks*. Seluruh data tersebut disimpan di dalam larik *L*[1..*n*]. Deklarasikan larik *count*[*NilaiMin..NilaiMaks*] dan isi *count*[*i*] dengan banyaknya data yang berharga *i*. Kemudian tempatkan kembali setiap data yang bernilai *i* ke dalam larik *L* sebanyak *count*[*i*]. *L* akan berisi data yang telah terurut menaik. Tulislah algoritma pengurutan *count sort* tersebut.

17

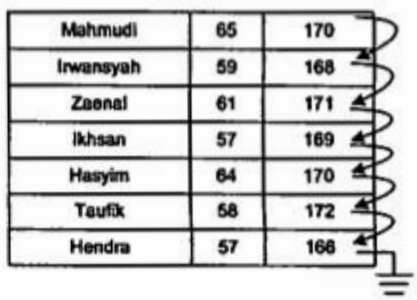
Pemrosesan Arsip Beruntun

Arsip (*file*) atau berkas adalah struktur penyimpanan data di dalam memori sekunder seperti *disk*. Data disimpan di dalam arsip agar sewaktu-waktu dapat dibuka kembali. Struktur arsip memungkinkan kita menyimpan data secara permanen dan mengaksesnya kembali bila perlu. Banyak aplikasi yang membutuhkan kemampuan ini.

Pada umumnya, arsip menyimpan informasi dari kategori yang sama. Misalnya, data karyawan disimpan di dalam arsip karyawan, data properti barang disimpan di dalam arsip properti, data keuangan disimpan di dalam arsip keuangan, dan sebagainya. Setiap arsip dikenali melalui namanya.

Setiap *item* data yang direkam di dalam arsip disebut **rekaman** (*record*). Semua rekaman di dalam arsip diorganisasikan penyimpanannya, dan pengaksesan rekaman di dalam arsip bergantung kepada metode pengorganisasiannya itu. Ada dua cara pengorganisasian data di dalam arsip: **beruntun** (*sequential*) dan **acak** (*random*) - yang terakhir ini juga dikenal sebagai arsip akses-langsung (*direct access*).

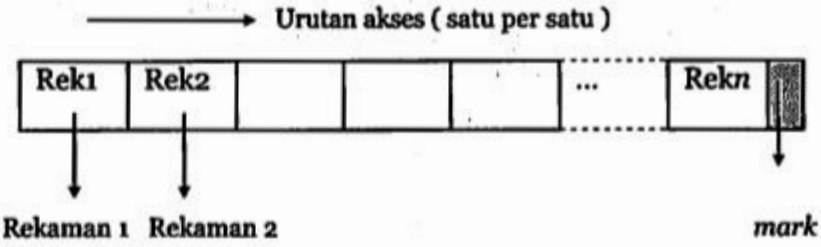
Arsip beruntun (*sequential file*) menyimpan rekaman secara berurutan, rekaman yang satu sesudah rekaman yang lain. Untuk mengakses data tertentu, program harus mengakses mulai dari rekaman pertama sampai rekaman yang diinginkan. Pengaksesan arsip jenis ini pada umumnya sangat lambat, khususnya bila arsip berukuran besar. Namun, arsip beruntun mudah dibuat dan dipelihara (*maintain*).



Pada arsip acak, rekaman diakses secara langsung, tanpa perlu memulai dari awal. Keuntungan jenis arsip ini, pengaksesan data individual lebih cepat daripada arsip beruntun, tetapi arsip acak lebih sulit dibuat dan dipelihara. Dari kedua cara pengorganisasian itu, pengorganisasian rekaman secara beruntun adalah metode yang paling sederhana.

17.1 Definisi Arsip Beruntun

Arsip beruntun adalah sekumpulan rekaman bertipe sama yang diakses secara berurutan mulai dari rekaman pertama sampai rekaman yang dituju atau sampai dengan rekaman yang terakhir. Rekaman di dalam arsip beruntun diakses satu per satu, yaitu rekaman demi rekaman, secara searah. Gambar 17.1 memperlihatkan gambaran lojik pengaksesan arsip beruntun. Setiap kotak pada gambar tersebut menyatakan sebuah rekaman. Kita dapat menganalogikan arsip beruntun seperti kumpulan lagu di dalam pita kaset. Untuk memainkan lagu pada urutan ke-6, kita harus memutar pita kaset mulai dari awal pita sampai awal lagu yang ke-6 ditemukan.



Gambar 17.1 Struktur arsip beruntun. Seperti halnya pada larik, kita dapat membayangkan struktur lojik arsip beruntun dengan sederetan kotak yang memanjang horizontal atau vertikal.

Perhatikanlah bahwa struktur arsip beruntun tidak jauh berbeda dengan struktur larik. Perbedaannya, elemen larik didefinisikan di dalam memori, sedangkan elemen arsip didefinisikan di dalam media penyimpanan sekunder. Skema pemrosesan pada larik pada hakikatnya sama dengan skema pemrosesan pada arsip. Anda dapat melihat bahwa beberapa algoritma traversal pada larik, seperti algoritma mencari nilai maksimum/minimum dan algoritma pencarian elemen tertentu, juga dapat digunakan pada arsip.

Perbedaan arsip dengan larik hanya pada pilihan cara pengaksesan elemen data dan "arah" pemrosesan. Rekaman di dalam arsip beruntun tidak dapat diakses secara langsung karena harus dibaca dari awal rekaman, sementara pada larik elemennya dapat diakses secara langsung melalui indeksnya. Jika pada arsip beruntun pemrosesan rekaman hanya dapat dilakukan dalam

satu arah, dimulai dari rekaman pertama sampai rekaman terakhir tetapi tidak dapat dari arah sebaliknya, maka pada larik kita dapat melakukan pemrosesan dari elemen terakhir menuju elemen pertama.

Jika pada larik kita dapat mengakhiri pemrosesan beruntun bilamana pencatat indeks larik sudah melebihi ukuran larik (biasa disimbolkan dengan *N*), maka pada arsip pembacaan rekaman berakhir jika sudah sampai pada tanda (*mark*) yang menandakan akhir arsip (*end of file*). Tanda akhir arsip biasanya karakter khusus yang bergantung pada bahasa pemrograman yang digunakan. Kita mengasumsikan bahwa fungsi pustaka yang dapat mendeteksi akhir arsip sudah tersedia sehingga kita hanya memfokuskan pada algoritma pemrosesan arsip saja. Pemrograman tidak perlu tahu bagaimana cara fungsi tersebut mendeteksi akhir arsip.

17.2 Pendeklarasian Arsip di dalam Algoritma

Sebelum melakukan pemrosesan arsip beruntun, arsip tersebut harus dideklarasikan terlebih dahulu. Cara mendeklarasikan arsip beruntun di dalam bagian DEKLARASI sebagai berikut (sebagai peubah):

```
DEKLARASI
    arsip : File of tipe rekaman
```

Algoritma 17.1 Deklarasi arsip beruntun sebagai peubah (variabel)

Tipe rekaman dapat berupa tipe dasar (*integer*, *real*, *char*, boolean, *string*) atau tipe terstruktur (*record*). Setiap rekaman di dalam arsip beruntun harus bertipe sama, baik dari tipe dasar maupun bertipe terstruktur. Rekaman bertipe terstruktur terdiri atas satu atau lebih *field* yang bertipe tertentu.

Adapun cara mendeklarasikan arsip beruntun sebagai tipe bentukan sebagai berikut:

```
DEKLARASI
   type nama tipe arsip : File of tipe rekaman     { tipe bentukan }

   arsip : nama tipe arsip
```

Algoritma 17.2 Deklarasi arsip beruntun sebagai tipe bentukan

Berikut ini contoh-contoh pendeklarasian arsip beruntun di dalam bagian DEKLARASI:

1. Arsip *Bil* yang berisi sekumpulan bilangan bulat. Setiap rekaman adalah sebuah bilangan bulat.

```
DEKLARASI
    Bil : File of integer
```

Algoritma 17.3 Deklarasi arsip integer

2. Arsip *Mhs* yang berisi data mahasiswa (*NIM*, *Nama*, dan *IP*). Keterangan: *NIM* = Nomor Induk Mahasiswa, *IP* = Indeks Prestasi (skala 0 – 4). Setiap rekaman di dalam arsip *Mhs* bertipe terstruktur (*record*).

```
DEKLARASI
    { tipe rekaman }
   type DataMhs : record <NIM : integer, Nama : string, IP : real>

   { arsip }
   Mhs : File of DataMhs
```

Algoritma 17.4 Deklarasi arsip data mahasiswa

3. Arsip *Kar*, yaitu arsip yang berisi data bertipe karakter. Setiap rekaman adalah satu karakter. Perhatikan bahwa arsip karakter tidak sama dengan teks (*text*).

```
DEKLARASI
   Kar : File of char
```

Algoritma 17.5 Deklarasi arsip karakter

4. Mendefinisikan arsip *integer* sebagai tipe bentukan.

```
DEKLARASI
   type ArsipInt : File of integer    { tipe bentukan }

   Bil : ArsipInt     { arsip bertipe ArsipInt }
```

Algoritma 17.6 Deklarasi arsip beruntun sebagai tipe bentukan.

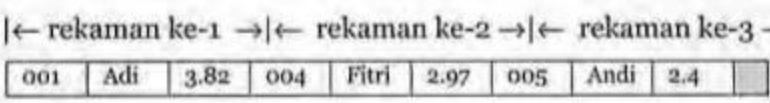
5. Mendefinisikan arsip data mahasiswa pada contoh 2 di atas sebagai tipe bentukan.

```
DEKLARASI
    { tipe rekaman }
   type DataMhs : record <NIM : integer, Nama : string, IP : real>

   { tipe arsip }
   type ArsipMhs : File of DataMhs

   Mhs : ArsipMhs     { arsip bertipe ArsipMhs }
```

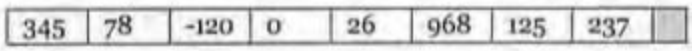
Algoritma 17.7 Deklarasi arsip data mahasiswa sebagai tipe bentukan.

Gambar 17.2 memperlihatkan contoh tiga buah arsip yang disebutkan di atas, berturut-turut (1) arsip data mahasiswa (*Mhs*), (2) arsip bilangan bulat (*Bil*), dan (3) arsip karakter (*Kar*). Garis-garis yang membatasi antara satu rekaman dengan rekaman lainnya hanyalah garis fiktif saja dan garis tersebut tidak terdapat dalam implementasi fisik arsip beruntun pada media penyimpanan sekunder. Rekaman yang diarsir menyatakan *mark*. Suatu arsip kosong adalah arsip yang hanya berisi *mark*.

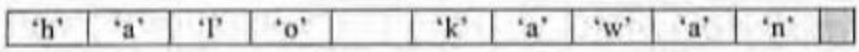
(1)



(2)



(3)



Gambar 17.2 Contoh tiga buah arsip: (1) Arsip data mahasiswa, (2) Arsip bilangan bulat, dan (3) Arsip karakter.

Contoh-contoh pendeklarasian arsip beruntun lainnya:

1. Mendeklarasikan arsip *Freal* sebagai arsip yang berisi data bilangan riil.

```
DEKLARASI
    Freal : File of real
```

Algoritma 17.8 Deklarasi arsip bilangan riil

2. Mendeklarasikan *ArsipTitik* yang berisi titik-titik (x, y) di bidang kartesian.

```
DEKLARASI
  type  Titik = record <x : real, y : real>

  ArsipTitik : File of Titik
```

Algoritma 17.9 Deklarasi arsip data titik di bidang kartesian

17.3 Fungsi Pustaka untuk Arsip Beruntun

Kita mendefinisikan sejumlah instruksi baku untuk pemrosesan arsip. Instruksi baku itu dinyatakan sebagai fungsi/prosedur pustaka (*library function*). Fungsi pustaka tersebut diasumsikan sudah tersedia sehingga kita dapat langsung memakainya. Kita hanya perlu mengetahui purwarupa (*prototype*) fungsi pustaka tersebut agar kita mengetahui cara memanggilnya (termasuk parameter yang dilewatkan).

Sejumlah instruksi baku yang dianggap sudah tersedia untuk pemrosesan arsip beruntun adalah:

1. ***Open***

Fungsi:

Membuka arsip beruntun untuk siap dibaca/ditulis. Setelah memanggil fungsi ini, penunjuk arsip (*file pointer*) akan menunjuk ke karakter pertama di awal arsip (mungkin karakter *mark*). Sebuah arsip beruntun yang dibuka hanya bisa untuk masukan (dibaca) saja atau sebagai keluaran (ditulis) saja, tidak bisa untuk dibaca dan ditulis sekaligus.

Purwarupa prosedur *Open*:

```
procedure Open (input nama arsip : tipe arsip, input kode : integer)
{ Membuka arsip beruntun untuk siap dibaca/ditulis. Kode = 1
artinya arsip dibuka untuk pembacaan, kode = 2 artinya arsip dibuka
untuk perekaman. }
{ K.Awal : Sembarang. }
{ K.Akhir: pointer pembacaan menunjuk ke awal rekaman }
```

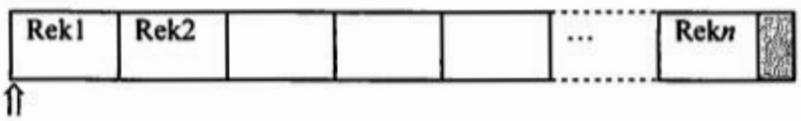
Algoritma 17.10 Purwarupa prosedur *Open*

Contoh-contoh membuka arsip:

```
Open(Mhs, 1)      { Arsip Mhs dibuka untuk dibaca }
Open(Bil, 2)      { Arsip Bil dibuka untuk ditulis}
Open(Kar, 1)      { Arsip Kar dibuka untuk dibaca }
```

Keterangan:

Di dalam implementasinya, prosedur `Open` berintekasi dengan sistem operasi (detailnya tidak perlu menjadi perhatian pemrogram) dan mengembalikan *pointer file* yang menunjuk ke awal rekaman (Gambar 17.3). Jika arsip kosong, maka *pointer* menunjuk ke tanda akhir arsip. Sekali arsip beruntun dibuka untuk dibaca (ditulis), maka arsip hanya dapat dibaca (ditulis) saja. Dengan kata lain, arsip yang dibuka untuk pembacaan tidak dapat digunakan untuk penulisan, begitu juga sebaliknya.



Gambar 17.3 Ketika sebuah arsip dibuka, sebuah *pointer* (digambarkan dengan anak panah) menunjuk ke awal arsip. Jika arsip dibuka untuk dibaca, *pointer* menunjuk ke awal rekaman pertama; jika arsip dibuka untuk ditulis, *pointer* menunjuk ke karakter akhir arsip (*mark*).

2. *Fread*

Fungsi:

Membaca rekaman yang sekarang sedang ditunjuk oleh *pointer*.
Purwarupa prosedur Fread:

```
procedure Fread (input nama arsip : tipe arsip,
                 output rekaman : tipe rekaman)
{ Membaca rekaman yang sekarang sedang ditunjuk oleh pointer baca
dari arsip nama arsip.
{K.Awal : nama arsip sudah dibuka dengan perintah Open . }
{K.Akhir : rekaman berisi nilai rekaman yang sedang ditunjuk oleh
pointer baca. Selanjutnya, setelah membaca satu rekaman, pointer
akan menunjuk ke awal rekaman berikutnya }
```

Algoritma 17.11 Purwarupa prosedur *Fread*

Contoh-contoh perintah membaca arsip:

```
Fread(Mhs, RekMhs)        { RekMhs bertipe DataMhs }
Fread(Bil, I)             { I bertipe integer }
Fread(Kar, ch)            { ch bertipe karakter }
```

Andaikan masing-masing *pointer* pembacaan sekarang sedang menunjuk ke awal rekaman ke-2 di dalam arsip pada Gambar 17.2, maka hasil ketiga perintah Fread di atas adalah:

RekMhs berisi <359604, 'Fitri', 2.97>
I berisi 78
ch berisi 'a'

Selanjutnya, *pointer* akan menunjuk ke awal rekaman ke-3 dan siap untuk membaca rekaman tersebut.

3. *Fwrite*

Fungsi:

Menulis rekaman ke dalam arsip beruntun.

Purwarupa prosedur *Fwrite*:

```
procedure Fwrite(input nama arsip : tipe arsip,
                 input rekaman : tipe rekaman)
{Menulis  rekaman ke dalam arsip nama arsip. }
{K.Awal : pointer penulisan sudah berada pada posisi siap merekam
{K.Akhir: rekaman tertulis ke dalam arsip nama arsip. Pointer
penulisan maju satu posisi ke depan }
```

Algoritma 17.12 Purwarupa prosedur *Fwrite*

Contoh-contoh perintah menulis ke arsip:

```
Fwrite(Mhs, <456091, 'Ida Bagus Adi Sudewa', 17.23>)
Fwrite(Bil, 765)
Fwrite(Kar, 'R')

{ isi data rekaman }
RekMhs.NIM ← 456087
RekMhs.Nama ← 'Hikmat Sadikin'
RekMhs.IP ← 2.76
Fwrite(Mhs, RekMhs)        { simpan rekaman tersebut ke arsip Mhs }
```

Keterangan:

`Fwrite(Bil, 765)` artinya menuliskan nilai 765 ke dalam arsip Bil.
`Fwrite(Mhs, RekMhs)` artinya menulis rekaman RekMhs ke dalam arsip Mhs.

4. *Close*

Fungsi:

Menutup arsip yang sudah dibuka.

Purwarupa prosedur *Close*:

```
procedure Close(input nama arsip : tipe arsip)
{ Menutup arsip yang telah dibuka
  K.Awal : Sembarang
  K.Akhir: nama arsip telah ditutup, tidak dapat diproses lagi
}
```

Algoritma 17.13 Purwarupa prosedur *Close*.

Contoh-contoh perintah menutup arsip:

```
Close(Mhs)
Close(Bil)
Close(Kar)
```

Selain keempat instruksi baku di atas, kita mendefinisikan sebuah fungsi *EOF* (*end of file*) yang berfungsi untuk mendeteksi akhir arsip. Fungsi *EOF* ini mengembalikan nilai *true* jika tanda akhir arsip dicapai, atau *false* jika tidak.

```
function EOF(input nama arsip : tipe arsip) → boolean
{ Mengembalikan nilai true jika pointer menunjuk ke tanda akhir arsip;
atau false jika tidak }
```

Algoritma 17.14 Purwarupa fungsi *Mark*

17.4 Membuat Arsip Beruntun

Arsip hanya dapat diproses jika sudah terdefinisi isinya. Langkah pertama dalam membuat arsip adalah menyiapkan arsip untuk perekaman (menggunakan perintah *Open* dengan Kode = 2). Langkah kedua adalah membaca data yang akan direkam (misalnya pembacaan dari piranti masukan), barulah kemudian menuliskan data tersebut ke dalam arsip (perintah *Fwrite*). Jika sebuah arsip sudah selesai diisi, arsip tersebut ditutup dengan perintah *Close*.

Contoh 17.1. Mengisi arsip bilangan bulat dengan nilai 1 sampai *n*. Nilai *n* dibaca dari piranti masukan. Tanda akhir arsip didefinisikan 9999.

```
PROGRAM BuatArsipBilanganBulat1
{ Contoh program yang memperagakan cara menyimpan data ke dalam arsip
beruntun }

DEKLARASI
   Bil : File of integer    { arsip bilangan bulat }
   n   : integer              { banyaknya bilangan bulat }
   i   : integer              { pencacah pengulangan }

ALGORITMA:
   Open(Bil, 2)    { buka arsip Bil untuk perekaman }
   read(n)
   for i ← 1 to n do
      Fwrite(Bil, i)
   endfor

   Close(Bil)        { tutup arsip }
```

Algoritma 17.15 Pengisian arsip integer dengan bilangan dari 1 sampai n.

Contoh 17.2. Mengisi arsip bilangan bulat dengan nilai yang dibaca dari papan ketik. Tanda akhir arsip adalah 9999.

```
PROGRAM BuatArsipBilanganBulat2
{       Contoh program yang memperagakan cara menyimpan data ke dalam
arsip, data dibaca dari papan ketik. }

DEKLARASI
   Fint : File of integer      { arsip bilangan bulat }
   n    : integer                { banyaknya bilangan bulat }
   i    : integer                { pencacah pengulangan }
   x    : integer              { peubah bilangan bulat yang dibaca dari
          piranti masukan }
ALGORITMA:
   Open(Fint, 2)          { buka arsip Fint untuk perekaman }
   read(n)
   for i ← 1 to n do
      read(x)
      Fwrite(Fint,x)
   endfor
   Close(Fint)            { tutup arsip }
```

Algoritma 17.16 Pengisian arsip *integer* dari piranti masukan

Contoh 17.3. Mengisi arsip *Mhs* dengan data mahasiswa yang dibaca dari papan ketik. Pembacaan data diakhiri bila *NIM* yang dimasukkan adalah 9999.

```
PROGRAM BuatArsipMahasiswa
{ Membuat arsip data mahasiswa. Data mahasiswa dibaca dari papan
ketik. }

DEKLARASI
   type DataMhs : record <NIM : integer, Nama : string, IP : real>
   Msiswa : DataMhs          { peubah untuk menampung pembacaan data
                               mahasiswa }
   Mhs : File of DataMhs

ALGORITMA:
   Open(Mhs, 2)              { buka arsip Mhs untuk perekaman }
   read(Msiswa.NIM)          { baca NIM mahasiswa pertama, mungkin 9999 }
   while(Msiswa.NIM ≠ 9999) do
      read(Msiswa.Nama, Msiswa.IP)
      Fwrite(Mhs, Msiswa)    { rekam Msiswa ke dalam arsip Mhs }
      read(Msiswa.NIM)
   endwhile
   { Msiswa.NIM = 9999}
   Close(Mhs)
```

Algoritma 17.17 Pengisian arsip data mahasiswa dari piranti masukan

17.5 Membaca Arsip Beruntun

Proses pembacaan arsip beruntun merupakan kebalikan dari proses pembuatan arsip. Langkah pertama dalam membaca arsip adalah menyiapkan arsip untuk pembacaan (menggunakan perintah *Open* dengan Kode = 1). Setiap kali akan membaca suatu rekaman di dalam arsip, kita harus memastikan bahwa tanda akhir arsip belum dicapai.

Rekaman dibaca satu per satu, dimulai dari rekaman pertama hingga rekaman yang diinginkan sudah ditemukan atau seluruh rekaman selesai dibaca. Karena kita tidak tahu jumlah data di dalam arsip, maka konstruksi pengulangan yang cocok adalah dengan *WHILE* seperti yang ditunjukkan pada Algoritma 17.18.

```
PROGRAM PembacaanArsip
{ Skema pembacaan arsip beruntun tanpa penanganan kasus kosong }

DEKLARASI
   type Rekaman : TipeRekaman
   Arsip : File of Rekaman
   Rek : Rekaman

ALGORITMA:
  Inisialisasi
```

```
Open(Arsip, 1)        { buka arsip untuk dibaca }

while not EOF(Arsip) do
   Fread(Arsip, Rek)
   Proses Rek
endwhile
{ EOF(Arsip) }

Terminasi
Close(Arsip)
```

Algoritma 17.18 Skema umum membaca arsip beruntun

Sebagai contoh, diberikan sebuah arsip *Bil* yang berisi data bilangan bulat (proses pengisian dapat dilihat pada Contoh 17.1). Semua data di dalam arsip *Bil* dibaca kemudian ditampilkan ke layar. Algoritmanya adalah:

```
PROGRAM BacaArsipBilanganBulat

{ Mencetak isi arsip ke layar peraga. Data dibaca dari arsip. }

DEKLARASI
     Bil : File of integer
   n   : integer                { banyaknya bilangan bulat }
   x   : integer                { peubah rekaman }

ALGORITMA:
   Open(Bil, 1)    { buka arsip Bil untuk pembacaan }
   while not EOF(Bil) do
      Fread(Bil, x)   { baca rekaman pertama }
      write(x)
   endwhile
   { EOF(Bil) }

   Close(Bil)   { tutup arsip }
```

Algoritma 17.19 Membaca data dari arsip integer dan mencetaknya ke layar

Di bawah ini kita berikan contoh-contoh algoritma lainnya yang di dalamnya ada proses pembacaan data dari arsip. Semua algoritma kita nyatakan sebagai prosedur saja agar lebih modular.

Contoh 17.4. Diberikan arsip *Mhs* (perhatikan bahwa arsip mungkin kosong). Algoritma di bawah ini adalah prosedur untuk menghitung jumlah mahasiswa yang lulus suatu tahap pendidikan (yaitu mahasiswa dengan *Indeks Prestasi atau* $P \geq 2.0$).

```
procedure HitungKelulusan(input Mhs : ArsipMhs,
                          output Ndiatas2 : integer)
{ Menghitung jumlah mahasiswa yang IP-nya di atas 2.0. Data mahasiswa
dibaca dari arsip Mhs.
  K.Awal : Arsip Mhs mungkin kosong.
  K.Akhir: Ndiatas2 berisi jumlah mahasiswa yang IP-nya di atas 2.0.
}
```

```
DEKLARASI
  RekMhs : DataMhs

ALGORITMA:
  Ndiatas2 ← 0
  Open(Mhs, 1)    { buka arsip Mhs untuk pembacaan }
  while not EOF(Mhs) do
    Fread(Mhs, RekMhs)    { baca data }

    if RekMhs.IP > 2.0 then
       Ndiatas2 ← Ndiatas2 + 1
    endif

  endwhile
  { EOF(Mhs) }
  Close(Mhs)    { tutup arsip }
```

Algoritma 17.20 Menghitung jumlah mahasisa yang mempunya IP di atas 2.0

Contoh 17.5. Diberikan arsip *Mhs* (perhatikan bahwa arsip mungkin kosong). Prosedur di bawah ini mencari data mahasiswa dengan *NIM* tertentu. Jika data mahasiwa tersebut ada di dalam arsip, maka sebuah parameter *boolean* diisi nilai *true*; sebaliknya diisi nilai *false*.

```
procedure CariNIM(input Mhs : ArsipMhs, input Nomor : integer,
                  output ketemu : boolean)
{ Mencari data mahasiswa yang NIM-nya = Nomor. }
{ K.Awal : Arsip Mhs mungkin kosong, Nomor sudah berisi NIM mahasiswa
  yang dicari. }
{ K.Akhir: Jika mahasiswa yang NIM-nya = Nomor ditemukan, ketemu
berisi true, sebaliknya jika tidak ada maka ketemu berisi false }

DEKLARASI:
  RekMhs : DataMhs

ALGORITMA:
  ketemu ← false
  Open(Mhs, 1)    { buka arsip Mhs untuk pembacaan }
  while (not EOF(Mhs)) and (not ketemu) do

     Fread(Mhs, RekMhs)    { baca data mahasiswa }

     if RekMhs.NIM = Nomor then
        ketemu ← true
     endif

  endwhile
  { EOF(Mhs) or ketemu }

  Close(Mhs)    { tutup arsip }
```

Algoritma 17.21 Mencari NIM tertentu dari arsip data mahasiswa

Contoh 17.6. Diberikan arsip *Mhs* yang berisi rekaman data mahasiswa (arsip mungkin kosong). Buatlah sebuah prosedur yang menghitung berapa rata-rata *IP* seluruh mahasiswa. Perhatikan bahwa kita tidak mengetahui jumlah seluruh mahasiswa, tetapi karena setiap rekaman di dalam asrip merepresentasikan data setiap mahasiswa, maka jumlah mahasiswa sama dengan jumlah seluruh rekaman di dalam arsip. Jadi, ketika arsip dibaca, sekaligus kita cacah jumlah rekaman di dalamnya. Jika arsip kosong, maka nilai rata-ratanya tidak terdefinisi (diisi –9999).

```
procedure HitungRataRata(input Mhs : ArsipMhs, output U : real)
{ Menghitung rata-rata IP seluruh mahasiswa. Rata-rata IP dihitung
dengan rumus: U = Total IP/jumlah mahasiswa. }
{ K.Awal : Arsip Mhs mungkin kosong. }
{ K.Akhir: U berisi rata-rata IP seluruh mahasiswa.}

DEKLARASI
  RekMhs    : DataMhs
  TotalIP   : real
  Nmhs      : integer  { jumlah mahasiswa }

ALGORITMA:
  TotalIP ← 0
  Nmhs ← 0
  Open(Mhs, 1)    { buka arsip Mhs untuk pembacaan }
  while not EOF(Mhs)) do
      Fread(Mhs, RekMhs)    { baca rekaman  }
      Nmhs ← Nmhs + 1
      TotalIP ← TotalIP + RekMhs.IP
  endwhile
  { EOF(Mhs) }

  if Nmhs ≠ 0 then    { arsip tidak kosong }
     U ← TotalIP/Nmhs  { rata-rata IP seluruh maahsiswa }
  else  { arsip kosong }
     U ← -9999    { nilai rata-rata tidak terdefinisi  }
  endif

  Close(Mhs)   { tutup arsip }
```

Algoritma 17.22 Menghitung IP rata-rata dari seluruh mahasiswa

Contoh 17.7. Diberikan arsip *Mhs* yang berisi rekaman data mahasiswa (arsip mungkin kosong). Prosedur di bawah ini menemukan data mahasiswa yang mempunyai *IP* tertinggi.

```
procedure CariIPTertinggi(input Mhs : ArsipMhs, output M : DataMhs)
{ Mencari data mahasiswa yang mempunyai IP tertinggi.}
{ K.Awal : Arsip Mhs mungkin kosong. }
{ K.Akhir: M berisi data mahasiswa (NIM, Nama, IP) yang mempunyai IP
  tertinggi.}

DEKLARASI
  RekMhs : DataMhs
```

```
ALGORITMA:
  Fread(Mhs, RekMhs)    { baca rekaman pertama }
  Maks ← -9999          { IP tertinggi simum sementara }
  Open(Mhs, 1)    { buka arsip Mhs untuk pembacaan }
  while not EOF(Mhs) do
     if RekMhs.IP > Maks then
        Maks ←  RekMhs.IP
        M ← RekMhs
     endif
     Fread(Mhs, RekMhs)
  endwhile
  { Mark(RekMhs) }

   Close(Mhs)   { tutup arsip }
```

Algoritma 17.23 Mencari data mahasiswa dengan IP tertinggi

Bila kita ingin menampilkan data mahasiswa yang mempunyai IP tertinggi tersebut, maka panggil prosedur `CariIPTertinggi` di atas, lalu cetak keluarannya, seperti ditunjukkan oleh algoritma di bawah ini:

```
PROGRAM CetakDataMahasiswaTerbaik
{ Menampilkan data mahasiswa yang mempunyai IP tertinggi. Data mahasiswa
yang mempunyai IP tertinggi ditampilkan ke piranti keluaran. }

DEKLARASI
    type DataMhs : record
                   <NIM : integer,
                    Nama : string,
                    IP : real
                   >
  Msiswa : DataMhs
  Mhs : File of DataMhs

  M : DataMhs

  procedure CariIPTertinggi(input Mhs : ArsipMhs, output M : DataMhs)
  { Mencari data mahasiswa yang mempunyai IP tertinggi. }

ALGORITMA:
   CariIPTertinggi(Mhs, M)
   { cetak data mahasiswa yang mempunyai IP tertinggi }
   write('NIM      : ', M.NIM)
   write('Nama     : ', M.Nama)
   write('IP       : ', M.IP)
```

Algoritma 17.24 Program utama yang memanggil Algoritma 17.26

17.6 Contoh Kasus Pengelolaan Data Mahasiswa

Misalkan *Kuliah* adalah sebuah arsip beruntun yang berisi data nilai seluruh mata kuliah yang diambil mahasiswa pada satu semester. Satu orang mahasiwa dapat memiliki beberapa buah nilai karena dalam satu semester ia mengambil beberapa mata kuliah dan setiap mahasiswa mungkin tidak sama mata kuliah yang diambilnyanya. Rekaman dengan NIM yang sama disimpan berturutan. Setiap rekaman akan berisi data:

a. Nomor Induk Mahasiswa (NIM),
b. nama mahasiswa,
c. kode mata kuliah (MK) yang diambil,
d. bobot SKS (Satuan Kredit Semester) mata kuliah yang diambil,
e. indeks nilai mata kuliah yang bersangkutan berupa huruf: A, B, C, D, E.

Contoh arsip yang memuat rekaman tersebut misalnya:

NIM ↓	Nama ↓	Kode MK ↓	SKS ↓	Nilai ↓
1359601	'Ezra Ibrahim'	'TI451'	3	'A'
1359601	'Ezra Ibrahim'	'KU301'	2	'B'
1359601	'Ezra Ibrahim'	'IF321'	3	'B'
1359606	'Agus Purwoko'	'KU302'	2	'C'
1359606	'Agus Purwoko'	'IF400'	4	'B'
1359607	'Leyla Khodra'	'IF323'	3	'C'
1359607	'Leyla Khodra'	'IF321'	3	'A'
1359607	'Leyla Khodra'	'IF400'	4	'A'
1359607	'Leyla Khodra'	'IF401'	2	'B'
1359609	'Ahmad Baiquni'	'IF221'	3	'C'
1359610	'Arif Bramantoro'	'IF400'	4	'A'

Pada bagian DEKLARASI di dalam program utama, kita definisikan tipe arsip sebagai berikut:

```
DEKLARASI
  type DataMhs : record <NIM : integer, Nama : string,
                         KodeMK : string, SKS : integer, Indeks : char>
  type ArsipNilai : File of DataMhs

  Kul  : ArsipNilai     { nama arsip data mahasiswa }
```

Algoritma 17.25 Deklarasi tipe arsip data mahasiswa

1. Data mahasiswa dibaca dari papan ketik, lalu disimpan ke dalam arsip *Kuliah*. Jumlah data mahasiswa yang akan dimasukkan tidak ditentukan sebelumnya. Proses pembacaan data dari papan kunci akan berhenti jika kode *NIM* yang dibaca adalah 9999.

```
procedure  RekamDataMahasiswa(output Kul : ArsipNilai)
{ Merekam data ujian mahasiwa ke dalam arsip Kul, data di baca dari
papan ketik, proses perekaman berhenti jika NIM yang dimasukkan =
9999. }
{ K.Awal : sembarang. }
{ K.Akhir: arsip Kul berisi data ujian mahasiswa, mungkin kosong }

DEKLARASI
   KuliahMhs  : DataMhs

ALGORITMA:
   Open(Kul, 2)    { buka arsip Kul untuk penulisan }
   read(KuliahMhs.NIM)    { baca data NIM pertama, mungkin 9999 }
   while(KuliahMhs.NIM ≠ 9999) do
      read(KuliahMhs.Nama)
      read(KuliahMhs.KodeMK)
      read(KuliahMhs.SKS)
      read(KuliahMhs.Indeks)
      Fwrite(Kul, KuliahMhs) {rekam KuliahMhs ke dalam arsip Kul}
      read(KuliahMhs.NIM)
   endwhile
   { KuliahMhs.NIM = 9999}

   Close(Kul)
```

Algoritma 17.26 Merekam data mahasiswa yang dibaca dari piranti masukan

2. Membuat daftar nilai untuk mahasiswa dengan NIM tertentu dan menampilkannya ke layar peraga. Contoh keluarannya kira-kira sebagai berikut:

```
Daftar Nilai Mata Kuliah
NIM  : 1359601
Nama : Ezra Ibrahim
```

No	Mata Kuliah	SKS	Nilai
1	TI451	3	A
2	KU301	2	B
3	IF321	3	B

Langkah pertama adalah mencari data berdasarkan kode NIM yang dimasukkan. Setelah data mahasiswa ditemukan, maka data mahasiswa yang bersangkutan dicetak ke piranti keluaran. Karena data mahasiswa dengan NIM yang sama disimpan secara berurutan, maka pembacaan rekaman berlanjut sampai terbaca NIM yang tidak sama dengan NIM yang dicari.

```
procedure  CetakDaftarNilai(input Kul : ArsipNilai,
                            input KodeNIM : integer)
{ Mencetak daftar nilai untuk mahasiswa dengan NIM = KodeNIM. }
{ K.Awal : Arsip Kul mungkin kosong, KodeNIM sudah terdefinisi
  harganya. Arsip Kul sudah dibuka untuk pembacaan. }
{ K.Akhir: Daftar nilai mahasiswa dengan KodeNIM tercetak ke piranti
  keluaran. Jika arsip kosong, cetak pesan 'Arsip kosong' }
```

```
DEKLARASI
  KuliahMhs  : DataMhs
  no     : integer        { nomor urut kuliah }
  ketemu : boolean        { true jika KodeNIM ditemukan di dalam arsip }
  stop   : boolean        { true jika tidak ada lagi nilai mata kuliah
                            milik KodeNIM }

ALGORITMA:
  Open(Kul, 1)    { buka arsip Kul untuk pembacaan. }
  ketemu ← false

 { cari KodeNIM di dalam arsip Kuliah }
  while (not EOF(Kul) and (not ketemu) do
     Fread(Kul, KuliahMhs)
     if KuliahMhs.NIM = KodeNIM then
        ketemu ← true        { data kuliah KodeNIM ditemukan }
     endif
  endwhile
  {EOF(Kul) or ketemu }

  if ketemu then

        { cetak seluruh daftar nilai untuk KodeNIM }
        write('   Daftar Nilai Mata Kuliah      ')
        write('     NIM : ', KuliahMhs.NIM)
        write('     Nama: ', KuliahMhs.Nama)
        write('-----------------------------------')
        write(' No.  Mata Kuliah  SKS   Nilai   ')
        write('-----------------------------------')
        no ← 1
        write(no, KuliahMhs.KodeMK, KuliahMhs.SKS, KuliahMK.Indeks)
        while (KuliahMhs.NIM = KodeNIM) and (not EOF(Kul))
           Fread(Kul, KuliahMhs)
           no ← no + 1
           write(no, KuliahMhs.KodeMK, KuliahMhs.SKS, KuliahMK.Indeks)
      endwhile
      { KuliahMhs.NIM ≠ KodeNIM or EOF(Kul) }

      write('-----------------------------------')
  else
     write('data mahasiswa dengan NIM = ', KodeNIM, ' tidak ada.')
  endif

  Close (Kul)
```

Algoritma 17.27 Mencetak daftar nilai mahasiswa tertentu

3. Mencetak daftar nilai untuk setiap mahasiswa. Contoh keluarannya kira-kira sebagai berikut:

Daftar Nilai Mata Kuliah
NIM : 1359601
Nama : Ezra Ibrahim

No	Mata Kuliah	SKS	Nilai
1	TI451	3	A
2	KU301	2	B
3	IF321	3	B

Daftar Nilai Mata Kuliah
NIM : 1359606
Nama : Agus Purwoko

No	Mata Kuliah	SKS	Nilai
1	KU302	2	C
2	IF400	4	B

Daftar Nilai Mata Kuliah
NIM : 1359607
Nama : Leyla Khodra

No	Mata Kuliah	SKS	Nilai
1	IF323	3	C
2	IF321	3	A
3	IF400	4	
4	IF401	2	B

Daftar Nilai Mata Kuliah
NIM : 1359609
Nama : Ahmad Baiquni

No	Mata Kuliah	SKS	Nilai
1	IF221	3	C

Daftar Nilai Mata Kuliah
NIM : 1359610
Nama : Arif Bramantoro

No	Mata Kuliah	SKS	Nilai
1	IF400	4	A

Algoritmanya sebagai berikut:

```
procedure CetakDaftarNilaiSeluruhMahasiswa(input Kul : ArsipMhs)
{ Mencetak daftar nilai untuk tiap-tiap mahasiswa. }
{ K.Awal : Arsip Kul mungkin kosong. }
{ K.Akhir: Daftar nilai tiap mahasiswa tercetak ke piranti keluaran. }

DEKLARASI
  KuliahMhs  : DataMhs
  NIMsekarang : integer
```

```
  no : integer

ALGORITMA:
  Open(Kul, 1)    { buka arsip Kul untuk pembacaan. }
  while not EOF(Kul) do
    Fread(Kul, KuliahMhs)    { baca rekaman }

        write('  Daftar Nilai Mata Kuliah      ')
        write('   NIM : ', KuliahMhs.NIM)
        write('   Nama: ', KuliahMhs.Nama)
        write('--------------------------------')
        write(' No.  Mata Kuliah  SKS   Nilai  ')
        write('--------------------------------')

        no ← 1
        write(no, KuliahMhs.KodeMK, KuliahMhs.SKS, KuliahMhs.Indeks)
        NIMsekarang ← KuliahMhs.NIM
        While (KuliahMhs.NIM = NIMsekarang) and (not EOF(Kul))
          Fread(Kul, KuliahMhs)
          no ← no + 1
          write(no, KuliahMhs.KodeMK, KuliahMhs.SKS, KuliahMhs.Indeks)
        endwhile
        { KuliahMhs.NIM ≠ NIMsekarang) or EOF(Kul) }

        write('--------------------------------')
  endwhile

  Close (Kul)
```

Algoritma 17.28 Mencetak daftar nilai setiap mahasiswa

4. Menampilkan daftar mahasiswa yang mengambil mata kuliah tertentu tertentu saja beserta nilainya (berdasarkan kode mata kuliah yang diberikan). Contoh keluaran kira-kira sebagai berikut:

```
Daftar Peserta Mata Kuliah
 Kode mata kuliah : IF400
```

No.	NIM	Nama	Nilai
1	1359606	Agus Purwoko	B
2	1359607	Leyla Khodra	A
3	1359610	Arif Bramantoro	A

```
procedure  CetakDaftarPesertaMK(input Kul : ArsipNilai,
                                input MatKul : string)
{ Mencetak peserta mata kuliah MatKul. }
{ K.Awal :Arsip Kul mungkin kosong, MatKul sudah terdefinisi
  harganya. }
{ K.Akhir: Daftar peserta mata kuliah dengan kode MatKul tercetak ke
  piranti keluaran. }

DEKLARASI
  KuliahMhs  : DataMhs
  no : integer
```

```
ALGORITMA
  Open(Kul, 1)    { buka arsip Kul untuk pembacaan. }

    write('    Daftar Peserta Mata Kuliah     ')
    write('     Kode mata kuliah : ', MatKul)
    write('-----------------------------------')
    write(' No.  NIM    Nama          Nilai    ')
    write('-----------------------------------')
    no ← 0
    while not EOF(Kul) do
      Fread(Kul, KuliahMhs)  { baca rekaman }
      if KuliahMhs.KodeMK = MatKul then
        no ← no + 1
        write(no, KuliahMhs.NIM, KuliahMhs.Nama, KuliahMhs.Indeks)
      endif
    endwhile
    { EOF(Kul) }
    write('-----------------------------------')
  endif

  Close (Kul)
```

Algoritma 17.29 Mencetak daftar peserta kuliah tertentu

5. Menghitung nilai rata-rata (*NR*) setiap mahasiswa. Untuk setiap mahasiswa, dicetak nilai-rata-ratanya. Ekivalensi nilai indeks dalam angka: A = 4, B = 3, C = 2, D = 1, E = 0). Rumus untuk menghitung *NR* adalah

   NR = total nilai dalam angka / jumlah SKS

   Contoh keluaran kira-kira sebagai berikut:

Daftar Nilai Rata-rata Mahasiswa

No.	NIM	Nama	NR
1	1359601	Ezra Ibrahim	3.38
1	1359606	Agus Purwoko	2.67
2	1359607	Leyla Khodra	3.40
3	1359610	Arif Bramantoro	17.00

```
procedure CetakNilaiRataRataMahasiswa(input Kul : ArsipNilai)
{ Mencetak daftar nilai rata-rata. }
{ K.awal: arsip Kul mungkin kosong }
{ K.Akhir: Daftar nilai rata-rata tiap mahasiswa tercetak ke piranti
  keluaran. }

DEKLARASI
  KuliahMhs    : DataMhs
  NIMsekarang  : integer   { NIM yang sedang diproses }
  totalnilai   : integer   { total nilai untuk seorang mahasiswa }
  angka        : integer   { ekivalensi nilai indeks dengan angka }
  totalSKS     : integer   { jumlah total SKS seorang mahasiswa }
  no           : integer   { nomor urut }
  NR           : real
```

```
ALGORITMA:

  Open(Kul, 1)    { buka arsip Kul untuk pembacaan. }
  write('  Daftar Nilai Rata-rata Mahasiswa  ')
  write('-------------------------------------')
  write('   No.  NIM      Nama          NR    ')
  write('-------------------------------------')

  while not EOF(Kul) do
     no ← 0
     totalnilai ← 0
     totalSKS ← 0
     NIMsekarang ← KuliahMhs.NIM
     while (KuliahMhs.NIM = NIMsekarang) and (not EOF(Kul))

         Fread(Kul, KuliahMhs)      { baca rekaman
         case KuliahMhs.Indeks
            'A' : angka ← 4
            'B' : angka ← 3
            'C' : angka ← 2
            'D' : angka ← 2
            'E' : angka ← 1
         endcase

         totalnilai ← totalnilai + angka*KuliahMhs.SKS
         totalSKS ← totalSKS + KuliahMhs.SKS

         NR ← totalnilai/totalSKS
         write(no, KuliahMhs.NIM, KuliahMhs.Nama, NR)

     endwhile
     { KuliahMhs.NIM ≠ NIMsekarang) or EOF(Kul) }
     write('-------------------------------------')
     Close (Kul)

  endwhile
  { EOF(Kul) }
```

Algoritma 17.30 Menghitung NR setiap mahasiswa

17.7 Menyalin Arsip

Rekaman di dalam suatu arsip adakalanya perlu disalin (*copy*) ke arsip yang lain. Penyalinan dapat dilakukan terhadap seluruh rekaman atau hanya rekaman tertentu saja. Di bawah ini kita menuliskan dua buah contoh algoritma penyalinan arsip. Algoritma pertama menyalin seluruh rekaman dari arsip *Mhs1* ke arsip *Mhs2*, sedangkan algoritma kedua hanya menyalin data mahasiswa yang lulus pada suatu tahap pendidikan (misalnya seorang mahasiswa dikatakan lulus bila IP-nya di atas 2.0).

(a) Menyalin seluruh rekaman di dalam arsip *Mhs1* ke arsip baru *Mhs2*

```
procedure SalinArsip(input Mhs1 : ArsipMhs, output Mhs2 : ArsipMhs)
{ Menyalin seluruh isi arsip Mhs1 ke arsip Mhs2. }
{ K.Awal : arsip Mhs1 mungkin kosong }
{ K.Akhir: Mhs2 berisi salinan seluruh rekaman dari arsip Mhs1 }

DEKLARASI
  RekMhs    : DataMhs

ALGORITMA:
  Open(Mhs1, 1)    { buka arsip Mhs1 untuk pembacaan. }
  Open(Mhs2, 2)    { buka arsip Mhs2 untuk penulisan. }

  while (not EOF(Mhs1)) do
    Fread(Mhs1, RekMhs)    { baca rekaman dari arsip Mhs1 }
    Fwrite(Mhs2, RekMhs)   { salin RekMhs ke arsip Mhs2 }
  endwhile
  { EOF(Mhs1) }

  Close(Mhs1)
  Close(Mhs2)
```

Algoritma 17.31 Menyalin isi arsip Mhs1 ke dalam arsip Mhs2

(b) Menyalin data mahaiswa yang *IP*-nya di atas 2.0 dari arsip *Mhs1* ke arsip baru *Mhs2*

```
procedure SalinLulusSaja(input Mhs1 : ArsipMhs, output Mhs2 :
ArsipMhs)
{ Menyalin data mahasiswa yang IP-nya di atas 2.0. Data mahasiswa
dibaca dari arsip Mhs1. }
{ K.Awal : arsip Mhs1 mungkin kosong }
{ K.Akhir: Mhs2 berisi data mahasiswa yang lulus dengan IP > 2.0. }

DEKLARASI
  RekMhs : DataMhs

ALGORITMA:

  Open(Mhs1, 1)    { buka arsip Mhs1 untuk pembacaan. }
  Open(Mhs2, 2)    { buka arsip Mhs2 untuk penulisan. }

  while not EOF(Mhs1) do

    Fread(Mhs1, RekMhs)    { baca rekaman dari arsip Mhs1 }
    if RekMhs.IP > 2.0 then
       Fwrite(Mhs2, RekMhs)
    endif
  endwhile
  { EOF(Mhs1) }

  Close(Mhs1)
  Close(Mhs2)
```

Algoritma 17.32 Menyalin data mahasiswa yang IP-nya di atas 2.0

17.8 Menggabung Dua Buah Arsip

Penggabungan arsip (*merging*) adalah menyalin isi dua buah arsip ke dalam sebuah arsip baru. Arsip-arsip yang akan digabung mungkin belum terurut (berdasarkan *field* tertentu) atau mungkin sudah terurut.

Cara penggabungan yang paling sederhana adalah penggabungan yang dilakukan dengan cara penyambungan (*concatenation*), yaitu menyalin isi arsip yang kedua setelah isi arsip pertama. Cara ini tidak membuat arsip hasil penggabungan terurut. Jika arsip hasil proses penggabungan harus tetap terurut, penggabungan dilakukan dengan cara yang mirip seperti penggabungan dua larik terurut (upabab 2.7).

Penggabungan Dua Buah Arsip dengan Cara Penyambungan

Misalkan diberikan dua buah arsip data mahasiswa, *Mhs1* dan *Mhs2*. Arsip baru hasil penggabungannya adalah *Mhs3*.

Contoh:

Arsip *Mhs1*:

359601	'Adi Purwanto'	3.82
359604	'Fitria Susanti'	2.97
359605	'Dewi Ramadhani'	2.74
359606	'Andrie Yanto'	3.02
359608	'Cahyo Kumolo'	2.05
359610	'Anton Previanto'	3.99

Arsip *Mhs2*:

359603	'Irwanti Sukma'	2.81
359609	'Melati Rahmi'	3.01
359614	'Gus Hamid'	2.46

Arsip *Mhs3*:

359601	'Adi Purwanto'	3.82
359604	'Fitria Susanti'	2.97
359605	'Dewi Ramadhani'	2.74
359606	'Andrie Yanto'	3.02
359608	'Cahyo Kumolo'	2.05
359610	'Anton Previanto'	3.99
359603	'Irwanti Sukma'	2.81
359609	'Melati Rahmi'	3.01
359614	'Gus Hamid'	2.46

Pendeklarasian arsip-arsip:

```
DEKLARASI
   type DataMhs : record <NIM : integer, Nama : string, IP : real>
   type ArsipMhs : File of DataMhs
```

Algoritma 17.33 Deklarasi arsip data mahasiswa

Prosedur penggabungan arsip:

```
procedure  GabungArsip1(input Mhs1, Mhs2 : ArsipMhs,
                        output Mhs3 : ArsipMhs)
{ Menggabungkan dua buah arsip, Mhs1 dan Mhs2, menjadi sebuah arsip
baru, Mhs3, dengan cara menyambung isi arsip Mhs1 dengan isi arsip
Mhs2 ke dalam arsip Mhs3. }
{ K.Awal : Arsip Mhs1 atau Mhs2 mungkin kosong }
{ K.Akhir: arsip Mhs3 berisi hasil penggabungan Mhs1 dan Mhs2 }

DEKLARASI
  RekMhs : DataMhs

ALGORITMA:

  Open(Mhs1, 1)    { buka arsip Mhs1 untuk pembacaan. }
  Open(Mhs2, 1)    { buka arsip Mhs2 untuk pembacaan. }
  Open(Mhs3, 2)    { buka arsip Mhs3 untuk penulisan. }

  while (not EOF(Mhs1) do
    Fread(Mhs1, RekMhs)
    Fwrite(Mhs3, RekMhs)
  endwhile
  { EOF(Mhs1) }

   { Sambung dengan data dari Mhs2, letakkan setelah data dari Mhs1 }
   while (not EOF(Mhs2) do
      Fread(Mhs2, RekMhs)
      Fwrite(Mhs3, RekMhs)
   endwhile
   { EOF(Mhs2) }

   Close(Mhs1)
   Close(Mhs2)
   Close(Mhs3)
```

Algoritma 17.34 Penggabungan arsip dengan cara penyambungan

Penggabungan Dua Buah Arsip Terurut

Misalkan arsip *Mhs1* dan *Mhs2* sudah terurut menaik berdasarkan *field NIM*. Kita ingin arsip hasil penggabungan *Mhs1* dan *Mhs2* juga terurut menaik di dalam arsip *Mhs3*.

Contoh:

Arsip *Mhs1*:

359601	'Adi Purwanto'	3.82
359604	'Fitria Susanti'	2.97
359605	'Dewi Ramadhani'	2.74
359606	'Andrie Yanto'	3.02
359608	'Cahyo Kumolo'	2.05
359610	'Anton Previanto'	3.99

Arsip *Mhs2*:

359603	'Irwanti Sukma'	2.81
359609	'Melati Rahmi'	3.01
359614	'Gus Hamid'	2.46

Arsip *Mhs3*:

359601	'Adi Purwanto'	3.82
359603	'Irwanti Sukma'	2.81
359604	'Fitria Susanti'	2.97
359605	'Dewi Ramadhani'	2.74
359606	'Andrie Yanto'	3.02
359608	'Cahyo Kumolo'	2.05
359609	'Melati Rahmi'	3.01
359610	'Anton Previanto'	3.99
359614	'Gus Hamid'	2.46

Cara penggabungan dua buah arsip terurut menjadi sebuah arsip terurut sama seperti penggabungan dua buah larik terurut (lihat pembahasan Bab 2.7). Jika kedua buah arsip kosong, maka arsip hasil penggabungan juga kosong.

```
procedure  GabungArsip2(input Mhs1, Mhs2 : ArsipMhs,
                        output Mhs3 : ArsipMhs)

{ Menggabungkan dua buah arsip, Mhs1 dan Mhs2, yang sudah terurut
menjadi sebuah arsip baru, Mhs3, yang juga tetap terurut. }
{ K.Awal :   Arisp Mhs1 atau Mhs2 mungkin kosong. }

{ K.Akhir: arsip Mhs3 berisi hasil penggabungan Mhs1 dan Mhs2. Jika
kedua buah arsip kosong, maka arsip hasil penggabungan juga kosong }

DEKLARASI
  RekMhs1, RekMhs2 : DataMhs
  Akhir1, akhir2 : boolean  { peubah yang mengindikasikan akhir arsip1
                              dan akhir arsip2 terdeteksi }

ALGORITMA:

  Open(Mhs1, 1)    { buka arsip Mhs1 untuk pembacaan. }
  Open(Mhs2, 1)    { buka arsip Mhs2 untuk pembacaan. }
  Open(Mhs3, 2)    { buka arsip Mhs3 untuk penulisan. }
```

```
if (not EOF(Mhs1)) and (not EOF(Mhs2) do

    Fread(Mhs1, RekMhs1) { baca rekaman dari Mhs1 }
    Fread(Mhs2, RekMhs2) { baca rekaman dari Mhs2 }
    akhir1 ← false
    akhir2 ← false
    while (not akhir1) and (not akhir2) do
      if (RekMhs1.NIM ≤ RekMhs2.NIM) then
         Fwrite(Mhs3, RekMhs1)
         if not EOF(Mhs1) then
            Fread(Mhs1, RekMhs1)  { arsip Mhs1 maju satu rekaman }
         else
            akhir1 ← true
         endif
      else
         Fwrite(Mhs3, RekMhs2)
         if not EOF(Mhs1) then
            Fread(Mhs2, RekMhs2)  { arsip Mhs2 maju satu rekaman }
         else
            akhir2 ← true
         endif
      endif
  endwhile
  { akhir1 or akhir2 }

endif

{ salin rekaman yang tersisa, dari arsip Mhs1 atau Mhs2 }

while not akhir1 do  { jika yang tersisa adalah arsip Mhs1 }
   Fwrite(Mhs3, RekMhs1)
   if not EOF(Mhs1) then
      Fread(Mhs1, RekMhs1)  { arsip Mhs1 maju satu rekaman }
   else
      akhir1 ← true
   endif
endwhile
{ akhir1 }

while not akhir2 do     { jika yang tersisa adalah arsip Mhs2 }
   Fwrite(Mhs3, RekMhs2)
   if not EOF(Mhs2) then
      Fread(Mhs2, RekMhs2)  { arsip Mhs2 maju satu rekaman }
   else
      akhir2 ← true
   endif
endwhile
{ akhir2 }

Close(Mhs1)
Close(Mhs2)
Close(Mhs3)
```

Algoritma 17.35 Penggabungan dua arsip terurut menjadi satu arsip terurut

17.9 Pemutakhiran Arsip

Pemutakhiran (*updating*) adalah proses yang dilakukan untuk mengubah nilai rekaman tertentu di dalam arsip, menambah rekaman baru, atau menghapus rekaman tertentu. Untuk dua pemutakhiran pertama, arsip dimutakhirkan dengan nilai baru yang dibaca dari piranti masukan atau dibaca dari arsip transaksi lain. Adapun untuk penghapusan, rekaman yang akan dihapus dibaca dari piranti masukan.

Yang harus diperhatikan adalah sebuah arsip beruntun tidak dapat dibaca sekaligus ditulisi. Sebuah arsip hanya dapat dibuka untuk pembacaan saja atau untuk penulisan saja. Jadi, tidak mungkin membaca rekaman di dalam arsip, mengubahnya, dan kemudian menuliskannya kembali ke arsip tersebut.

Keterbatasan ini menyebabkan pemutakhiran arsip merupakan proses yang rumit. Pemutakhiran arsip memerlukan dua buah arsip. Satu arsip (arsip "lama") adalah arsip asal yang akan dimutakhirkan, dibuka untuk pembacaan. Arsip kedua (arsip "baru"), yang dibuka untuk penulisan, berfungsi sebagai arsip temporer, berisi hasil pemutakhiran arsip lama. Jadi, arsip "lama" merupakan arsip masukan, sedangkan arsip temporer merupakan arsip keluaran. Jika kita menginginkan arsip asal berisi hasil pemutakhiran, maka setelah pemutakhiran data, isi arsip temporer disalin kembali ke arsip asal. Arsip temporer dapat dihapus secara fisik karena tidak diperlukan lagi

```
DEKLARASI
    type DataMhs : record <NIM : integer, Nama : string, IP : real>
    type ArsipMhs : File of DataMhs
```

Algoritma 17.36 Deklarasi arsip data mahasiswa (untuk pemutakhiran)

Contoh 17.8. Misalkan arsip *Asal* menyimpan data mahasiswa (*NIM, Nama, IP*) – arsip *Asal* mungkin kosong. Kita akan menambahkan (*append*) beberapa data baru ke arsip *Asal*. Penambahan data dilakukan setelah rekaman terakhir dari arsip *Asal* dengan sejumlah data baru yang dibaca dari papan ketik. Pembacaan data berakhir jika *NIM* yang dimasukkan adalah 9999. Langkah-langkahnya adalah:

(i) buka arsip *Asal* untuk pembacaan;
(ii) buka arsip *Temp* untuk penulisan sementara;
(iii) baca seluruh rekaman dari arsip *Asal*, lalu tuliskan ke arsip *Temp;*
(iv) baca data baru dari papan ketik, tulis data tersebut ke arsip *Temp;*
(v) buka arsip *Temp* untuk pembacaan;
(vi) buka arsip *Asal* untuk penulisan;
(vii) baca seluruh rekaman dari arsip *Temp*, lalu tuliskan ke arsip *Asal*.

```
procedure  TambahDataMahasiswa(input/output Asal : ArsipMhs)
{ Menambah sejumlah data baru ke arsip Asal; data dibaca dari papan
ketik, proses penambahan data berhenti jika NIM yang dibaca = 9999. }
{ K.Awal : arsip Asal mungkin kosong. }
{ K.Akhir: arsip Asal berisi data lama plus data yang baru. }

DEKLARASI
   RekMhs  : DataMhs
   Temp : ArsipMhs  { nama arsip temporer }

ALGORITMA:
   Open(Asal, 1)
   Open(Temp, 2)

   { baca seluruh rekaman dari Asal, tulis ke Temp }
   while not EOF(Asal) do
      Fread(Asal, RekMhs)
      Fwrite(Temp, RekMhs)
   endwhile
   { EOF(Asal) }
   Close(Asal)

   read(RekMhs.NIM) {baca data NIM pertama dari papan ketik, mungkin 9999}
   while(RekMhs.NIM ≠ 9999) do        {baca data field lainnya}
      read(RekMhs.Nama)
      read(RekMhs.IP)
      Fwrite(Temp, RekMhs)     {rekam RekMhs ke dalam arsip Temp }
      read(RekMhs.NIM)
   endwhile
   { RekMhs.NIM = 9999}

   Close(Temp)

   Open(Temp, 1)          { buka arsip Temp untuk pembacaan. }
   Open(Asal, 2)          { buka arsip Asal untuk perekaman. }

   { baca seluruh rekaman dari Temp, tulis ke Asal }
   while not EOF(Temp) do
      Fread(Temp,RekMhs)
      Fwrite(Asal, RekMhs)
   endwhile
   { EOF(Temp) }

   Close(Temp)
   Close(Asal)
```

Algoritma 17.37 Penambahan data baru ke dalam arsip Asal

Contoh 17.9. Misalkan kita akan memperbaiki *IP* seorang mahasiswa. Nilai *IP* yang baru dibaca dari piranti masukan. Langkah-langkahnya sebagai berikut:

(i) buka arsip *Asal* untuk pembacaan
(ii) buka arsip *Temp* untuk penulisan
(iii) baca dan salin rekaman dari arsip *Asal* sampai ditemukan *NIM* yang dicari. Jika ditemukan, baca *IP* yang baru dari piranti masukan, ubah *IP* lama dengan *IP* yang baru, tulis rekamannya ke arsip *Temp*.

(iv) baca setiap rekaman yang yang tersisa lalu tulis ke arsip *Temp*.
(v) buka arsip *Temp* untuk pembacaan
(vi) buka arsip *Asal* untuk penulisan
(vii) baca seluruh rekaman dari arsip *Temp*, lalu tuliskan ke arsip *Asal*

Sebagai contoh, misalkan arsip *Asal* berisi lima buah rekaman sebagai berikut:

```
<359601, 'Andi', 2.96>
<359602, 'Rudi', 3.04>
<359604, 'Ahmad', 2.67>
<359607, 'Siti', 2.86>
<359608, 'Yanti', 2.94>
```

Data mahasiswa dengan *NIM* = 359604 akan diubah *IP*-nya, misalnya menjadi 2.98.

Langkah-langkahnya sebagai berikut:
(i) Buka arsip *Asal* untuk pembacaan
(ii) Buka arsip *Temp* untuk penulisan
(iii) Baca dan salin arsip *Asal* ke dalam arsip *Temp* sampai ditemukan *NIM* = 359604
Arsip *Temp* sekarang:

```
<359601, 'Andi', 2.96>
<359602, 'Rudi', 3.04>
```

(iv) Rekaman dengan *NIM* = 359604 ditemukan pada rekaman ketiga. Baca *IP* yang baru, misalnya 2.98, lalu ubah *IP* yang lama menjadi 2.98.

```
<359604, 'Ahmad', 2.98>
```

Salin rekaman yang telah dimutakhirkan ke dalam arsip *Temp*
Arsip *Temp* sekarang:

```
<359601, 'Andi', 2.96>
<359602, 'Rudi', 3.04>
<359604, 'Ahmad', 2.98>
```

(v) Salin sisa arsip *Asal* ke dalam arsip *Temp*
Arsip *Temp* sekarang:

```
<359601, 'Andi', 2.96>
<359602, 'Rudi', 3.04>
<359604, 'Ahmad', 2.98>
<359607, 'Siti', 2.86>
<359608, 'Yanti', 2.94>
```

(vi) Salin kembali arsip *Temp* ke arsip *Asal*
Arsip *Asal* sekarang:

```
<359601, 'Andi', 2.96>
<359602, 'Rudi', 3.04>
<359604, 'Ahmad', 2.98>
```

```
    <359607, 'Siti', 2.86>
    <359608, 'Yanti', 2.94>
```

```
procedure  MutakhirkanData(input/output Asal : ArsipMhs,
                           input  X : integer)
{ Mengubah IP mahasiswa yang NIM = X dengan IP yang baru. IP yang baru
dibaca dari papan ketik. }
{ K.Awal : X sudah berisi NIM mahasiswa yang akan diubah IP-nya. }
{ K.Akhir: IP mahasiswa yang NIM-nya = X sudah berubah menjadi IP yang
baru }

DEKLARASI
     RekMhs : DataMhs
     IPbaru : real
     ketemu : boolean
     Temp : ArsipMhs  { nama arsip temporer }

ALGORITMA:
        Open(Asal, 1)
     Open(Temp, 2)

     { cari rekaman dengan NIM = X, salin ke Temp jika NIM ≠ X }
     ketemu ← false
     while (not ketemu) and (not EOF(Asal) do
        Fread(Asal, RekMhs)   { baca rekaman  }
        if RekMhs.NIM = X then
           ketemu ← true
        else
           Fwrite(Temp, RekMhs)  { salin ke arsip Temp rekaman yang
                                   NIM-nya ≠ X }
        endif
     endwhile
     { ketemu or EOF(Asal) }

     if ketemu then
         {tampilkan data yang lama}
         write(RekMhs.NIM, RekMhs.Nama, RekMhs.IP)

         read(IPbaru)              { baca IP yang baru }
         RekMhs.IP ← IPbaru        { ubah IP lama dengan IPbaru }
         Fwrite(Temp, RekMhs)      { salin rekaman yang baru ke Temp }

         { salin rekaman yang tersisa yang terletak sesudah rekaman X }
         while (not EOF(Asal) do
           Fread(Asal, RekMhs)
           Fwrite(Temp, RekMhs)
         endwhile
         { EOF(Asal) }

     else
         write(X, ' tidak ada di dalam arsip Asal')
     endif

     Close(Asal)
     Close(Temp)

     { Buka kembali kedua arsip dengan fungsi yang berbeda, yaitu
       untuk menyalin isi arsip Temp ke Asal }
```

```
Open(Temp, 1)
Open(Asal, 2)
while (not EOF(Temp)) do
   Fread(Temp, RekMhs)
   Fwrite(Asal, RekMhs)
endwhile
{ EOF(Temp) }

Close(Temp)
Close(Asal)
```

Algoritma 17.38 Pemutakhiran data di dalam arsip Asal

17.10 Arsip Beruntun dalam PASCAL dan C

(a) Bagian Deklarasi

ALGORITMIK

(i) Bentuk umum pendeklarasian arsip

```
DEKLARASI
   type nama tipe arsip : File of tipe rekaman    { tipe bentukan }
   arsip : nama tipe arsip
```

(ii) Contoh arsip data mahasiswa.

```
DEKLARASI
    { tipe rekaman }
   type DataMhs : record <NIM : integer, Nama : string, IP : real>

   RekMhs : DataMhs    { rekaman }

   Mhs : File of DataMhs   { arsip }
```

(iii) Contoh arsip bilangan bulat.

```
DEKLARASI
   type ArsipInt : File of integer    { tipe bentukan }

   i   : integer      { rekaman }
   Bil : ArsipInt     { arsip }
```

(iv) Arsip karakter

```
DEKLARASI
   type ArsipKar : File of char    { tipe bentukan }

   c   : char       { rekaman }
   Kar : ArsipKar   { arsip }
```

PASCAL

(i) Bentuk umum pendeklarasian arsip

```
DEKLARASI
   type nama tipe arsip = file of tipe rekaman;    { tipe bentukan }
   arsip : nama tipe arsip;
```

(ii) Contoh arsip data mahasiswa.

```
type
  DataMhs = record
              NIM  : longint;
              Nama : string[25];
              IP   : real;
            end;

  ArsipMhs = file of DataMhs;

var
  RekMhs : DataMhs;    { rekaman }
  Mhs    : ArsipMhs;   { arsip }
```

(iii) Contoh arsip bilangan bulat.

```
type
  ArsipInt = file of integer;

var
  i   : integer;  { rekaman }
  Bil : ArsipInt; { arsip }
```

(iv) Contoh arsip karakter.

```
type
   ArsipKar = file of char;

var
   c   : char;       { rekaman }
   Kar : ArsipKar;   { arsip }
```

C

(i) Bentuk umum pendeklarasian arsip

```
FILE *arsip;
```

(ii) Contoh arsip data mahasiswa.

```
typedef  struct { long  NIM;
                  char  Nama[25];
                  float IP;
                } DataMhs;

 DataMhs RekMhs;   { rekaman }
 FILE *Mhs;    { arsip }
```

(iii) Contoh arsip bilangan bulat.

```
int i;   /* rekaman */

FILE *Bil;   /* arsip */
```

(iv) Contoh arsip karakter.

```
char c;   /* rekaman */

FILE *Kar;  /* arsip */
```

(b) Fungsi Pustaka untuk Pemrosesan Arsip

ALGORITMIK

1. `Open` (nama arsip, kode)

Contoh

```
Open (Mhs, 1)          {buka arsip Mhs untuk pembacaan }
Open (Bil, 2)          {buka arsip Bil untuk penulisan }
Open (Kar, 1)          {buka arsip Kar untuk pembacaan }
```

2. `Fread` (nama arsip, rekaman)

Contoh

```
Fread(Mhs, RekMhs)
Fread(Bil, i)
Fread(Kar, ch)
```

3. `Fwrite` (NamaArsip, NamaRek)

Contoh

```
Fwrite(Mhs, <456091, 'Ida Bagus Adi Sudewa', 3.23>)
Fwrite(Bil, 765)
Fwrite(Kar, 'R')

RekMhs.NIM ← 456087
RekMhs.Nama ← 'Hikmat Sadikin'
Rek.Mhs.IP ← 2.76
Fwrite(Mhs, RekMhs)
```

4. `Close` (NamaArsip)

Contoh

```
Close(Mhs)
Close(Bil)
Close(Kar)
```

PASCAL

1. `assign(`*arsip, nama arsip fisik*`);`
   `reset(`*arsip*`);` `{ untuk pembacaan, atau }`
   `rewrite(`*arsip*`)` `{ untuk penulisan }`

Contoh

```
assign(Mhs,'mhs.dat'); {mhs.dat adalah nama arsip fisik }
reset (Mhs);

assign(Bil,'bil.rek'); {bil.rek adalah nama arsip fisik }
rewrite(Bil);

assign(Kar,'kar.dat'); {kar.dat adalah nama arsip fisik }
reset (Kar);
```

2. `read(`*arsip, rekaman*`)`

Contoh

```
read(Mhs, RekMhs);
read(Bil, I);
read(Kar, C);
```

3. `write(`*arsip, rekaman*`)`

Contoh

```
RekMhs.NIM := 456091;
RekMhs.Nama := 'Ida Bagus Adi Sudewa';
RekMhs.Nama := 3.23;
write(Mhs, RekMhs);

write(Bil, 765);
write(Kar, 'R');

RekMMhs.NIM := 456087;
RekMhs.Nama := 'Hikmat Sadikin';
Rek.Mhs.IP := 2.76;
write(Mhs, RekMhs) ;
```

4. `Close (`*arsip*`)`

Contoh

```
Close(Mhs);
Close(Bil);
Close(Kar);
```

C

1. *arsip* = `fopen`(nama arsip fisik, `"rb"`); / untuk pembacaan, atau */
   arsip = `fopen`(nama arsip fisik, `"wb"`); / untuk penulisan */

   Keterangan: `"rb"` `r` menyatakan buka arsip untuk pembacaan,
   `b` mengindikasikan arsip biner
   `"wb"` `w` menyatakan buka arsip untuk pembacaan,
   `b` mengindikasikan arsip biner

   Contoh

```
Mhs = fopen("mhs.dat", "rb");
/* mhs.dat adalah nama arsip fisik */

Bil = fopen("bil.rek", "wb");
/* bil.rek adalah nama arsip fisik */

Kar = fopen("kar.dat", "rb");
/* kar.dat adalah nama arsip fisik */
```

2. `fread`(&rekaman, `sizeof`(rekaman), jumlah rekaman yang dibaca, arsip);

   Contoh

```
fread(&RekMhs, sizeof(RekMhs), 1, Mhs);
fread(&i, sizeof(i), 1, Bil);
fread(&c, sizeof(c), 1, Kar);
```

   Catatan: jika *rekaman* bertipe *string* atau larik, maka karakter "&" tidak boleh ditambahkan di depan *rekaman*.

3. `fwrite`(&rekaman, `sizeof`(rekaman), jumlah rekaman yang dibaca, arsip);

   Contoh

```
fwrite(&RekMhs, sizeof(RekMhs), 1, Mhs);
fwrite(&I, sizeof(I), 1, Bil);
fwrite(&C, sizeof(C), 1, Kar);
```

   Catatan:
   jika rekaman bertipe string atau larik, maka karakter "&" tidak boleh ditambahkan di depan rekaman.

4. `fclose`(*arsip*);

   Contoh

```
fclose(Mhs);
fclose(Bil);
fclose(Kar);
```

Bahasa pemrogram *Pascal* dan bahasa *C* memiliki fungsi standar *EOF*. Di dalam bahasa *Pascal*, fungsi *EOF* dinyatakan sebagai

```
EOF(arsip)
```

Fungsi tersebut mengembalikan nilai *true* jika akhir arsip terdeteksi; sebaliknya *false* jika tidak.

Dalam bahasa *C*, fungsi *EOF* dinyatakan sebagai

```
feof(arsip)
```

Fungsi ini mengembalikan nilai tidak-nol jika akhir arsip dicapai.

(c) Contoh-contoh translasi algoritma

1. Membuat arsip data mahasiswa

ALGORITMIK

```
PROGRAM BuatArsipMahasiswa
{ Membuat arsip data mahasiswa. Data mahasiswa dibaca dari papan
ketik. }

DEKLARASI
   type DataMhs : record <NIM : integer, Nama : string, IP : real>
   Msiswa : DataMhs          { peubah untuk menampung pembacaan data
                               mahasiswa }
   Mhs : File of DataMhs

ALGORITMA:
   Open(Mhs, 2)          { buka arsip Mhs untuk perekaman }
   read(Msiswa.NIM)      { baca NIM mahasiswa pertama, mungkin 9999 }
   while(Msiswa.NIM ≠ 9999) do
      read(Msiswa.Nama, Msiswa.IP)
      Fwrite(Mhs, Msiswa)  { rekam Msiswa ke dalam arsip Mhs }
      read(Msiswa.NIM)
   endwhile
   { Msiswa.NIM = 9999}

   Close(Mhs)
```

PASCAL

```
PROGRAM BuatArsipMahasiswa;
{       Membuat arsip data mahasiswa. Data mahasiswa dibaca dari papan
ketik. Nama arsip adalah 'data.dat'. }

(* DEKLARASI *)
type
    DataMhs = record
                NIM  : longint;
```

```
                Nama : string[25];
                IP   : real;
              end;
var
   Msiswa : DataMhs;          { peubah untuk menampung pembacaan data
                                mahasiswa }
   Mhs : File of DataMhs;

(* ALGORITMA:*)
begin
   assign(Mhs, 'data.dat');
   rewrite(Mhs);          { buka arsip Mhs untuk perekaman }

   { baca NIM mahasiswa pertama, mungkin 9999 }
   write('NIM   : '); readln(Msiswa.NIM);
   while(Msiswa.NIM <> 9999) do
    begin
      write('Nama  : '); readln(Msiswa.Nama);
      write('IP    : '); readln(Msiswa.IP);
      write(Mhs, Msiswa);  { rekam Msiswa ke dalam arsip Mhs }
      write('NIM   : '); readln(Msiswa.NIM);
   end; {while}
   { Msiswa.NIM = 9999}

   close(Mhs);
end.
```

C

```
/* PROGRAM BuatArsipMahasiswa */
/* Membuat arsip data mahasiswa. Data mahasiswa dibaca dari papan
ketik. Nama arsip adalah 'data.dat'. */

main()
{
   /* DEKLARASI */
   typedef struct { long NIM;
                    char Nama[25];
                    float IP;
                  } DataMhs;
   DataMhs Msiswa;        /* peubah untuk menampung pembacaan data
                             mahasiswa */
   FILE *Mhs;

   /* ALGORITMA:*/
   Mhs = fopen("data.dat", "wb"); /* buka arsip Mhs untuk perekaman */

   /* baca NIM mahasiswa pertama, mungkin 9999 */
   printf("NIM   : "); scanf("%ld", &Msiswa.NIM);
   while(Msiswa.NIM != 9999)
    {
      printf("Nama  : "); scanf("%s", Msiswa.Nama);
      printf("IP    : "); scanf("%f", &Msiswa.IP);
      /* rekam Msiswa ke dalam arsip Mhs */
      fwrite(&Msiswa, sizeof(Msiswa), 1, Mhs);
      printf("NIM   : "); scanf("%ld", &Msiswa.NIM);
    }/*while*/
```

```
   /* Msiswa.NIM = 9999 */

   fclose(Mhs);
}
```

2. Membaca rekaman dari arsip data mahasiswa

ALGORITMIK

```
PROGRAM BacaArsipMahasiswa
{ Membaca rekaman dari arsip data mahasiswa, dan menampilkannya ke
  layar. }

DEKLARASI
   type DataMhs : record <NIM : integer, Nama : string, IP : real>
   Msiswa : DataMhs { peubah untuk menampung pembacaan data mahasiswa}
   Mhs : File of DataMhs

ALGORITMA:
   Open(Mhs, 1)          { buka arsip Mhs untuk pembacaan }
   While not EOF(Mhs)
      Fread(Mhs, Msiswa)
      write(Msiswa.NIM, Msiswa.Nama, Msiswa.IP)
   endwhile
   { EOF(Mhs) }

   Close(Mhs)
```

PASCAL

```
PROGRAM BacaArsipMahasiswa;
{ Membaca rekaman dari arsip data mahasiswa dan menampilkannya ke
layar. Nama arsip data mahasiswa adalah 'data.dat'. }
(* DEKLARASI *)
type
    DataMhs = record
                NIM   : longint;
                Nama  : string[25];
                IP    : real;
              end;
var


   Msiswa : DataMhs;        { peubah untuk menampung pembacaan data
                              mahasiswa }
   Mhs : File of DataMhs;

(* ALGORITMA:*)
begin
   assign(Mhs, 'data.dat');
   reset(Mhs);          { buka arsip Mhs untuk pembacaan }

   while not EOF(Mhs) do
    begin
```

```
      read(Mhs, Msiswa)
      writeln('NIM : ', Msiswa.NIM);
      writeln('Nama: ', Msiswa.Nama);
      writeln('IP  : ', Msiswa.IP);
    end;{while}
   { EOF(Mhs) }

   close(Mhs);
end.
```

C

```
/* PROGRAM BacaArsipMahasiswa */
/* Membaca rekaman dari arsip data mahasiswa dan menampilkannya ke
layar. Data mahasiswa dibaca dari papan ketik.
*/

typedef enum {true = 1, false = 0} boolean;   /* tipe data boolean */

main()
{
   /* DEKLARASI */
   typedef struct { long NIM;
                    char Nama[25];
                    float IP;
                  } DataMhs;

   DataMhs Msiswa;  /* peubah untuk menampung pembacaan data mahasiswa
*/
   FILE *Mhs;


   /* ALGORITMA:*/
   Mhs = fopen("data.dat", "rb"); /* buka arsip Mhs untuk pembacaan */

    /* baca rekaman pertama, mungkin MARK */
   while (!feof(Mhs)
   {
      fread(&Msiswa, sizeof(Msiswa), 1, Mhs);
      printf("NIM : %ld \n", Msiswa.NIM);
      printf("Nama: %s  \n", Msiswa.Nama);
      printf("IP  : %f  \n", Msiswa.IP);
   }

   fclose(Mhs);
}
```

17.11 Arsip sebagai Parameter Prosedur

Sering arsip dilewatkan sebagai parameter prosedur/fungsi. Contoh 17.5 dan seterusnya memperlihatkan penggunaan arsip sebagai parameter prosedur. Pada kasus seperti ini, arsip haruslah dilewatkan sebagai parameter keluaran (meskipun di dalam notasi algoritmik suatu arsip dilewatkan sebagai

parameter masukan). Hal ini untuk menghindari kebutuhan penyalinan arsip, sebagaimana yang dilakukan jika arsip dilewatkan sebagai parameter masukan (*by value*). Contoh-contoh berikut ini memperlihatkan cara mentranslasi prosedur yang mengandung parameter arsip.

ALGORITMIK

```
procedure CariNIM(input Mhs : ArsipMhs, input Nomor : integer,
                  output ketemu : boolean)
{ Mencari data mahasiswa yang NIM-nya = Nomor. }
{ K.Awal : Arsip Mhs tidak kosong, Nomor sudah berisi NIM mahasiswa
  yang dicari. }
{ K.Akhir: Jika mahasiswa yang NIM-nya = Nomor ditemukan, ketemu
berisi true, sebaliknya jika tidak ada maka ketemu berisi false }

DEKLARASI:
  RekMhs : DataMhs

ALGORITMA:
  Open(Mhs, 1)
  ketemu ← false
  while (not EOF(Mhs)) and (not ketemu) do
     Fread(Mhs, RekMhs)
     if RekMhs.NIM = Nomor then
        ketemu ← true
     endif
  endwhile
  { EOF(Mhs) or ketemu }
```

PASCAL

```
procedure CariNIM(var Mhs : ArsipMhs; Nomor : integer;
                  var ketemu : boolean);
{ Mencari data mahasiswa yang NIM-nya = Nomor. }
{ K.Awal : Arsip Mhs tidak kosong, Nomor sudah berisi NIM mahasiswa
  yang dicari. }
{ K.Akhir: Jika mahasiswa yang NIM-nya = Nomor ditemukan, ketemu
berisi true, sebaliknya jika tidak ada maka ketemu berisi false }

(* DEKLARASI: *)
var
  RekMhs : DataMhs;

(* ALGORITMA: *)
begin
  assign(Mhs, 'data.dat');
  reset(Mhs);          { buka arsip Mhs untuk pembacaan }
  ketemu := false;
  while (not EOF(Mhs)) and (not ketemu) do
   begin
     read(Mhs, RekMhs);
     if RekMhs.NIM = Nomor then
        ketemu := true;
     {endif}
```

```
    end; {while}
  { EOF(Mhs) or ketemu }
end;
```

C

```
void CariNIM(FILE *Mhs, long Nomor, boolean *ketemu)
/* Mencari data mahasiswa yang NIM-nya = Nomor. */
/* K.Awal : Arsip Mhs tidak kosong, Nomor sudah berisi NIM mahasiswa
   yang dicari. Perhatikan, di sini arsip Mhs harus dilewatkan sebagai
parameter keluaran. */
/* K.Akhir: Jika mahasiswa yang NIM-nya = Nomor ditemukan, ketemu
berisi true, sebaliknya jika tidak ada maka ketemu berisi false.

   Tipe boolean dianggap sudah dideklarasikan di luar prosedur sebagai
   berikut:

     typedef enum {true = 1, false = 0} boolean;
*/

{
  /* DEKLARASI: */

  DataMhs Msiswa;

  /* Fungsi Mark diletakkan di luar prosedur */

  /* ALGORITMA: */

  Mhs = fopen("data.dat", "rb"); /* buka arsip Mhs untuk pembacaan */
  *ketemu = false;
  while (!feof(Mhs) && !(*ketemu))
    {
      fread(&Msiswa, sizeof(Msiswa), 1, Mhs);
      if (Msiswa.NIM == Nomor)
         *ketemu = true
      /*endif */
    }/*while*/
  /* feof(Mhs) || *ketemu */
}
```

17.12 Arsip Teks

Arsip teks adalah jenis arsip beruntun yang khusus. Arsip teks berbeda dengan tipe *File of char* yang telah kita gunakan pada bagian sebelumnya. Teks terdiri dari baris-baris karakter, setiap baris diakhiri dengan karakter *end of line*, sesuatu yang tidak terdapat di dalam *file of char*. Di dalam bab-bab sebelumnya kita sudah menggunakan arsip teks sebagai tempat membaca data masukan. Jika kita mengabaikan pemrosesan karakter *end of line*, maka kita dapat membayangkan teks sebagai *string* yang panjang, seperti contoh pada Gambar 17.4. Algoritma pemrosesan arsip beruntun yang sudah kita bahas sebelum ini dapat kita terapkan pada arsip teks.

```
...sejak 100 tahun yang lalu kota bandung sudah dihuni oleh...
```

Gambar 17.4 Contoh sebuah teks

Informasi di dalam teks diproses dan dimanipulasi. Contoh pemrosesan teks antara lain mencari (*search*) sebuah karakter atau *string*, menghitung jumlah karakter atau jumlah kata, menempatkan *head* pembacaan ke karakter/kata tertentu, dan sebagainya.

Banyak aplikasi yang di dalamnya melakukan pemrosesan teks. Program pengolah kata (*text editor*) seperti *Notepad* dan *Word* adalah contoh nyata aplikasi pengolahan teks. Dengan program pengolah kata tersebut kita dapat mencari kata tertentu atau mengganti sebuah kata dengan kata lain. Kompilator bahasa pemrograman seperti *Turbo Pascal, Turbo C, Borland Delphi, Borland C*, dan sebagainya, sering dilengkapi dengan editor teks untuk menyunting program. Tugas pertama kompilator pada waktu kompilasi adalah membaca program sumber untuk mengambil simbol-simbol (*keyword*, peubah, konstanta, dan sebagainya). Komponen kompilator yang melakukan aktivitas ini dinamakan program *scanner*.

Misalkan kita mempunyai sebuah arsip teks yang kita beri nama *T*. Deklarasi teks di dalam algoritma kita notasikan sebagai berikut:

```
DEKLARASI
   T : text    { T adalah peubah teks }
```

Algoritma 17.39 Deklarasi T sebagai teks di dalam algoritma

Di bawah ini kita berikan beberapa contoh pemrosesan karakter di dalam arsip teks.

Contoh 17.10. Buatlah algoritma untuk menghitung banyaknya karakter di dalam teks (tidak termasuk karakter '.') .

Penyelesaian

```
procedure HitungBanyakKar(input T : text, output n : integer)
{ Menghitung banyaknya karakter di dalam teks. }
{ K.Awal: T mungkin kosong. }
{ K.Akhir: n berisi banyaknya karakter di dalam teks. }

DEKLARASI
   C : char

ALGORITMA:
   n ← 0
   Open(T, 1)     { buka arsip teks untuk pembacaan }
```

```
  while not EOF(T) do
     Fread(T, C)    { baca karakter }
     n ← n + 1
  endwhile
  { EOF(T) }
```

Algoritma 17.40 Menghitung jumlah karakter di dalam teks

Contoh 17.11. Buatlah algoritma untuk mencari karakter tertentu di dalam teks. Hasil pencarian adalah sebuah peubah *boolean* yang bernilai *true* jika karakter ditemukan, atau bernilai *false* jika tidak ditemukan.

Penyelesaian

Misalkan karakter yang dicari disimpan di dalam peubah *X*, dan karakter yang dibaca dari teks disimpan di dalam peubah *C*.

```
procedure Cari(input T : text, input X : char, output ketemu :
boolean)
{ Mencari X di dalam teks. }
{ K.Awal: X sudah terdefinisi dengan suatu karakter. }
{ K.Akhir: ketemu bernilai true jika X ditemukan. Jika X tidak
ditemukan, ketemu bernilai false. }

DEKLARASI
   C : char

ALGORITMA:
    ketemu ← false
    Open(T, 1)
    while (not EOF(T)) and (not ketemu) do
       Fread(T, C)
       if C = X then
         ketemu ← true
       endif
    endwhile
    { EOF(T) or ketemu }
```

Algoritma 17.41 Mencari karakter di dalam teks (versi 2).

Contoh program yang memanggil prosedur `Cari` (baik untuk algoritma versi 1 maupun algoritma versi 2):

```
PROGRAM PencarianDidalamTeks
{ Program untuk mencari karakter tertentu di dalam teks }

DEKLARASI
  T     : text
  X     : char
  found : boolean  { true jika X ditemukan, atau false jika tidak }

  procedure Cari(input T : text, input X : char, output ketemu : boolean)
```

```
{ Mencari X di dalam teks. Tanda akhir teks adalah karakter '.' }

ALGORITMA:
   read(X)
   Cari(T, X,found)
   if found then
      write('tidak ditemukan')
   else
      write('ditemukan! )
   endif
```

Algoritma 17.42 Program pemanggil prosedur Cari.

Contoh 17.12. Buatlah algoritma untuk menghitung jumlah karakter 'a' di dalam teks. Tanda akhir teks adalah karakter titik.

Penyelesaian

```
procedure HitungKarakter_a(input T : text, output n : integer)
{ Menghitung banyaknya karakter 'a' di dalam teks. }
{ K.Akhir: T mungkin kosong. }
{ K.Akhir: n berisi banyak karakter 'a'di dalam teks. }

DEKLARASI
   C : char

ALGORITMA:
   n ← 0
   Open(T, 1)
   while not EOF(T) do
     Fread(T, C)
     if C = 'a' then
        n ← n + 1
     endif
   endwhile
   { EOF(T) }
```

Algoritma 17.43 Menghitung jumlah kemunculan karakter 'a'.

Contoh 17.13. Tulislah algoritma untuk menghitung jumlah kemunculan pasangan huruf *'an'* di dalam teks. Misalnya pada contoh teks di bawah ini jumlah *'an'* ada 3 buah (ditandai dengan garis bawah).

```
pandangan mata
```

Penyelesaian

Gagasan yang digunakan di dalam algoritma adalah dengan menggunakan dua buah nama peubah, *C* dan *lastC*. Peubah *C* menyimpan karakter yang sedang dibaca, sedangkan *lastC* menyimpan karakter yang sudah dibaca sebelumnya. Setiap kali membaca, kita memeriksa apakah *lastC* = 'a' dan *C* = 'n'. Di awal algoritma, *lastC* kita inisialisasi dengan karakter spasi.

```
procedure HitungPasangan_an(input T : text, output n : integer)
{ Menghitung banyak pasangan huruf 'an' di dalam teks. }
{ K.Awal: teks T sudah terdefinisi isinya. T mungkin kosong. }
{ K.Akhir: nAN berisi banyaknya 'an' di dalam teks. }

DEKLARASI
   C, lastC : char

ALGORITMA:
   n ← 0
   lastC ← ' '
   Open(T, 1)

   while not EOF(T) do
      Fread(T, C)
      if (lastC = 'a') and (C = 'n') then
         n ← n + 1
      endif
      lastC ← C { untuk pembacaan berikutnya, C sekarang menjadi lastC }
   endwhile
   { EOF (T) }
```

Algoritma 17.44 Menghitung jumlah kemunculan 'an'

Contoh 17.14. Buatlah algoritma untuk menghitung banyaknya kata di dalam pita karakter. Kata adalah deretan karakter bukan spasi yang diakhiri dengan spasi atau *end of file* (*EOF*).

Contohnya, teks berikut ini:

```
1 hari ini  hujan badai
```

berisi 5 buah kata (yaitu: `1`, `hari`, `ini`, `hujan`, dan `badai`), dan teks yang berikut ini:

```
A
```

hanya berisi satu buah kata (yaitu: `A`).

Algoritma menghitung jumlah kata di dalam teks kita tulis dala dua versi. Versi pertama menggunakan informasi karakter yang sekarang dibaca dan karakter sebelumnya, sedangkan versi kedua mengabaikan spasi sampai ditemukan karakter bukan spasi.

Versi 1:
Gagasannya sama seperti pada Contoh 17.44. Karakter yang sekarang dibaca disimpan di dalam *C*, sedangkan karakter yang dibaca sebelumnya disimpan di dalam *lastC*. Sebuah kata ditandai dengan salah satu dari ciri berikut:
(i) *lastC* sembarang karakter bukan spasi dan *C* = ' '
(ii) *lastC* sembarang karakter bukan spasi dan *C* = karakter *end of file*

```
procedure HitungBanyakKata(input T : text, output nKata : integer)
{ Menghitung jumlah kata kata di dalam teks. }
{ K.Awal: T mungkin kosong. }
{ K.Akhir: nKata berisi banyaknya kata di dalam teks.}

DEKLARASI
   C, lastC : char

ALGORITMA:
   nKata ← 0
   lastC ← ' '
   Open(T, 1)
   while not EOF(T) do
      Fread(T, C)
      if (lastC ≠ ' ') and (C = ' ') then
         nKata ← nKata + 1
      endif
      lastC ← C  {untuk pembacaan berikutnya, C sekarang menjadi lastC }
   endwhile
   { EOF(T) }

   { kata terakhir diproses khusus }
   if lastC ≠ ' ' { dan 'end of file' terdeteksi} then
      nKata ← nKata + 1
   endif
```

Algoritma 17.45 Menghitung jumlah kata (versi 1).

Versi 2:

Gagasan yang dipakai dalam versi 2 adalah "mengabaikan" (*ignore*) semua karakter spasi hingga ditemukan sebuah karakter yang bukan spasi. Selama karakter yang dibaca bukan spasi atau baca terus teks sampai ditemukan karakter spasi lagi atau *EOF*. Pada kondisi ini, satu kata ditemukan. Sebelum dan sesudah suatu kata mungkin saja ada spasi, karena itu pengabaian spasi perlu dilakukan di awal dan di akhir kata.
Prosedur untuk mengabaikan karakter spasi:

```
procedure AbaikanSpasi(input T : text)
{ Membaca deretan spasi dan mengabaikannya. }
{ K.Awal: sembarang. }
{ K,Akhir: karakter bukan spasi ditemukan. }

DEKLARASI
  C : char

ALGORITMA:
   {selama ada spasi }
    while (C = ' ') and (not EOF(T)) do
      Fread(T, C)
    endwhile
    { C ≠ ' '}
```

Algoritma 17.46 Mengabaikan spasi yang dibaca (*ignore blank*)

Prosedur menghitung jumlah kata:

```
procedure HitungBanyakKata(input T : text, output nKata : integer)
{ Menghitung banyak kata di dalam teks. }
{ K.Awal: nKata belum terdefinisi nilainya. }
{ K.Akhir: nKata berisi banyak kata di dalam teks. }

DEKLARASI
   C : char

   procedure AbaikanSpasi(input T : text)
   { Membaca deretan spasi dan mengabaikannya. }

ALGORITMA:
   nKata ← 0
   Open(T, 1)
   while not EOF(T) do

      read(T, C)
      AbaikanSpasi(T)

      if not EOF(T) then

        { baca terus teks sampai ketemu spasi atau EOF }
        repeat
          read(T, C)
        until (C = ' ') or (EOF(T))
        nKata ← nKata + 1    { satu kata ditemukan }

      else { arsip hanya 1 karakter, langsung EOF }
        nKata ← nKata + 1    { satu kata ditemukan }
      endif
   endwhile
   { EOF(T) }

   Close(T)
```

Algoritma 17.47 Menghitung jumlah kata (versi 2)

Contoh 17.15. Tulislah algoritma untuk mencari apakah kata tertentu ditemukan di dalam teks. Tanda akhir teks adalah karakter titik.

Penyelesaian

Gagasan algoritma untuk masalah ini adalah menyambung setiap karakter (bukan spasi atau titik) yang ditemukan dengan karakter yang sudah dibaca sebelumnya. Penyambungan ini dapat menggunakan operator '+' (ingatlah bahwa tipe *string* memiliki operator '+'). Misalkan peubah yang menyimpan karakter pembentuk kata bernama kata. Pada mulanya, kata kita diinisialisasi dengan karakter kosong (*null*, bukan spasi):

```
kata ← ''
```

Setiap kali karakter (*C*) yang dibaca bukan spasi atau titik, sambungkan *C* dengan kata:

```
kata ← kata + C
```

Setelah satu kata ditemukan, bandingkan apakah kata sama dengan *string s*. Bila kata 'PROGRAM' ditemukan, pemrosesan teks dihentikan. Untuk maksud ini, peubah *boolean* yang bernama ketemu digunakan. Peubah ketemu bernilai `true` jika kata yang dicari ditemukan, dan bernilai `false` jika tidak ditemukan.

```
procedure CariKata(input T : text, input s : string,
                   output ketemu : boolean)
{ Mencari kata tertentu di dalam teks T. }
{ K.Awal: T mungkin kosong; s sudah berisi kata yang akan dicari.  }
{ K.Akhir: ketemu bernilai true jika s ditemukan, atau false jika
sebaliknya.}

DEKLARASI
   C     : char
   kata  : string

   procedure AbaikanSpasi(input T : text)
   { Membaca deretan spasi dan mengabaikannya. }

ALGORITMA:
   Open(T, 1)
   ketemu ← false
   while (not EOF(T)) and (not ketemu) do
      kata ← ''
      Fread(T, C)
      AbaikanSpasi(T)

      { baca terus teks sampai ditemukan spasi atau EOF }
      if not EOF(T) then

         repeat
           kata ← kata + C
           Fread(T, C)
         until EOF(T) or (C = ' ')
         if EOF(T) then  { jika akhir arsip tercapai, sambungkan karakter
                           terakhir yang dibaca, C, dengan kata }
            kata ← kata + C
         endif

      else { arsip hanya 1 karakter, langsung EOF }
         kata ← kata + C
      endif

      { bandingkan kata dengan string yang dicari, s }
      if kata = s then
         ketemu ← true
      endif

   endwhile
   { C = '.' or ketemu }

   Close(T)
```

Algoritma 17.48 Mencari kata tertentu di dalam teks

Contoh program pemanggil:

```
PROGRAM PencarianKataDidalamTeks
{ Program utama untuk pencarian kata tertentu di dalam teks. }

DEKLARASI
  T     : text
  word  : string
  ketemu : boolean

  procedure CariKata(input T : text, input s : string,
                     output ketemu : boolean)
  { Mencari kata tertentu di dalam teks T. }

ALGORITMA:
   read(word) {baca kata yang akan dicari, misalnya 'PROGRAM'}
   CariKata(T, word,ketemu)
   if ketemu then
      write(' ditemukan')
   else
      write(' tidak ditemukan!')
   endif
```

Algoritma 17.49 Program pemanggil CariKata

Contoh 17.16. Misalkan kita memiliki sebuah teks program dalam bahasa *C* yang sudah benar secara sintaks. Kita ingin mencari jumlah pasangan '{' dan '}' di dalam teks program tersebut. Setiap pasangan '{' dan '}' menyatakan satu buah blok program.

Sebagai contoh, teks program *C* di bawah ini mempunyai dua buah pasangan '{' dan '}'.

```
#include <stdio.h>
main()
{
  int m, p;

  p=0;
  for(m=1;m<=100;m++)
   {  p+=2;
      printf("%d",p);
   }
}
```

Penyelesaian

Gagasan dari algoritma di bawah ini sederhana. Karena teks program *C* diasumsikan sudah benar secara sintaks, maka kita cukup hanya menghitung jumlah kemunculan karakter '{' saja (sedangkan '}' pasangannya tidak perlu diperiksa keberadaannya).

```
procedure HitungJumlahKurawal(input T : text, output n : integer)
{ Menghitung jumlah pasangan '{' dan '}' di dalam program C yang dinyatakan
sebagai teks T. Diasumsikan program C sudah benar secara sintaks. }
{ K.Awal: T mungkin kosong. }
{ K.Akhir: n berisi jumlah pasangan '{' dan '}' .}

DEKLARASI
   C     : char
ALGORITMA:
   n ← 0
   Open(T, 1)
   while not EOF(T) do
     Fread(T, C)
     if C = '{' then
        n ← n + 1
     endif
   endwhile
   { EOF(T) }

   Close(T)
```

Algoritma 17.50 Menghitung jumlah kemunculan pasangan "{" dan "}"

Contoh 17.17. Sering diperlukan data di dalam arsip disimpan ke dalam larik agar pemrosesan data dapat lebih cepat. Diberikan sebuah arsip teks *T*. Buatlah algoritma untuk membaca teks dan menyimpan setiap karakter di dalam teks di dalam sebuh larik karakter.

Penyelesaian

Perhatikan bahwa teks mungkin kosong. Jika teks kosong, maka larik karakter tidak terdefinisi (ukurannya = 0).

```
procedure BacaTeksKeLarik(input T : text, output K : LarikKar,
                          output n : integer)
{ Membaca rangkaian karakter di dalam teks dan menyimpan setiap
karakter di dalam dalam larik K[1..n]. }
{ K.Awal: T mungkin kosong. }
{ K.Akhir: K berisi karakter-karakter di dalam teks; n berisi
  banyaknya karakter di dalam teks (= ukuran larik). Jika T kosong,
maka LarikKar tidak terdefinisi dan n = 0 }

DEKLARASI
   C : char

ALGORITMA:
   n ← 0
   Open(T, 1)
   while not EOF(T) do
      read(T;C)
      n ← n + 1
      K[n] ← C     { simpan C ke dalam elemen K[n] }
   endwhile
   { EOF(T) }
   Close(T)
```

Algoritma 17.51 Menyimpan karakter di dalam arsip teks ke dalam larik

Soal Latihan Bab 17

1. Dimisalkan arsip *Mhs* sudah berisi data mahasiswa (arsip tidak kosong). Tuliskan algoritma untuk menghapus sebuah rekaman dengan *NIM* tertentu (*NIM* dibaca dari papan ketik).

2. Dimisalkan arsip *Mhs* sudah berisi data mahasiswa (arsip tidak kosong). Tuliskan algoritma untuk menampilkan data rekaman yang ke-*k* (*k* dibaca dari papan ketik). Jika rekaman ke-*k* tidak ada, maka ditampilkan pesan bahwa rekaman ke-*k* tidak ada.

3. Diberikan 2 buah arsip data mahasiswa, *Mhs1* dan *Mhs2*. Buatlah fungsi untuk menentukan apakah kedua arsip tersebut sama.

4. Sebuah arsip *integer* menyimpan nilai-nilai positif. Bagaimana mengurutkan isi arsip dengan bantuan larik? Tuliskanlah algoritmanya.

5. Misalkan arsip *Mhs* sudah terurut menaik berdasarkan NIM. Tuliskan algoritma untuk menyisipkan data baru ke dalam arsip pada posisi yang tepat (sehingga arsip tetap terurut).

6. Dapatkah metode pencarian bagidua dilakukan pada arsip beruntun (yang datanya telah terurut menaik)? Mengapa?

7. Misalkan terdapat dua buah arsip beruntun yang berisi data mahasiswa. Struktur arsip pertama: *NIM*, *Nama*, *Alamat*, *Nomor Telpon*, dan struktur arsip kedua: *NIM*, *Jenis Kelamin*, *IP*. Masing-masing arsip sudah terurut berdasarkan *NIM*. Tuliskan algoritma untuk menghasilkan sebuah arsip baru yang berisi irisan atau *intersection* dari kedua arsip tersebut berdasarkan *NIM*. Arsip yang baru memuat data *NIM*, *Nama*, dan *IP*. Misalnya, jika pada arsip pertama terdapat rekaman <13596021, 'Ahmad', 'Jl. Dago 27', 250324> dan pada arsip kedua terdapat rekaman <13596021, 'P', 3.06>, maka arsip baru berisi <13596021, 'Ahmad', 3.06>.

8. Untuk semua soal 1 sampai 9, translasikan algoritmanya ke dalam bahasa *Pascal* dan bahasa *C*.

9. Tulislah fungsi untuk mencari kemunculan pertama kali untaian 'an' di dalam sebuah arsip teks. Fungsi mengembalikan nilai *true* jika 'an' terdapat di dalam teks dan *false* jika tidak ada.

10. Modifikasilah algoritma pada Contoh 17.13 sehingga dapat digunakan untuk menghitung jumlah kemunculan untaian 'kan'.

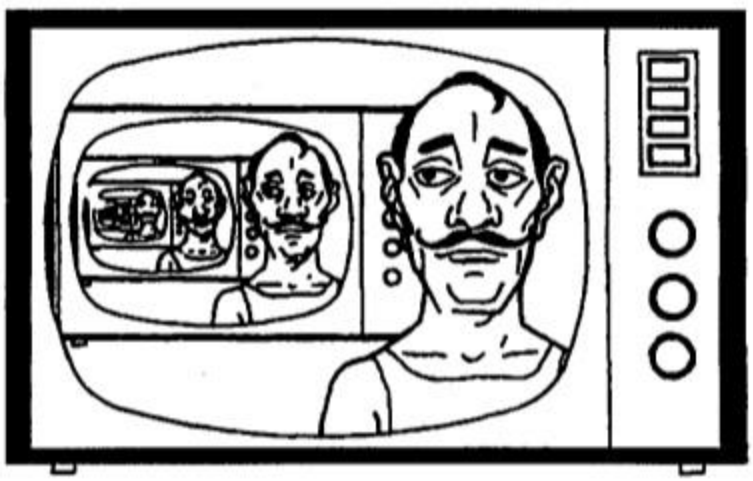
11. Diberikan sebuah arsip teks yang telah berisi program dalam bahasa *Pascal*. Teks mungkin saja kosong. Diasumsikan program tersebut sudah benar secara sintaks dan semua kata kunci di dalam program tersebut ditulis dalam huruf kecil (*lowercase*). Tulislah prosedur yang memberikan keluaran jumlah pasangan 'begin' dan 'end'.

12. Tulislah prosedur untuk menghitung jumlah kemunculan huruf hidup/vokal di dalam arsip teks. Teks mungkin saja kosong.

13. Tulislah berupa prosedur untuk mencatat pada urutan keberapa – dihitung dari awal teks – ditemukan kemunculan pertama kali karakter 'x' di dalam teks.

14. Tulislah prosedur untuk mencari kata terpanjang di dalam teks. Prosedur memberikan keluaran kata terpanjang. Jika teks kosong, maka kata terpanjang yang dihasilkan adalah *string* kosong.

18
Algoritma Rekursif



Gambar TV yang rekursif (Sumber: WIR[76])

Kehidupan bersifat ofensif,
mengarah ke pengulangan mekanisme alam semesta.
(Alfred Whitehead - *Adventures of Ideas*, 1933)

Bab 18 ini memperkenalkan rekursifitas, yaitu salah satu kakas (*tool*) yang sangat penting dalam pemrograman. Disebut sangat penting karena rekursif menyediakan teknik penyelesaian persoalan yang di dalamnya mengandung definisi persoalan itu sendiri. J.S. Rohl, seorang dosen di Universitas Western Australia, menyatakan bahwa rekursif adalah *cinderella*-nya teknik pemrograman [ROH84]. Sayangnya, rekursif merupakan materi yang paling sulit dimengerti oleh pemula pemrograman. Pembahasan materi algoritma rekursif menggunakan banyak contoh agar pembaca mempunyai pemahaman yang kuat tentang rekursif.

18.1 Proses Rekursif

Banyak objek di dalam matematika yang didefinisikan dengan cara menyatakan suatu proses (algoritma) yang menghasilkan objek tersebut. Sebagai contoh, π diperoleh dengan membagi keliling lingkaran (K) dengan diameternya (d). Dengan kata lain, proses (algoritma) untuk memperoleh nilai π dinyatakan sebagai berikut:

1. baca keliling lingkaran, K
2. baca diameter lingkaran, d
3. hitung $\pi = K/d$

Yang jelas, proses tersebut harus berhenti dengan memberikan hasil yang didefinisikan.

Contoh objek lain yang diperoleh dari suatu proses adalah faktorial. Faktorial dari bilangan bulat tak-negatif n didefinisikan sebagai berikut:

$$n! = \begin{cases} 1 & ,n = 0 \\ 1 \times 2 \times \cdots \times (n-1) \times n & ,n > 0 \end{cases}$$

Sebagai contoh,

$0! = 1$
$1! = 1$
$2! = 1 \times 2$
$3! = 1 \times 2 \times 3$
$4! = 1 \times 2 \times 3 \times 4$

Proses untuk menghitung faktorial dari bilangan bulat n tak-negatif dinyatakan dalam algoritma berikut ini:

```
function Faktorial(input n : integer)→ integer
{mengembalikan nilai n!}

DEKLARASI
   i : integer
   F : integer

ALGORITMA:
   F ← 1
   i ← 1
   while i ≤ n do
      F ← F * i
      i ← i + 1
   endwhile
   { i > n }

   return F
```

Algoritma 18.1 Fungsi Faktorial (Iteratif)

Algoritma `Faktorial` seperti di atas dinamakan algoritma **iteratif** karena ia mengerjakan pengulangan sekumpulan aksi sampai kondisi berhentinya tercapai [TEN86]. Fungsi `Faktorial` mengembalikan nilai faktorial dari *n*.

Algoritma iteratif adalah algoritma yang mengerjakan pengulangan aksi selama kondisi pengulangan masih dipenuhi (*true*). Secara umum, skema algoritma iteratif sebagai berikut:

```
procedure Iteratif(input x : xType)
{ prosedur yang memanipulasi nilai x
  K.Awal : x terdefinisi nilainya
    K.Akhir: sekumpulan aksi yang memanipulasi nilai x telah dilakukan
}

DEKLARASI
   { kamus data lokal, jika ada }

ALGORITMA:
  Inisialisasi

  while KONDISI(x) do
      AKSI(x)   { kumpulan aksi yang memanipulasi nilai x }
      F(x)      { aksi yang mengubah nilai x }
  endwhile
  { not KONSISI(x) }
  Terminasi
```

Algoritma 18.2 Skema Algoritma Iteratif

Sekarang coba perhatikan bahwa faktorial dari *n* dapat didefinisikan dalam terminologi faktorial juga:

$$0! = 1$$
$$1! = 0! \times 1$$
$$2! = 1! \times 2$$
$$3! = 2! \times 3$$
$$4! = 3! \times 4$$

Nyatalah, bahwa untuk $n > 0$ kita melihat bahwa

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n = (n-1)! \times n.$$

Dengan menggunakan notasi matematika, maka $n!$ didefinisikan dalam hubungan rekursif sebagai berikut:

$$n! = \begin{cases} 1 & , n = 0 \\ n \times (n-1)! & , n > 0 \end{cases}$$

Kita dapat melihat bahwa dalam proses perhitungan faktorial terdapat definisi faktorial itu sendiri. Cara pendefinisian seperti itu, yang mendefiniskan sebuah objek dalam terminologi dirinya sendiri dinamakan

definisi rekursif [TEN86]. Proses yang melibatkan rekursifitas disebut dengan proses rekursif.

18.2 Definisi Rekursif

Definisi rekursif diturunkan secara matematik. Definisi yang tidak formal menyatakan bahwa sebuah objek dikatakan *rekursif* jika ia didefinisikan menjadi lebih sederhana dalam terminologi dirinya sendirinya. Nicklaus Wirth mendefinisikan rekursif sebagai berikut:

> *An object is said be recursive if it partially consist or is defines in terms of itself [WIR76]*

Dalam kehidupan sehari-hari banyak terdapat objek yang rekursif. Tahukah Anda bahwa, kalau diamati dengan saksama, daun pakis (Gambar 18.1) dibentuk oleh ranting-ranting daun yang mempunyai pola yang mirip dengan daun pakis itu sendiri. Setiap ranting daun disusun lagi oleh ranting daun dengan pola yang mirip. Hal yang sama juga tampak pada pohon cemara. Objek rekursif yang khas ini disebut **fraktal**. Dalam bidang grafik dan seni, fraktal dimanfaatkan untuk membangkitkan gambar-gambar yang indah dan menawan.

Gambar 18.1 Fraktal tanaman pakis

Taufik Ismail, seorang penyair sufistik, pernah membuat puisi yang menyiratkan rekursifitas sebagai berikut:

Fariduddin Attar bangunlah pada malam hari[1]
Dan dia memikirkan tentang dunia ini

Ternyata dunia ini
Adalah sebuah peti

Sebuah peti yang besar dan tertutup di atasnya
Dan kita manusia berputar-putar di dalamnya

Dunia sebuah peti yang besar
Dan tertutup di atasnya

Dan kita terkurung di dalamnya
Dan kita berjalan-jalan di dalamnya
Dan kita bermenung di dalamnya
Dan kita beranak di dalamnya
Dan kita membuat peti di dalamnya

Dan kita membuat peti
di dalam peti ini
....

Temukan bagian syair mana yang menyiratkan rekursifitas itu!

Daya guna rekursif terletak pada kemampuannya mendefinisikan sekumpulan objek yang tidak terbatas dengan sebuah pernyataan terbatas [WIR76]. Sejumlah perhitungan yang tidak berhingga misalnya, dapat digambarkan dengan algoritma rekursif (teks algoritma adalah terbatas), seolah-olah algoritma tersebut mengandung pengulangan yang tidak "tampak" secara eksplisit.

Seperti yang sudah dinyatakan pada bagian awal Bab 18 ini, rekursif merupakan kakas yang sangat penting di dalam matematika dan pemrograman. Dalam matematika, banyak ditemukan fungsi yang rekursif, misalnya polinom *Chebysev* berikut:

$$T(n,x)=\begin{cases}1, & n=0\\ x, & n=1\\ 2xT(n-1,x)-T(n-2,x), & \textit{lainnya}\end{cases}$$

Berikut ini diberikan contoh-contoh fungsi rekursif yang lain:

1. Sembarang fungsi rekursif:
   (a) $F(x) = 0$, $x = 0$
   (b) $F(x) = 2F(x - 1) + x2$, $x \neq 0$

[1]*Taufiq Ismail Membaca Puisi*, Taman Ismail Marzuki, 30-31 Januari 1980, hlm. 23

2. Turunan fungsi:

   (a) $\frac{d(x^2)}{dx} = 2x, \quad \frac{d(5x)}{dx} = 5$

   (b) $\frac{d(x^2+5x)}{dx} = \frac{d(x^2)}{dx} + \frac{d(5x)}{dx}$

3. Bilangan asli:
   (a) 1 adalah bilangan asli
   (b) suksesor bilangan asli adalah bilangan asli

4. Polinom interpolasi Newton $p_n(x)$ yang melalui titik-titik (x_0,y_0), (x_1,y_1), ..., (x_n,y_n) didefinisikan sebagai berikut:
   (a) $p_0(x) = y_0$
   (b) $p_n(x) = p_{n-1}(x) + a_n(x - x_0)(x - x_1)...(x - x_{n-1})$

5. Dalam pemrograman, salah satu struktur data yang penting adalah pohon biner (*binary tree*). Sebuah pohon biner didefinisikan sebagai berikut:
   (a) kosong adalah pohon (disebut juga pohon kosong)
   (b) Jika t_1 dan t_2 pohon, maka  adalah pohon

Jika diperhatikan dari contoh-contoh di atas adalah maka definisi rekursif disusun oleh dua bagian:

(i) *Basis*
Bagian yang berisi kasus yang terdefinisi secara eksplisit. Bagian ini juga sekaligus menghentikan rekursif (dan memberikan sebuah nilai yang terdefinisi pada fungsi rekursif).

(ii) *Rekurens*
Bagian ini mendefinisikan objek dalam terminologi dirinya sendiri.

Bagian (b) menyatakan bahwa definisi rekursif memungkinkan komputasi yang tidak berhenti. Pada setiap kali pendefinisian, akan dihasilkan "bentuk" yang makin sederhana, sehingga pada suatu saat pendefinisian itu akan berhenti. Bagian (a) berisi kasus yang menghentikan pendefinian rekursif.

Tinjau kembali perhitungan $n!$ secara rekursif. Dengan mengingat kembali definisi rekursif dari faktorial:

(i) $n! = 1$, jika $n = 0$ { basis }
(ii) $n! = n \times (n-1)!$, jika $n > 0$ { rekurens }

maka 5! dihitung dengan langkah berikut:

(1) $5! = 5 \times 4!$ (rekurens)
(2) $4! = 4 \times 3!$
(3) $3! = 3 \times 2!$
(4) $2! = 2 \times 1!$
(5) $1! = 1 \times 0!$
(6) $0! = 1$

Pada baris (6) kita memperoleh nilai yang terdefinisi secara langsung dan bukan faktorial dari bilangan lainnya. Dengan melakukan runut-balik (*backtrack*) dari baris (6) ke baris (1), kita mendapatkan nilai pada setiap baris untuk menghitung hasil pada baris sebelumnya:

(6') $0! = 1$
(5') $1! = 1 \times 0! = 1 \times 1 = 1$
(4') $2! = 2 \times 1! = 2 \times 1 = 2$
(3') $3! = 3 \times 2! = 3 \times 2 = 6$
(2') $4! = 4 \times 3! = 4 \times 6 = 24$
(1') $5! = 5 \times 4! = 5 \times 24 = 120$

Jadi, $5! = 120$.

Algoritma rekursif untuk menghitung $n!$ sebagai berikut:

```
function Fak(input n:integer)→integer
{ mengembalikan nilai n!. Algoritma rekursif.
  basis    : jika n = 0, maka 0! = 1
  rekurens: jika n > 0, maka n! = n * (n-1)!
}
DEKLARASI
    -
ALGORITMA:
    if n = 0 then
       return 1                          { basis }
    else
       return n* Faktorial(n - 1)        { rekurens }
    endif
```

Algoritma 18.3 Fungsi Faktorial (Rekursif)

Perhatikanlah bahwa dalam bagian rekurens terkandung pengulangan yang implisit. Pengulangan itu ditunjukkan oleh pemanggilan kembali fungsi `Fak` dengan parameter yang nilainya terus berkurang, $(n - 1)$. Bagaimana pemanggilan rekursif dilakukan adalah tanggung jawab *compiler*. Pemrogram hanya perlu memikirkan bagaimana skema algoritma rekursifnya. Fungsi `Fak` di atas memberikan kita contoh skema algoritma rekursif yang paling sederhana, yaitu rekursif lanjar (*linier recursion*), karena hanya ada satu pemanggilan rekursif.

18.3 Skema Umum Prosedur dan Fungsi Rekursif

Algoritma rekursif adalah algoritma yang memanggil dirinya sendiri. Oleh karena itu, algoritma rekursif harus dinyatakan dalam prosedur atau fungsi, karena hanya prosedur dan fungsi yang dapat dipanggil dalam sebuah program. Ingatlah kembali bahwa fungsi mengembalikan sebuah nilai sedangkan prosedur menghasilkan efek *netto*.

Fungsi faktorial Fak di atas memperlihatkan skema fungsi rekursif yang paling dasar. Secara umum, skema dasar fungsi rekursif sebagai berikut:

```
function F(input x : x_type) → function_type;
{ mengembalikan nilai F }

DEKLARASI
    y : y_type

ALGORITMA:
  if T(x) then
     return N(x)          { basis }
  else
     R1(x)
     return F(g(x))       { rekurens }
     R2(x)
  endif
```

Algoritma 18.4 Skema Fungsi Rekursif

Peubah *x* adalah parameter pemanggilan prosedur (dapat berupa parameter *by value* atau parameter *by reference*), R1 dan R2 masing-masing adalah prosedur (atau sekumpulan instruksi) yang dipanggil sebelum dan sesudah pemanggilan rekursif, g adalah fungsi yang mengubah nilai parameter x, dan T adalah fungsi untuk kasus basis, dan N adalah fungsi jika kasus basis tercapai. Perhatikanlah bahwa R1 dan R2 tidak selalu harus ada.

Skema prosedur rekursif ditulis dengan cara yang sama seperti pendefinisian fungsi rekursif, yaitu:

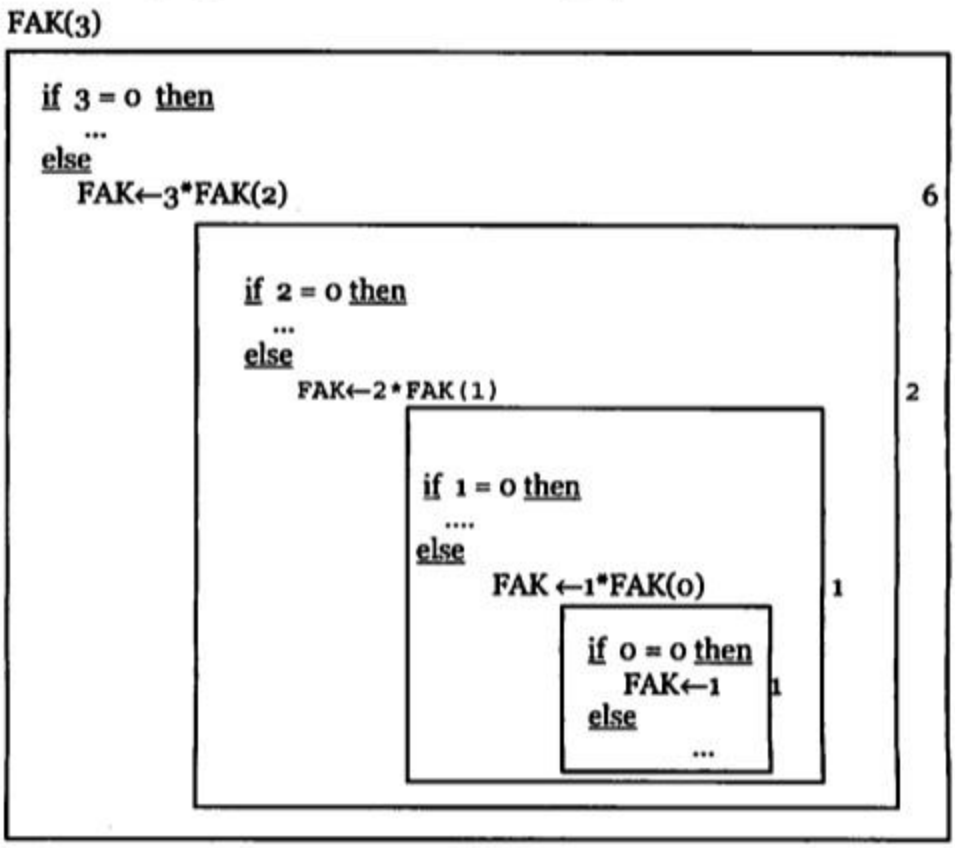
```
procedure C(input x : x_type)
{ prosedur yang memanipulasi nilai x
  K.Awal : x terdefinisi nilainya
  K.Akhir: sekumpulan aksi yang memanipulasi nilai x dilakukan
}
DEKLARASI
   { kamus lokal, jika ada }
ALGORITMA:
  if P(x) then
     M(x)          { basis }
  else
```

```
    S1(x)
    C(F(x))   { rekurens }
    S2(x)
endif
```

Algoritma 18.5 Skema Prosedur Rekursif

Untuk memahami bagaimana algoritma rekursif dilaksanakan (*execute*), tinjau kembali fungsi `Fak` di atas. Gambar 18.3 dapat membantu menjelaskan bagaimana rekursifitas pada pemanggilan `Fak(3)`. Setiap kali `Fak` dipanggil, sebuah "kotak" dibangun. "Kotak" berisi salinan (*copy*) fungsi `Fak` itu sendiri tetapi dengan nilai *n* yang telah berkurang satu. Angka yang terletak di sebelah kanan tiap kotak menyatakan nilai kembalian (*return value*) setiap kali fungsi rekursif selesai dilaksanakan (mencapai kasus basis).

Dalam merancang algoritma rekursif, langkah pertama yang harus dilakukan adalah menyatakan persoalan secara rekursif, lalu menentukan kasus yang menyebabkan pemanggilan rekursif berhenti (basis) dan kasus untuk rekurens-nya. Contoh-contoh di bawah ini diharapkan memberi pemahaman cara merancang algoritma rekursif untuk berbagai persoalan.



Gambar 18.2 Ilustrasi eksekusi fungsi FAK

Contoh 18.1. Perpangkatan a^n (a bilangan bulat dan $n > 0$)

(i) Nyatakan a^n dalam ungkapan rekursif

$a^n = a \times a \times a \times ... \times a$ (sebanyak n kali)

$= a \times a^{n-1}$ (rekurens)

(ii) Tentukan kasus eksplisit yang tidak memerlukan pemanggilan rekursif lagi (basis)

$a^n = 1$ jika $n = 0$

Jadi, secara matematis, fungsi rekursif untuk perpangkatan adalah:

$a^n =$

(i) 1 , jika $n = 0$ { basis }

(ii) $a \times a^{n-1}$, jika $n > 0$ { rekurens }

Algoritmanya:

```
function Pangkat(input a, n : integer) → integer
{ mengembalikan nilai a^n , n > 0
  basis     : a^n = 1 jika n = 0
  rekurens  : a^n = a x a^(n-1)
}
DEKLARASI
   -

ALGORITMA:
   if n = 0 then
      return 1
   else
      return a * Pangkat(a, n - 1)
   endif
```

Algoritma 18.6 Fungsi Menghitung a^n

Contoh 18.2. Fungsi *Chebysev*

$T(n, x) =$

(i) 1 , jika $n = 0$ { basis }

x , jika $n = 1$ { basis }

(ii) $2xT(n-1, x) - T(n-2, x)$, jika $n > 1$ { rekurens }

```
function T(input n : integer, input x : real) → real
{ mengembalikan nilai T(n,x)
  basis : T(n,x) = 1 jika n = 0
          T(n,x) = x jika n = 1
  rekurens : T(n,x) = 2*x*T(n-1,x) - T(n-2,x) untuk n > 1
}

DEKLARASI
   -

ALGORITMA:
   if n = 0 then
      return 1          { basis }
```

```
  else
    if n = 1 then
      return x                    { basis }
    else
      return  2 * x * T(n - 1, x) - T(n - 2, x)   { rekurens }
    endif
  endif
```

Algoritma 18.7 Fungsi *Chebysev*

Contoh 18.3. Menghitung $a \times b$ untuk a dan b bilangan bulat positif

$$a \times b = a + a + \ldots + a \quad \text{(sebanyak } b \text{ kali)}$$
$$= a + (a \times (b-1))$$

Dengan demikian,

$a \times b =$

(i) a , jika $b = 1$ { basis }

(ii) $a + (a \times (b-1))$, jika $b > 1$ { rekurens }

```
function Kali(input a, b : integer)→integer
{ mengembalikan nilai a*b , a dan b > 0
  basis     : a * b = a jika b = 1
  rekurens  : a * b = a + a * (b-1) jika b > 1
}
DEKLARASI
  -

ALGORITMA:
  if b = 1 then
    return a                      { basis }
  else
    return a + Kali(a, b - 1);    { rekurens }
  endif
```

Algoritma 18.8 Fungsi Menghitung a * b

Contoh 18.4. Membalikkan urutan angka-angka di dalam bilangan bulat n. Misalya 4567 dibalikkan menjadi 7654. Gagasan algoritma untuk persoalan ini adalah (secara iteratif):

- Harus dicetak nilai satuan (7) terlebih dahulu. Ini diperoleh dengan cara membagi n dengan 10 dan mengambil sisanya. Jadi, 4567 div 10 = 456 dan 4567 mod 10 = 7.
  Cetak 7.

- Bagi 456 dengan 10 dan cetak sisanya. Jadi, 456 div 10 = 45 dan 456 mod 10 = 6.
  Cetak 6.

- Bagi 45 dengan 10 dan cetak sisanya. Jadi, 45 div 10 = 4 dan 45 mod 10 = 5. Cetak 5.
- Cetak 4

Hasil cetakan: 7654

Algoritma iteratif untuk persoalan ini sebagai berikut:

```
procedure BalikAngka(input n : integer)
{ membalikkan urutan bilangan bulat n
  K.Awal : nilai n sudah terdefinisi
  K.Akhir: urutan angka bilangan n dicetak terbalik
}

DEKLARASI
   -

ALGORITMA:
   while i >= 10 do
      write(n mod 10)
      n ← n div 10
   endwhile
   { i > 10 }
   write(n)
```

Algoritma 18.9 Membalikkan Urutan Angka Bilangan Bulat (Iteratif)

Dengan uraian yang sama seperti di atas, kita juga dapat membuat versi algoritma rekursifnya, yaitu:

BalikAngka (n):
(i) jika $n < 10$, cetak (n) { basis }
(ii) jika $n > 10$ maka { rekurens }
- cetak (n mod 10)
- BalikAngka(n div 10)

Algoritma rekursif untuk persoalan ini sebagai berikut:

```
procedure BalikAngka(input n:integer)
{ membalikkan urutan bilangan bulat n
  K.Awal : nilai n sudah terdefinisi
  K.Akhir: urutan angka bilangan n dicetak terbalik

     basis: jika n < 10, cetak(n)
     rekurens: jika n > 10 maka
                    - cetak(n mod 10)
                    - BalikAngka(n div 10)
}

DEKLARASI
   -

ALGORITMA:
   if n < 10 then
```

```
    write(n)          { basis }
  else                { rekurens }
    write(n mod 10);
    BalikAngka(n div 10);
  endif
```

Algoritma 18.10 Membalikkan Urutan Angka Bilangan Bulat (Rekursif)

Contoh 18.5. Membalikkan urutan huruf di dalam sebuah kata. Misalnya, kata 'BARU' dibalikkan menjadi 'URAB'. Algoritma ini mirip dengan Contoh 18.4.

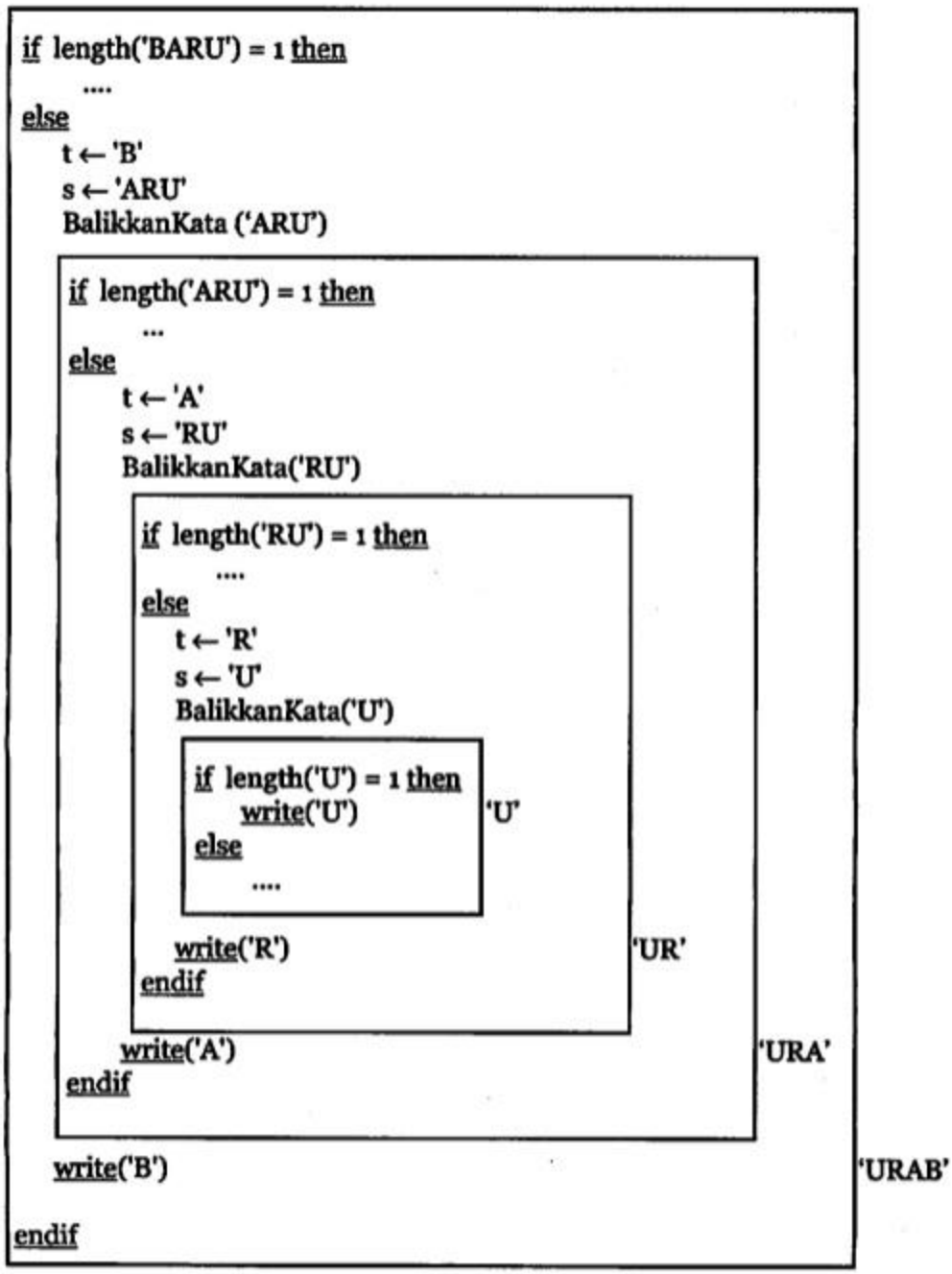
BalikkanKata(*s*):
(i) jika panjang(*s*) = 1, cetak(*s*) { basis }
(ii) jika panjang(*s*) > 1, { rekurens }
- ambil huruf pertama *s*
- BalikkanKata(sisa *s*)
- sambungkan dengan huruf yang diambil tadi

```
procedure BalikkanKata(input s : string)
{ membalikkan kata s
  K.Awal  : string s terdefinisi nilainya
  K.Akhir : string s dicetak terbalik

  basis: jika panjang(s)=1, cetak(s)
  rekurens: jika panjang(s) > 1
              - ambil huruf pertama s
              - BalikKata(sisa s)
              - sambungkan dengan huruf yang diambil tadi
}

DEKLARASI
    t : char

ALGORITMA:
    if length(s) = 1 then      { basis }
      write(s)
    else
      { rekurens }
      t ← HurufPertama(s)      { ambil satu huruf pertama dari s,
                                 simpan huruf itu di t }
      s ← Sisa(s)              { simpan sisa huruf lainnya di s }
      BalikkanKata(s)  { balikkan sisa katanya }
      write(t)         { sambungkan dengan t }
    end
```

Algoritma 18.11 Membalikkan Urutan Huruf di Dalam Kata

Gambar 18.3 dapat membantu menjelaskan bagaimana rekursifitas pada pemanggilan `BalikkanKata('BARU')`.

BalikkanKata('BARU'



Gambar 18.3 Ilustrasi eksekusi fungsi BalikkanKata

Contoh 18.6. Mencari pembagi bersama terbesar (*greatest common divisor - gcd*), x dan y ($x \geq y$).
Contoh: $gcd(4, 0) = 4$
$gcd(36, 24) = 12$
$gcd(20, 10) = 10$,

Algoritma Euclidean untuk menemukan $gcd(x, y)$:
(a) apakah $y = 0$? Jika $y = 0$, algoritma selesai; x adalah jawabannya, tetapi jika tidak, lanjutkan ke langkah (b)
(b) bagilah x dengan y, misalkan r adalah sisanya
(a) ganti nilai x dengan y, nilai y dengan nilai r, ulangi langkah (a)

Versi iteratif algoritma *Euclidean* adalah:

```
function gcd(input x, y : integer) → integer
{ mengembalikan gcd(x,y), dengan syarat x ≥ y }

DEKLARASI
  r : integer

ALGORITMA:
  while y ≠ 0 do
     r ← x mod y
     x ← y
     y ← r
  endwhile
  { y = 0 }

  return x
```

Algoritma 18.12 Menghitung pembagi bersama terbesar (iteratif)

Algoritma *Euclidean* lebih alami apabila dinyatakan secara rekursif, karena hanya membutuhkan satu kali evaluasi x mod y.

gcd (x, y) =
(i) x , jika $y = 0$ { basis }
(ii) gcd(y, x mod y) , jika $y > 0$ { rekurens }

Sebagai contoh, gcd(24, 9) = gcd(9, 6) = gcd(6, 3) = gcd(3, 0) = 3

Versi rekursif algoritma Euclidean adalah:

```
function gcd(input x, y : integer) → integer
{ mengembalikan gcd(x,y)
  basis: gcd(x, y) = a jika y = 0
  rekurens: gcd(x, y) = gcd(y, x mod y) jika y > 0
}

DEKLARASI

ALGORITMA:
   if y = 0 then
      return x                    {basis}
```

```
  else
     return gcd(y, x mod y)   {rekurens}
  endif
```

Algoritma 18.13 Menghitung pembagi bersama terbesar (Rekursif)

Contoh 18.7. Mengevaluasi polinom dengan cara perkalian bersarang (metode *Horner*):

$$p_n(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$$
$$= a_0 + x(a_1 + x(a_2 + x(a_3 + ... + x(a_{n-1} + a_nx))...)))$$

$p(a, x, i, n) =$

(i) a_n , jika $i = n$ { basis }

(ii) $a_i + x * p(a, x, i+1, n)$, jika $i \neq n$ { rekurens }

```
function p(input a : koef_poli, input x, i, n : integer) → real
{ mengembalikan nilai p(x) = a0 + a1x + a2x^2 + ... + anx^n
    basis: jika i = n, maka p(a,x,i,n) = a[n]
    rekurens:jika i <> n, maka p(a,x,i,n) = ai + x*p(a,x,i+1,n)
}
DEKLARASI
  -

ALGORITMA:
  if  i = n then
     return a[n]                 { basis }
  else
     return a[i] + x * p(a, x, i + 1, n)     { rekurens }
  endif
```

Algoritma 18.14 Mengevaluasi nilai polinom

Contoh pemanggilan fungsi p (misalkan untuk polinom derajat $n = 100$):

```
read(x)
write(p(a,x,0,100))
```

Perhatikanlah bahwa pada fungsi p di atas, larik koefisien polinom a, nilai x, dan n tidak pernah berubah selama pemanggilan. Kita dapat menghemat jumlah parameter dengan menghilangkan parameter yang tidak berubah tersebut, menjadi:

$p(i) =$

(i) a_n , jika $i = n$ { basis }

(ii) $a_i + x * p(i+1)$, jika $i \neq n$ { rekurens }

yang dalam hal ini, larik a dan n bersifat sebagai peubah (*variable*) global.

Catatan:
semakin banyak jumlah parameter pada fungsi/prosedur rekursif, semakin besar memori yang dibutuhkan pada setiap kali pemanggilannya. Hal ini akan dijelaskan pada bagian lain.

```
function p(input i : integer) → real
{ mengembalikan nilai p(x) = a0 + a1x + a2x^2 + ... + anx^n
    basis: jika i = n, maka p(a,x,i,n) = a[n]
    rekurens:jika i <> n, maka p(a,x,i,n) = ai + x*p(a,x,i+1,n)
}

DEKLARASI
  -

ALGORITMA:
  if  i = n then
      return a[n]              { basis }
  else
      return a[i] + x * p(i+1);     { rekurens }
  endif
```

Algoritma 18.15 Mengevaluasi nilai polinom dengan jumlah parameter lebih sedikit

Contoh pemanggilan fungsi p (untuk polinom derajat $n = 100$ elemen):

```
read(x)
write(p(0))
```

Atau, kalau kita tetap ingin mempertahankan a, x, dan n sebagai parameter fungsi, kita dapat membuat fungsi rekursif *dua-tingkat* sebagai berikut:

```
function p(input i : integer) → real
{ mengembalikan nilai p(x) = a0 + a1x + a2x^2 + ... + anx^n
    basis: jika i = n, maka p(a,x,i,n) = a[n]
    rekurens:jika i <> n, maka p(a,x,i,n) = ai + x*p(a,x,i+1,n)
}

DEKLARASI
  -

ALGORITMA:
  if i = n then
      return a[n]              { basis }
  else
      return a[i] + x*p(i+1)   { rekurens }
  endif
```

```
function Poli(input a:koef_poli, input x, n : integer) → real
{ fungsi tingkat pertama }

DEKLARASI
    y     : real
    function p(input i:integer)→real
    { mengembalikan nilai p(x) = a0 + a1x + a2x^2 + ... + anx^n

ALGORITMA:
  y ← p(0)
  write(y)
```

Algoritma 18.16 Fungsi rekursif dua tingkat

Contoh pemanggilan fungsi (misalkan untuk polinom derajat $n = 100$ elemen):

```
read(x)
write(Poli(a, x, 0, 100))
```

Contoh 18.8. Mencari indeks larik yang elemennya = x. Pencarian dilakukan dengan metode pencarian beruntun (*sequential search*).

Cari (A, n, x, j) =

(i) j	, jika $A[j] = x$	{ x ditemukan }	{ basis }
-1	, jika $j > n$	{ x tidak ditemukan}	{ basis }
(ii) Cari $(A, n, x, j+1)$	, jika $j \leq n$		{ rekurens }

```
function Cari(input A : LarikInt, input n, x, j : integer)→integer
{ j berisi indeks elemen larik yang dicari; atau j=0 jika x tidak
        ditemukan
     basis: cari = j, jika A[j]=x
                   cari = 0, jika j > n
   rekurens: jika j<=n, maka cari = cari(A,n,x,j+1)
}
DEKLARASI
  -
ALGORITMA:
   if A[j] = x then
      return j       { basis }
   else
      if j > n then
        return -1    { basis }
      else
        return cari(A, n, x, j+1)  { rekurens }
      endif
   endif
```

Algoritma 18.17 Pencarian di dalam larik Integer

Contoh pemanggilan fungsi `cari` (misalkan untuk larik A yang berukuran 100 elemen):

```
read(x)
write(Cari(A, 100, x, 1))
```

Seperti pada Contoh 18.7, parameter A, n, dan x tidak berubah selama pemanggilan. Oleh karena itu, kita dapat membuat fungsi rekursifnya menjadi lebih mangkus sebagai berikut:

Cari (j) =

(i) j	, jika $A[j] = x$	{ x ditemukan}	{ basis }
-1	, jika $j > n$	{ x tidak ditemukan }	{ basis }
(ii) Cari $(j+1)$	, jka $j \leq n$		{ rekurens }

```
function cari(j : integer) → integer
{ j berisi indeks elemen larik yang dicari; j=0 jika x tidak
ditemukan.
basis: cari = j, jika A[j] = x
       cari = -1, jika j > n
rekurens: jika j <= n, maka cari = cari(A, n, x, j+1)
}

DEKLARASI
  -

ALGORITMA:
    if A[j] = x then
       return j          { basis }
    else
       if j > n then
          return 0       { basis }
       else
          return Cari(j+1)     { rekurens }
       endif
    endif
```

Algoritma 18.18 Pencarian di dalam larik Integer (jumlah parameter lebih sedikit)

Cobalah Anda ubah fungsi cari di atas menjadi fungsi rekursif dua-tingkat.

Contoh 18.9. Mencari nilai maksimum dalam larik *integer* $A[1..n]$

Maks(A, n) =

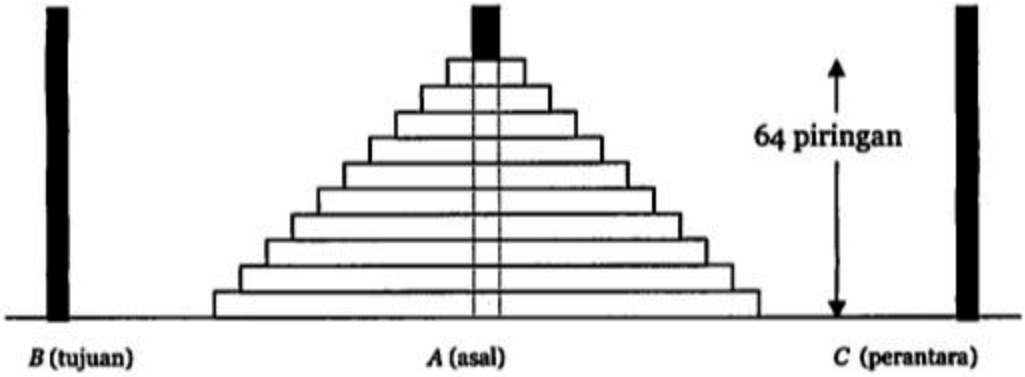
(i) $A[1]$	, jika $n = 1$	{ basis }
$A[n]$	, jika $A[n] >$ Maks$(A, n-1)$]	{ basis }
(ii) Maks$(A, n-1)$	, jika Maks$(A, n-1) > A[n]$	{ rekurens }

```
function Maks(input A : Larik, input n : integer) → integer
{ mengembalikan nilai maksimum larik A[1..n]
        basis: jika n = 1, maka maks(A,n) = A[1]
rekurens: jika n > 1 maka
                jika A[n] > maks(A,n-1) maka maksA(n)=A[n]
                kalau tidak, maks(A,n)=maks(A,n-1)
}

DEKLARASI
  -

ALGORITMA:
   if n = 1 then
      return A[1]                {basis}
   else
      if A[n] > Maks(A,n-1) then
         return A[n]             {basis}
      else
         return Maks(A,n-1)      {rekurens}
      endif
   endif
```

Algoritma 18.19 Mencari nilai maksimum di dalam larik

Di sini, parameter A tidak pernah berubah selama pemanggilan. Modifikasilah fungsi maks tadi dengan menghilangan parameter yang tidak berubah tersebut.

Contoh 18.10. Menara Hanoi (*Tower of Hanoi*). Contoh ini merupakan legenda klasik pendeta Budha. Dikisahkan bahwa di kota Hanoi, Vietnam, terdapat tiga buah tiang tegak setinggi 5 meter dan 64 buah piringan (*disk*) dari berbagai ukuran (Gambar 18.4). Tiap piringan mempunyai lubang di tengahnya yang memungkinkannya untuk dimasukkan ke dalam tiang. Pada mulanya piringan tersebut tersusun pada sebuah tiang sedemikian rupa sehingga piringan yang di bawah mempunyai ukuran lebih besar daripada ukuran piringan di atasnya. Pendeta Budha memberi pertanyaan kepada muridnya: bagaimana memindahkan seluruh piringan tersebut ke sebuah tiang yang lain; setiap kali hanya satu piringan yang boleh dipindahkan, tetapi tidak boleh ada piringan besar di atas piringan kecil. Tiang yang satu lagi dapat dipakai sebagai tempat peralihan dengan tetap memegang aturan yang telah disebutkan. Menurut legenda pendeta Budha, bila pemindahan seluruh piringan itu berhasil dilakukan, maka dunia akan kiamat!



Gambar 18.4 Menara Hanoi

Berdasarkan aturan yang ditetapkan oleh pendeta Budha, maka kita harus memindahkan piringan paling bawah terlebih dahulu ke tiang *B* sebagai alas bagi piringan yang lain. Untuk mencapai maksud demikian, berpikirlah secara rekursif: andaikan kita mengangkat 63 piringan teratas dari *A* ke *C*, lalu pindahkan piringan paling bawah dari *A* ke *B*, lalu angkat 63 piringan dari *C* ke *B*.

angkat 63 piringan dari A ke C
pindahkan 1 piringan terbawah dari A ke B
angkat 63 piringan dari C ke B

Selanjutnya dengan tetap berpikir rekursif-pekerjaan mengangkat 63 piringan dari sebuah tiang ke tiang lain dapat dibayangkan sebagai mengangkat 62 piringan antara kedua tiang tersebut, lalu memindahkan piringan terbawah dari sebuah tiang ke tiang lain, begitu seterusnya.

Prosedur rekursif untuk memindahkan n buah piringan dari A ke B adalah:

Hanoi(n, A, B, C)

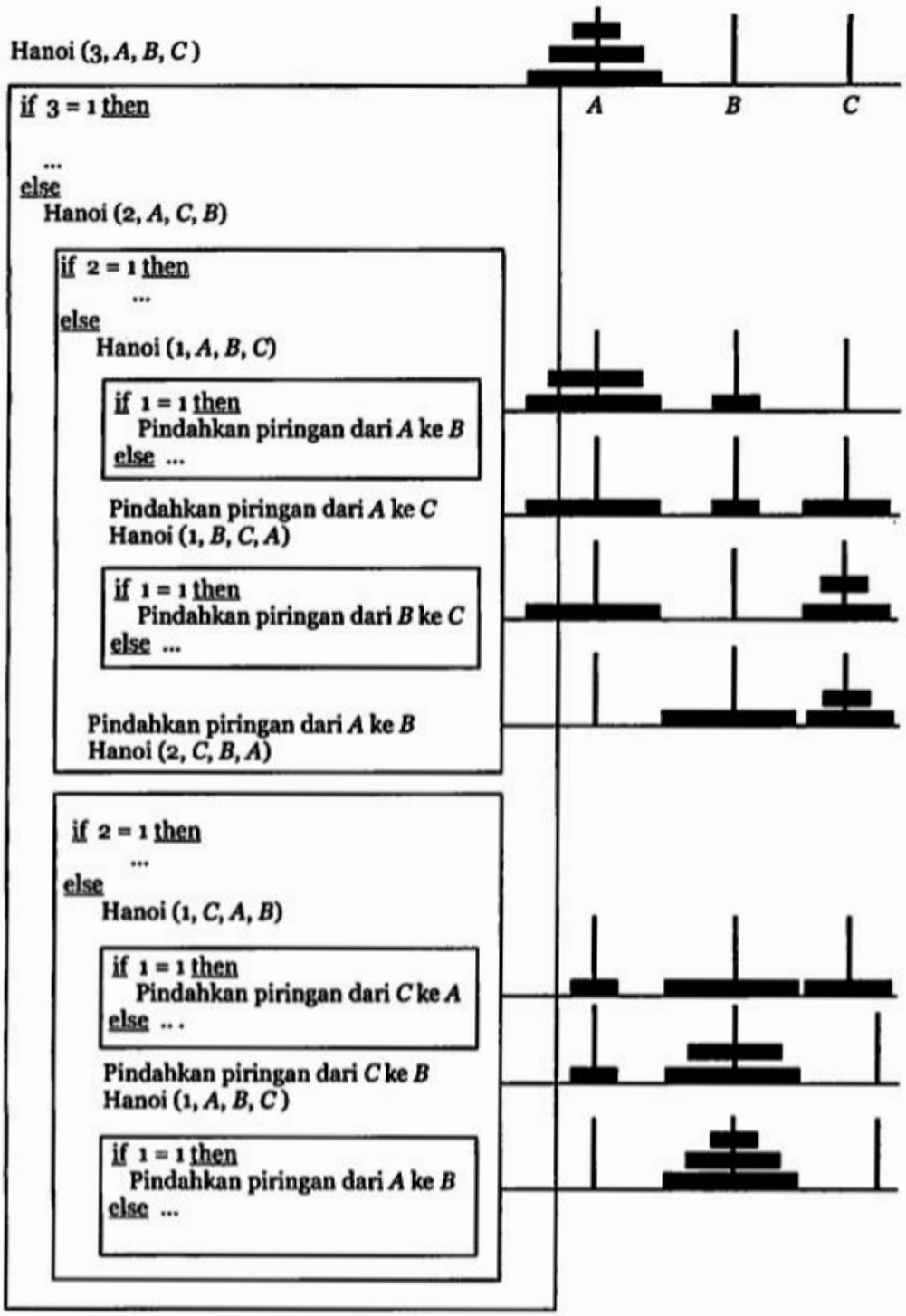
(a) basis:
jika $n = 1$, pindahkan piringan dari A ke B

(b) rekurens:
jika $n > 1$,
- Hanoi($n - 1$, A, C, B)
- pindahkan 1 piringan dari A ke B
- Hanoi($n - 1$, C, B, A)

```
procedure Hanoi(input n, A, B, C : integer)
{ memindahkan n buah piringan dari A ke B
  K.Awal : N adalah jumlah piringan, harganya sudah terdefinisi
           Pada mulanya seluruh piringan berada di tiang A,
           piringan terbesar berada paling bawah.
  K.Akhir: Seluruh piringan telah pindah ke tiang B, piringan
           paling besar berada paling bawah. Prosea perpindahan
           piring dicetak ke layar

           basis:
              jika n = 1, pindahkan piringan dari A ke B
    rekurens:
              jika n > 1,
                 - angkat n-1 buah piringan dari A ke C
                 - pindahkan 1 piringan dari A ke B
                 - angkat n-1 buah piringan dari C ke B
}

DEKLARASI
   -

ALGORITMA:
     if n = 1 then
        write('pindahkan piringan dari ',A,' ke ', B)
     else
        Hanoi(N-1,A, C, B)
        write('Pindahkan piringan dari ',A,' ke ', B)
        Hanoi(N-1, C, B, A)
     endif
```

Algoritma 18.20 Menara Hanoi

Contoh ilustrasi eksekusi prosedur Hanoi untuk $N = 3$ piringan diperlihatkan pada Gambar 18.5

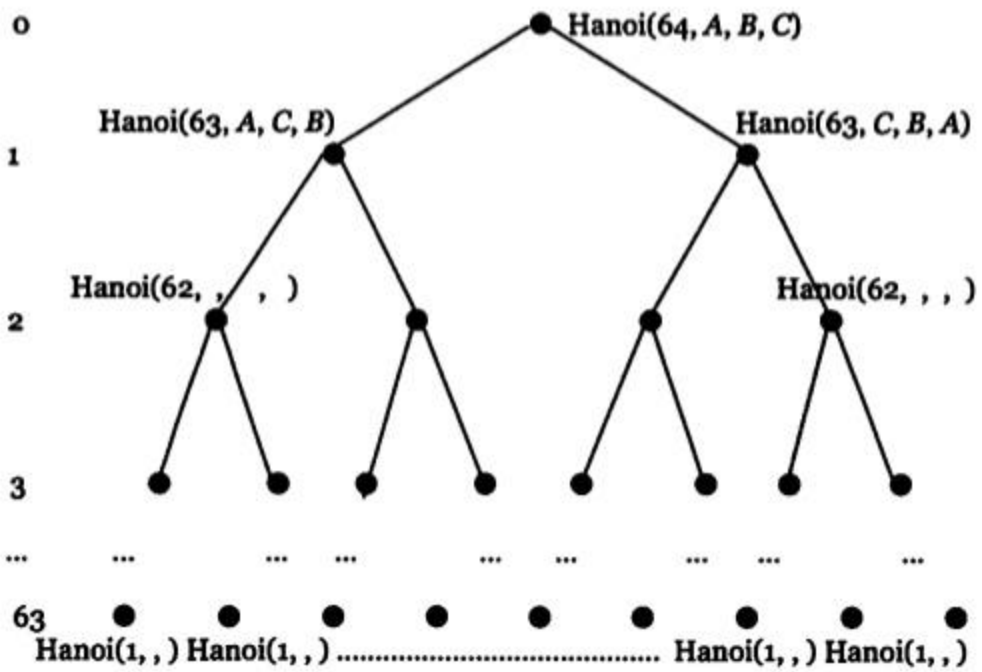


Gambar 18.5 Ilustrasi eksekusi prosedur Hanoi

TEOREMA 18.1. Untuk $n = 64$, *procedure* Hanoi dipanggil sebanyak $2^{64} - 1$ kali.

Bukti:
Berdasarkan prosedur Hanoi di atas, perhatikanlah bahwa untuk $n > 1$ akan ada 2 buah pemanggilan prosedur Hanoi. Pada pemangilan `Hanoi(64,A,B,C)`, nilai n semula = 64. Setiap kali terjadi pemanggilan rekursif, nilai n berkurang 1. Pemanggilan rekursif akan berakhir bila $n = 1$.

Pemanggilan rekursif ini dapat dinyatakan sebagai pohon pemanggilan (Gambar 18.6). Setiap simpul (digambarkan dengan lingkaran) menyatakan pemanggilan prosedur Hanoi. Setiap simpul akan memanggil dua kali prosedur Hanoi.



Gambar 18.6 Pohon pemanggilan prosedur Hanoi

Jumlah seluruh simpul di pohon adalah

$$S = 1 + 2 + 4 + 8 + ... + 2^{63}$$

yang merupakan deret geometri dengan jumlahnya adalah

$$S = 2^{64} - 1$$

Jadi, untuk $n = 64$, prosedur Hanoi dipanggil sebanyak $2^{64} - 1$ kali. Setiap kali pemanggilan *procedure* Hanoi, hanya satu buah piringan yang dipindahkan. Ini berarti terdapat $2^{64} - 1$ buah pemindahan piringan. Jika orang dapat memindahkan seluruh priringan itu dan setiap kali pemindahan piringan membutuhkan waktu satu detik, juga dengan andaian ia tidak pernah melakukan kesalahan, maka waktu yang dibutuhkan untuk memindahkan seluruh piringan adalah $2^{64} - 1$ detik atau setara dengan 600.000.000.000 tahun! Karena itu, legenda yang menyatakan bahwa dunia akan kiamat bila orang berhasil memindahkan 64 piringan di menara Hanoi ada juga benarnya, karena 600.000 000.000 tahun adalah waktu yang sangat lama, dunia semakin tua, dan akhirnya hancur. *Wallahualam*!.

Prosedur Hanoi di atas dapat dibuat menjadi lebih elegan dengan menuliskan pernyataan "pindahkan satu piring" cukup sekali saja di dalam teks algoritmanya:

Hanoi(n, A, B, C):
(i) Jika $n = 0$, maka (tidak melakukan apa-apa) { basis }
(ii) Jika $n > 0$, { rekurens }
- Hanoi($n - 1, A, C, B$)
- pindahkan 1 piringan dari sumber ke A ke B
- Hanoi($n - 1, C, B, A$)

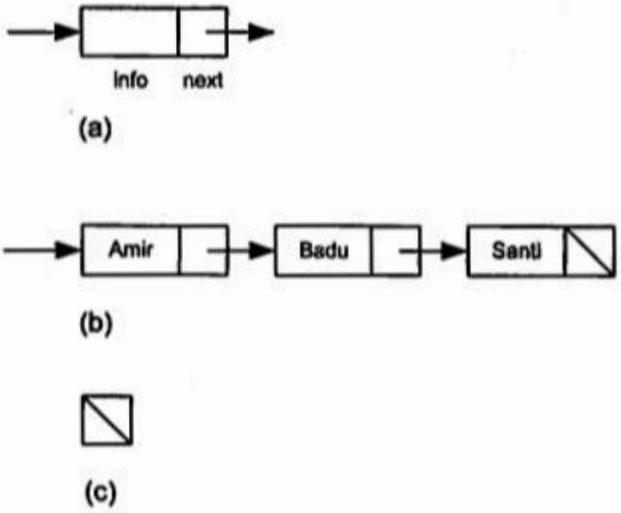
```
procedure Hanoi(input n, A, B, C : integer)
{ memindahkan n buah piringan dari A ke B
  K.Awal : n adalah jumlah piringan, harganya sudah terdefinisi
           Pada mulanya seluruh piringan berada di tiang A,
           piringan terbesar berada paling bawah.
  K.Akhir: Seluruh piringan telah pindah ke tiang B, piringan
           paling besar berada paling bawah. Proses perpindahan
           piring dicetak ke layar

           basis:
             jika n = 1,
                  (tidak melakukan apa-apa)
   rekurens:
               jika n > 1,
                   - angkat n - 1 buah piringan dari A ke C
                   - pindahkan 1 piringan dari A ke B
                   - angkat n - 1 buah piringan dari C ke B
}

DEKLARASI
  -

ALGORITMA:
      if n > 0 then
         Hanoi(n-1, A, C, B)
         write('Pindahkan piringan dari ',A,' ke ', B)
         Hanoi(n-1, C, B, A)
      Endif
```

Algoritma 18.21 Menara Hanoi (versi 2)

18.4 Rekursif dengan *List* Berkait

List berkait (*linked list*) adalah struktur data yang penting dalam pemrograman. *List* berkait adalah sekumpulan elemen (boleh kosong) yang setiap elemennya terdiri atas dua bagian: bagian data (*info*) dan bagian alamat (*next*). Bagian data berisi nilai yang akan dimanipulasi, sedangkan bagian alamat berisi alamat elemen *list* berikutnya. Sebuah *list* berkait dikenali dari alamat elemen pertamanya.

Gambar 18.7(a) adalah gambaran lojik elemen *list*. Anak panah menyatakan elemen berikutnya yang diacu oleh alamat yang disimpan di dalam bagian *next*. Gambar 18.8(b) adalah sebuah *list* berkait yang terdiri atas 3 buah elemen, masing-masing berisi info 'Amir', 'Badu', dan 'Santi'. Elemen terakhir tidak mengacu ke elemen manapun, sehingga bagian *next* diisi dengan konstanta Nil (Nil adalah satu-satunya konstanta yang terdefinisi jika alamat tidak mengacu ke elemen manapun). Nil digambarkan dengan kotak yang diberi garis diagonal. Gambar 18.8(c) adalah penggambran *list* kosong. *List* kosong adalah *list* yang tidak mempunyai sebuah elemen pun. Pembahasan lebih jelas mengenai *list* berkait dapat anda baca di dalam buku-buku tentang struktur data. Buku ini hanya akan membahas aspek rekursif pada *list* berkait.



Gambar 18.7 (a) Gambaran lojik elemen list berkait; (b) Sebuah list berkait dengan 3 buah elemen; (c) List kosong

Notasi algoritmik untuk pendeklarasian *list* terstruktur sederhana sebagai berikut:

```
type List = ↑ListElmt
type ListElmt : record <
                          Info : info_type,
                          Next : List
                        >
```

Algoritma 18.22 Pendefinisian *list* secara rekursif

Perhatikanlah bahwa pendefinisan tipe *list* di atas sudah bersifat rekursif. Hal ini ditunjukkan oleh tipe dari *field* `Next`, yaitu `List`. Definisi rekursif dari *list* berkait adalah:

List dengan tipe *S* adalah salah satu dari:
(i) kosong (Nil) { basis }
(ii) pengkaitan *S* dengan barisan elemen bertipe *S* { rekurens }

Dengan demikian, pemrosesan *list* dapat juga dikerjakan secara rekursif (selain secara beruntun/sekuensial.

Contoh 18.11. Mengunjungi (mencetak) seluruh elemen *list* berkait.
Cara berpikir rekursifnya adalah sebagai berikut: jika *list* kosong, maka tidak ada elemen *list* yang dikunjungi; sebaliknya, jika *list* tidak kosong, maka cetak elemen pertama *list*, kemudian dengan tetap berpikir secara rekursif-cetak *list* yang alamatnya ditunjuk oleh elemen pertama tadi.

CetakList(*L*):
(i) jika *L* = Nil, tidak melakukan apa-apa { basis }

(ii) jika *L* ≠ Nil { rekurens }
- write(Info(*L*))
- CetakList(Next(*L*))

```
procedure CetakList(input L : List)
{ mencetak setiap info elemen list yang dikunjungi
 K.Awal: L berisi alamat elemen pertama list
 K.Akhir: setiap info elemen list dicetak ke layar
          basis :
       jika L = Nil, tidak melakukan apa-apa
          rekurens:
       jika L ≠ Nil
          - write(Info(L))
          - CetakList(Next(L)) }

DEKLARASI
   -

ALGORITMA:
   if L = Nil then
      { tidak melakukan apa-apa -> berhenti }
   else
```

```
        write(Info(L))
        CetakList(Next(L))
    endif
```

Algoritma 18.23 Mencetak seluruh elemen *List*

Misalkan *Info* dan *Next* adalah dua buah fungsi yang memberikan nilai *field Info* dan *field Next* dari elemen *list* yang alamat elemen pertamanya adalah *L*:

```
function Info(input L : List) → info_type
{ mengembalikan nilai L↑.Info }

DEKLARASI
   -

ALGORITMA:
   return L↑.Info
```

```
function Next(input L : List)→ List
{ mengembalikan nilai L↑.Next }

DEKLARASI
   -

ALGORITMA:
    return L↑.Next
```

Algoritma 18.24 Fungsi Info dan *Next*

Prosedur `CetakList` di atas dapat disederhanakan sehingga menjadi lebih elegan karena pada kasus basis tidak aksi yang dilakukan, yaitu:

```
procedure CetakList(input L : List)
{ mencetak setiap info elemen list yang dikunjungi
  K.Awal: L berisi alamat elemen pertama list
  K.Akhir: setiap info elemen list dicetak ke layar
          basis :  jika L = Nil, tidak melakukan apa-apa
          rekurens: jika L ≠ Nil
                      - cetak(Info(L))
                      - CetakList(Next(L))
}

DEKLARASI
   -

ALGORITMA:
   if L ≠ Nil then
       write(Info(L))
       CetakList(Next(L))
   endif
```

Algoritma 18.25 Mencetak seluruh elemen *List* (versi 2)

Contoh 18.12. Menghitung panjang *list*. Yang dimaksud dengan panjang *list* adalah banyak elemen yang terdapat pada *list L*.

Panjang(L) =
(i) 0 , jika L = Nil { basis }
(ii) 1 + Panjang(Next(L)) , jika $L \neq$ Nil { rekurens }

```
function Panjang(input L : List) → integer
{ mengembalikan panjang list S
    basis: jika L = Nil, maka Panjang(L) = 0 jika L = Nil
    rekurens: jika L ≠ Nil, maka Panjang(L) = 1 + Panjang(Next(L))
}

DEKLARASI
   -
ALGORITMA:
   if L = Nil then
      return 0                              { basis }
   else
      return 1 + Panjang(Next(L))           { rekurens }
   endif
```

Algoritma 18.26 Menghitung panjang *List*

Contoh 18.13. Menghapus semua elemen *list*.
HapusList(L):
(i) jika L = Nil, tidak melakukan apa-apa { basis }
(ii) jika $L \neq$ Nil { rekurens }
- HapusList(Next(L))
- dealokasi(L)

Prosedur ini akan menghapus elemen terakhir lebih dahulu, lalu elemen kedua dari akhir, elemen ketiga dari akhir, dan seterusnya, sampai elemen pertama.

```
procedure HapusList(input/output L : List);
{ menghapus semua elemen list
  K.Awal: L berisi alamat elemen pertama list
  K.Akhir: setiap info elemen list dihapus dan memorinya
           dikembalikan ke sistem; L = Nil

        basis : jika L = Nil, tidak melakukan apa-apa
        rekurens: jika L ≠ Nil
                    - HapusList(Next(L))
                    - dealokasi(L)
}

DEKLARASI
  -

ALGORITMA:
   if L ≠ Nil then
```

```
      HapusList(Next(L))
      dealokasi(L)   { bebaskan memori yang dipakai L }
   endif
```

Algoritma 18.27 Menghapus semua elemen *List*

Contoh 18.14. Mencari elemen tertentu di dalam *list*.
Cari(*L*, *x*, *ketemu*):

(i) jika *L* = Nil, maka *ketemu* ← false { basis }
jika Info(*L*) = *x*, maka *ketemu* ← true { basis }
(ii) jika Info(*L*) ≠ *x*, maka Cari(Next(*L*), *x*, *ketemu*) { rekurens }

```
procedure Cari(input L: List, input x : integer, output ketemu :
boolean)
{ mencari apakah x terdapat di dalam list L
  K.Awal: L berisi alamat elemen pertama list, x adalah elemen
          yang dicari, harganya sudah terdefinisi,
  K.Akhir: ketemu bernilai true jika x ada, false jika sebaliknya

  basis:
       jika L = nil, maka ketemu←false
       jika Info(L) = x, maka ketemu←true
  rekurens:
       jika Info(L) ≠ x, maka Cari(Next(L),x,ketemu)
}

DEKLARASI
   -

ALGORITMA:
   if L = Nil then
      ketemu ← false
   else
      if Info(L) = x then
         ketemu ← true
      else
         Cari(Next(L),x, ketemu)
      endif
   endif
```

Algoritma 18.28 Pencarian di dalam *List*

Persoalan pencarian ini lebih tepat dinyatakan sebagai fungsi daripada prosedur. Dengan fungsi, kita dapat menghemat jumlah parameter dengan menghilangkan parameter `ketemu`:

```
function Cari(input L : List, input x : integer) → boolean
{ true jika x terdapat di dalam list S, false jika tidak }

  basis:
       jika L = nil, maka return false
       jika Info(S) = x, maka return true
```

```
  rekurens:
       jika Info(S) ≠ x, maka return Cari(Next(S),x,ketemu)
}

DEKLARASI
   -

ALGORITMA:
   if L = Nil then
      return false
   else
      if Info(L)=x then
         return true
      else
         return Cari(Next(L),x)
      endif
  endif
```

Algoritma 18.29 Pencarian di dalam *List* (sebagai fungsi)

Modifikasilah fungsi `Cari` di atas menjadi fungsi rekursif dua tingkat sehingga parameter `x` dapat dibuang dari fungsi `Cari`.

Contoh 18.15. Misalkan kita ingin membuat salinan (*copy*) *list L1* sehingga diperoleh *list L2* yang tepat sama dengan *list L1*.

SalinList(*L1*, *L2*):
(i) jika *L1* = Nil, maka *L2*←Nil { basis }
(ii) jika *L1* ≠ Nil maka { rekurens }
- alokasimemori(*L2*); Info(*L2*) ← Info(*L1*); Next(*L2*) ← Nil
- SalinList(Next(*L1*), Next(*L2*))

```
procedure SalinList(input L1 : List, output L2 : List)
{ menyalin list L1 ke L2
  K.Awal : list L1 sudah terdefinisi elemennya, L1 mungkin kosong
  K.Akhir: elemen list L2 sama dengan elemen list L1

basis:
   jika L1=Nil, maka L2←Nil
rekurens:
   jika L1 ≠ Nil maka
       - alokasimemori(L2);  Info(L2)←Info(L1);   Next(L2)←Nil
       - SalinList(Next(L1), Next(L2))
}

DEKLARASI
   -

ALGORITMA:
   if L1 = Nil then
      L2 ← Nil
   else
```

```
        AlokMem(L2)    { alokasi memori untuk elemen baru }
        Info(L2) ← Info(L1)
        Next(L2) ← Nil
        SalinList(Next(L1), Next(L2))
    endif
```

Algoritma 18.30 Menyalin *List*

Prosedur rekursif untuk `SalinList` merupakan contoh yang bagus untuk memperlihatkan kenyataan bahwa prosedur rekursif seringkali lebih mudah dibuat daripada versi prosedur iteratifnya. Pada versi iteratif, bila kita menyisipkan elemen baru setelah elemen terakhir (penyisipan di akhir - *InsertLast*), maka kita harus menyimpan alamat elemen terakhir. Persoalan akan muncul pada waktu penyisipan elemen untuk pertama kalinya, karena alamat elemen terakhir belum ada. Karena itu, penyisipan elemen pertama harus diproses secara khusus.

Versi iteratif dari algoritma menyalin *list* sebagai berikut:

```
procedure SalinList(input L1: List, output L2:List)
{ menyalin list L1 ke L2; versi iteratif
  K.Awal : list L1 sudah terdefinisi elemennya, mungkin kosong
  K.Akhir: elemen list L2 sama dengan elemen list L1
}

DEKLARASI
   P : List
   last : List { menyimpan alamat elemen terakhir }

ALGORITMA:
   if L1 = Nil then    { kasus kosong }
      L2 ← Nil
   else
      AlokMem(L2)
      Info(L2) ← Info(L1)
      Next(L2) ← Nil
      last ← L2                  { alamat elemen terakhir L2 }
      L1 ← Next(L1)              { L1 maju }
      while L1 ≠ Nil do
         AlokMem(P)
         Info(P) ← Info(L1)
         Next(P) ← Nil

         { sisipkan P setelah elemen terakhir }
         Next(last) ← P
         last ← P
         L1 ← Next(L1)
      endwhile
      { L1 = Nil }
   endif
```

Algoritma 18.31 Menyalin *List* (Iteratif)

Contoh 18.16. Misalkan kita ingin menyisipkan sebuah elemen yang nilanya = x ke dalam *list* L yang telah terurut membesar. Penyisipan elemen yang baru harus tetap mempertahankan keterurutan *list*. Jika x sudah ada di dalam L, x tidak perlu disisipkan lagi.

Sisip(L, x):
(i) jika L = Nil , maka SisipElemenBaru(L, x, Nil)
jika $L \neq$ Nil, maka
jika Info(L) = x (artinya x sudah ada dalam L), maka
{tidak melakukan apa-apa}
kalau tidak,
jika x < Info(L), maka
SisipElemenBaru(L, x, L)

(ii) jika x > Info(L), maka Sisip(Next(L), x)

```
procedure Sisip(input/output L : List, input x : integer)
{ menyisipkan x ke dalam list L yang terurut membesar
  K.Awal : list L mungkin kosong; x sudah terdefinisi harganya
  K.Akhir: x disisipkan pada posisi yang sesuai; list L tetap
           terurut membesar

  basis:
     jika L = Nil , maka SisipElemenBaru(L,x,Nil)
     jika L ≠ Nil, maka
     jika Info(L) = x (artinya x sudah ada dalam L), maka
        {tidak melakukan  apa-apa}
     kalau tidak,
        jika x < Info(L), maka
           SisipElemenBaru(L,x,L)
  rekurens:
        jika x > Info(L), maka Sisip(Next(L), x)
}

DEKLARASI
  -

ALGORITMA:
   if L = Nil then
      SisipElemenBaru(L,x,Nil)
   else
        if x = Info(L) then
            write(x,' sudah terdapat di dalam list! ')
        else
            if x < Info(L) then
               SisipElemenBaru(L,x,L)
            else
               Sisip(Next(L), x)
            endif
        endif
   endif
```

Algoritma 18.32 Penyisipan elemen di dalam *List* terurut

Prosedur `SisipElemenBaru` sebagai berikut:

```
procedure SisipElemenBaru(input/output L1:List, input x:integer,
                          input/output L2:List)
{ menyisipkan x ke dalam list L1 }

DEKLARASI
   temp : List

ALGORITMA:
   AlokMem(temp)
   Info(temp)← x
   Next(temp)← L2
   L1←temp
```

Algoritma 18.33 Penyisipan elemen baru

18.5 Bagaimanakah Program Rekursif Bekerja?

Memahami cara *compiler* menangani program rekursif dapat membantu kita untuk:

1. meminimumkan jumlah parameter yang dilewatkan pada waktu pemanggilan,
2. mengubah algoritma rekursif menjadi program iteratif dengan cara meniru (simulasi) mekanisme pelaksanaan program rekursif.

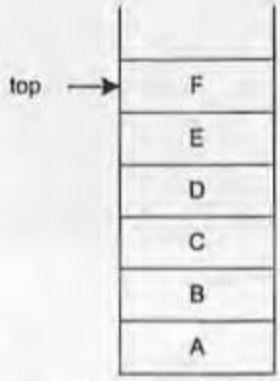
Pertama-tama kita harus mengetahui mekanisme pemanggilan prosedur/fungsi yang ditangani oleh *compiler*. Ketika sebuah prosedur/fungsi dipanggil, *area data* di memori dialokasikan untuk pemanggilan tersebut. Area data ini akan menyimpan:

1. parameter yang dilewatkan waktu pemanggilan.
2. semua peubah lokal di dalam prosedur/fungsi.
3. alamat kembali (*return address*), yaitu alamat instruksi berikutnya di dalam program pemanggil setelah prosedur/fungsi selesai dilaksanakan.

Setelah mengisi data area, kendali program dipindahkan ke prosedur/fungsi. Semua acuan terhadap nilai parameter dan peubah lokal adalah nilai yang terdapat di dalam data area. Bila prosedur/fungsi selesai dilaksanakan, alamat kembalinya diambil dari area data, kendali program bercabang ke alamat tersebut, dan alokasi memori untuk area data dibebaskan.

Tiap *compiler* berbeda-beda dalam mengimplemantasikan area data, tetapi pada umumnya area data diimplementasikan dalam bentuk tumpukan (*stack*). Tumpukan (*stack*) adalah sekumpulan elemen di mana semua penambahan (*insert*) elemen atau penghapusan (*delete*) elemen hanya dapat dilakukan pada sebuah sisi yang disebut *top*. Gambar 18.8 adalah contoh sebuah tumpukan yang berisi 6 buah elemen. Elemen yang paling atas

disebut ditunjuk oleh puncak (*top*). Penambahan elemen baru hanya dapat dilakukan di atas *top*, dan elemen yang dihapus hanyalah elemen pada posisi *top*.

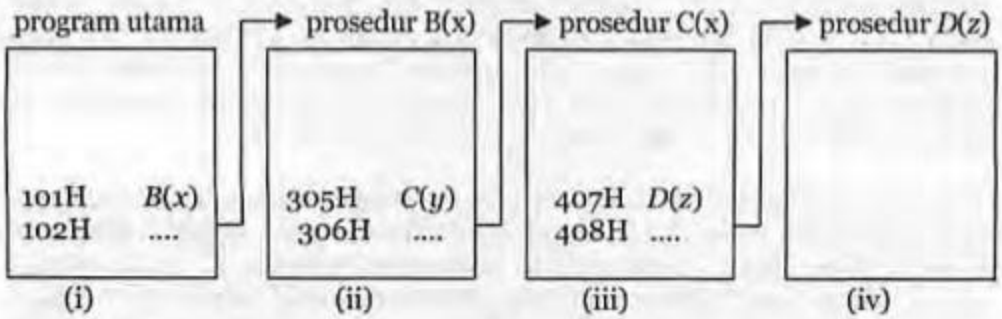


Gambar 18.8 Sebuah tumpukan dengan 6 buah elemen

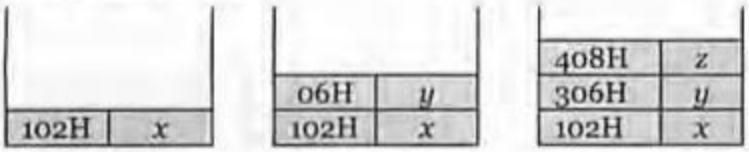
Struktur data tumpukan mempunyai dua buah operasi primitif, yaitu *push* dan *pop*.

Push(*s*, *x*) : menambahkan (*insert*) elemen *x* sebagai elemen puncak (*top*) dari tumpukan *s*
Pop(*s*) : mengambil (*delete*) elemen puncak (*top*) dari tumpukan *s*

Dengan menggunakan tumpukan, suatu prosedur/fungsi tidak kehilangan kendali ketika memanggil prosedur/fungsi lainnya. Pada setiap kali pemanggilan, akan dialokasikan memori (area data) untuk elemen baru tumpukan, kemudian elemen ini di-*push* ke dalam tumpukan. Ketika prosedur/fungsi selesai dilaksanakan, elemen di puncak tumpukan di-*pop*, alamat kembalinya diambil, dan alokasi memori untuk elemen tersebut dibebaskan (Gambar 18.9 dan 18.10). Angka heksadesimal 101H, 102H, 305H, dan seterusnya menyatakan alamat setiap instruksi di dalam memori.



Tumpukan (*stack*):

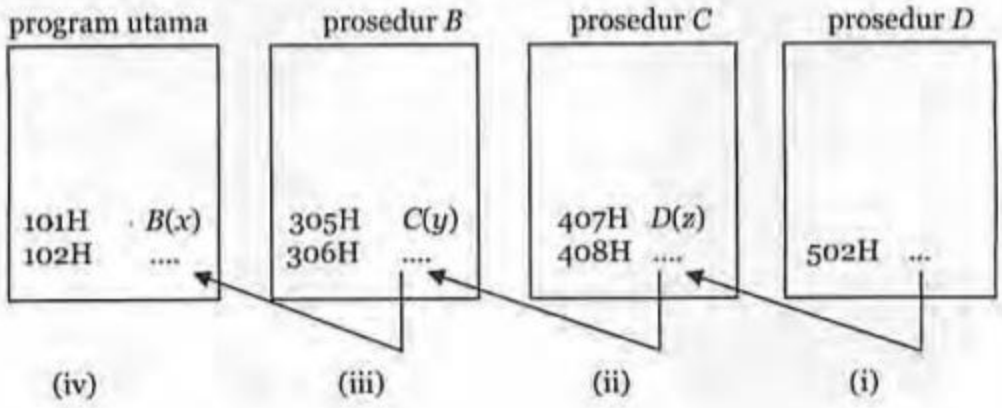


Gambar 18.10 Skema pemanggilan prosedur. Angka 101h, 102h, dan seterusnya, menyatakan alamat instruksi di dalam memori (dalam heksadesimal). Anak panah menyatakan kendali program. Setiap kali pemanggilan, elemen baru di-push ke dalam tumpukan

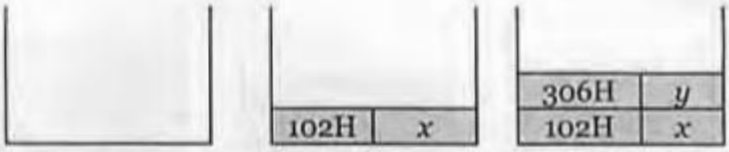
Perhatikanlah bahwa tumpukan ini tidak "terlihat" dari sisi pengguna program, sehingga disebut juga tumpukan implisit. Pengalokasian tumpukan beserta pengoperasinnya ditangani oleh *compiler*, bukan lagi urusan pemrogram.

Mekanisme yang sama juga berlaku pada waktu pemanggilan prosedur/ fungsi rekursif. Setiap kali prosedur rekursif dipanggil, alamat kembali, semua peubah lokal, dan semua parameter lokal dimasukkan (*push*) ke dalam tumpukan. Acuan terhadap parameter lokal atau peubah lokal adalah nilai yang terdapat pada elemen di puncak tumpukan.

Apabila prosedur rekursif yang "terdalam" selesai dilaksanakan, elemen di puncak tumpukan diambil (*pop*), alokasi memorinya dibebaskan, dan elemen puncak sekarang dipakai sebagai acuan parameter lokal dan peubah lokal untuk pelaksanaan prosedur rekursif sebelah luarnya.



Tumpukan (*stack*):



Gambar 18.10 Setelah prosedur selesai dilaksanakan, elemen di puncak tumpukan di-*pop*, kendali program kemudian berpindah ke instruksi yang ditunjukkan oleh alamat kembali. Alokasi memori elemen yang di-pop dibebaskan.

Contoh 18.17. Kita ingin mencetak setiap *digit* bilangan bulat *n*. Misalnya jika $n = 375$, maka dicetak 3, 7, 5 { keluaran di layar: 375 }

TulisBiasa(*n*):
(i) jika $n < 10$, cetak(*n*) { basis }
(ii) jika $n \geq 10$, { rekurens }
- TulisBiasa(*n* div 10)
- cetak(*n* mod 10)

```
098H   procedure TulisBiasa(input n:integer)
       { mencetak digit bilangan bulat dari depan ke belakang
         K.Awal : n terdefinisi
         K.Akhir: digit-digit n dicetak ke layar
              basis: jika n < 10, maka cetak(n)
              rekurens: jika n > 10 maka
                        - TulisBiasa(n div 10)
                        - cetak(n div 10)
       }
       DEKLARASI
099H     -
100H
101H   ALGORITMA:
102H     if n < 10 then
103H        write(n)
104H     else
105H        TulisBiasa(n div 10);
106H        write(n mod 10);
107H     endif
```

Algoritma 18.34 Penulisan *Integer* secara natural

Contoh pemanggilan dari program utama:

```
200H   TulisBiasa(375)
201H   writeln;
```

Ilustrasi isi tumpukan pada pemanggilan prosedur `TulisBiasa` ditunjukkan pada Gambar 18.11. Pada mulanya, tumpukan dalam keadaan kosong (Gambar 18.11(a)). Ketika instruksi

```
TulisBiasa(375)
```

dilaksanakan, alamat kembalinya, yaitu 201H dan parameter n = 375 dimasukkan ke dalam tumpukan (Gambar 18.11(b)).

Karena 375 > 10, maka

```
TulisBiasa(375 div 10)
```

dilaksanakan. Alamat kembalinya, yaitu 106H dan parameter n = 37 dimasukkan ke dalam tumpukan (Gambar 18.11(c)). Sekali lagi, karena 37 > 10, maka

```
TulisBiasa(37 div 10)
```

dilaksanakan. Alamat kembalinya, yaitu 106H dan parameter n = 3 dimasukkantaruh ke dalam tumpukan (Gambar 18.11(d)). Karena 3 < 10, maka

```
write(n)
```

dilaksanakan; nilai n = 3 dicetak ke layar. Keluar dari prosedur `TulisBiasa`, elemen puncak diambil (*pop*) dari tumpukan (Gambar 18.11(e)). Nilai n yang diacu sekarang adalah nilai n di puncak tumpukan (yaitu n = 37). Instruksi pada alamat kembali,

```
write(n mod 10)
```

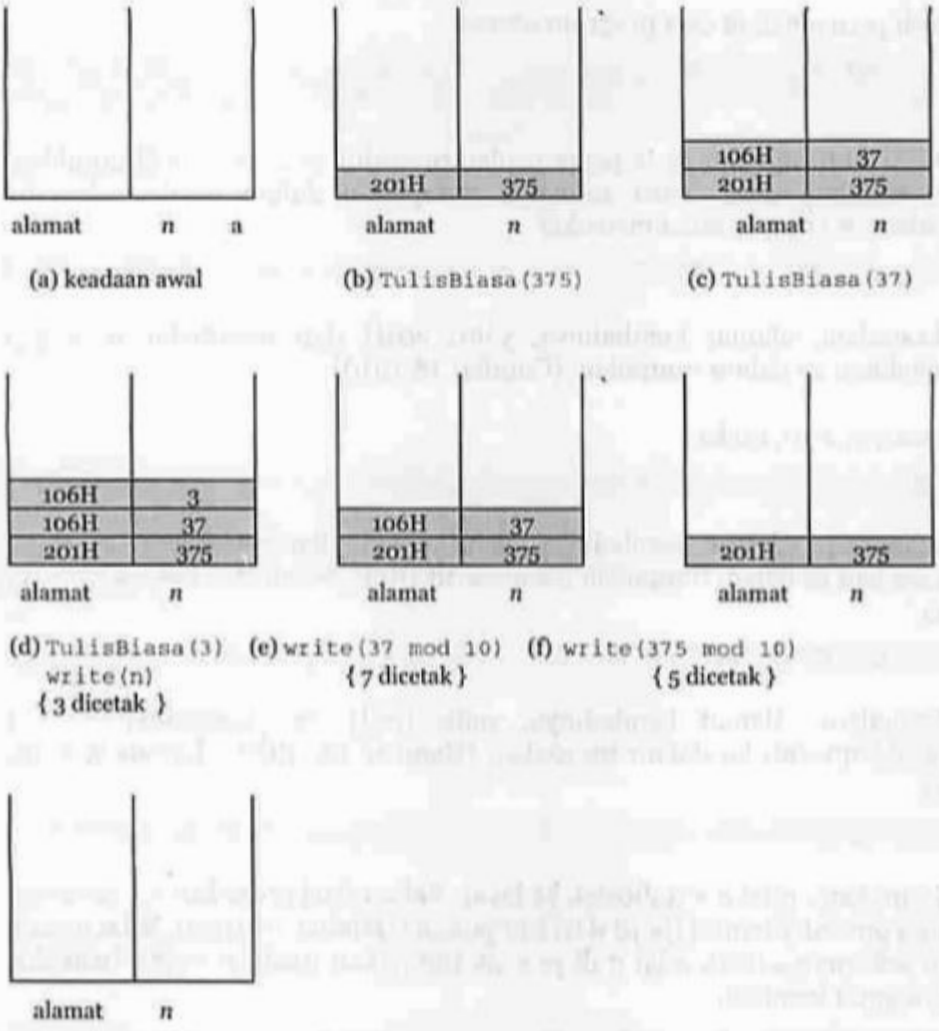
dilaksanakan, dan digit 7 dicetak ke layar. Keluar dari prosedur `TulisBiasa`, elemen puncak diambil (*pop*) dari tumpukan (Gambar 18.11(f)). Nilai n yang diacu sekarang adalah nilai n di puncak tumpukan (yaitu n = 375). Instruksi pada alamat kembali,

```
write(n mod 10)
```

dilaksanakan, dan digit 5 dicetak ke layar. Keluar dari prosedur `TulisBiasa`, elemen puncak diambil (*pop*) dari tumpukan (Gambar 18.11(g). Instruksi pada alamat kembali (yaitu 201H),

```
writeln
```

dilaksanakan. Program selesai dilaksanakan.



Gambar 18.11 Isi tumpukan pada pemanggilan TulisBiasa

Contoh 18.18. Fungsi faktorial rekursif untuk menghitung faktorial.

```
299H   function Fak(input n : integer)→integer
       { mengembalikan nilai n! }
300H
301H   DEKLARASI
302H       y : integer
303H   ALGORITMA:
304H       if n = 0 then
305H         return 1
```

```
306H        else
307H          y ← Fak(n-1)
308H          return n * y
            endif
```

Algoritma 18.35 Fungsi faktorial

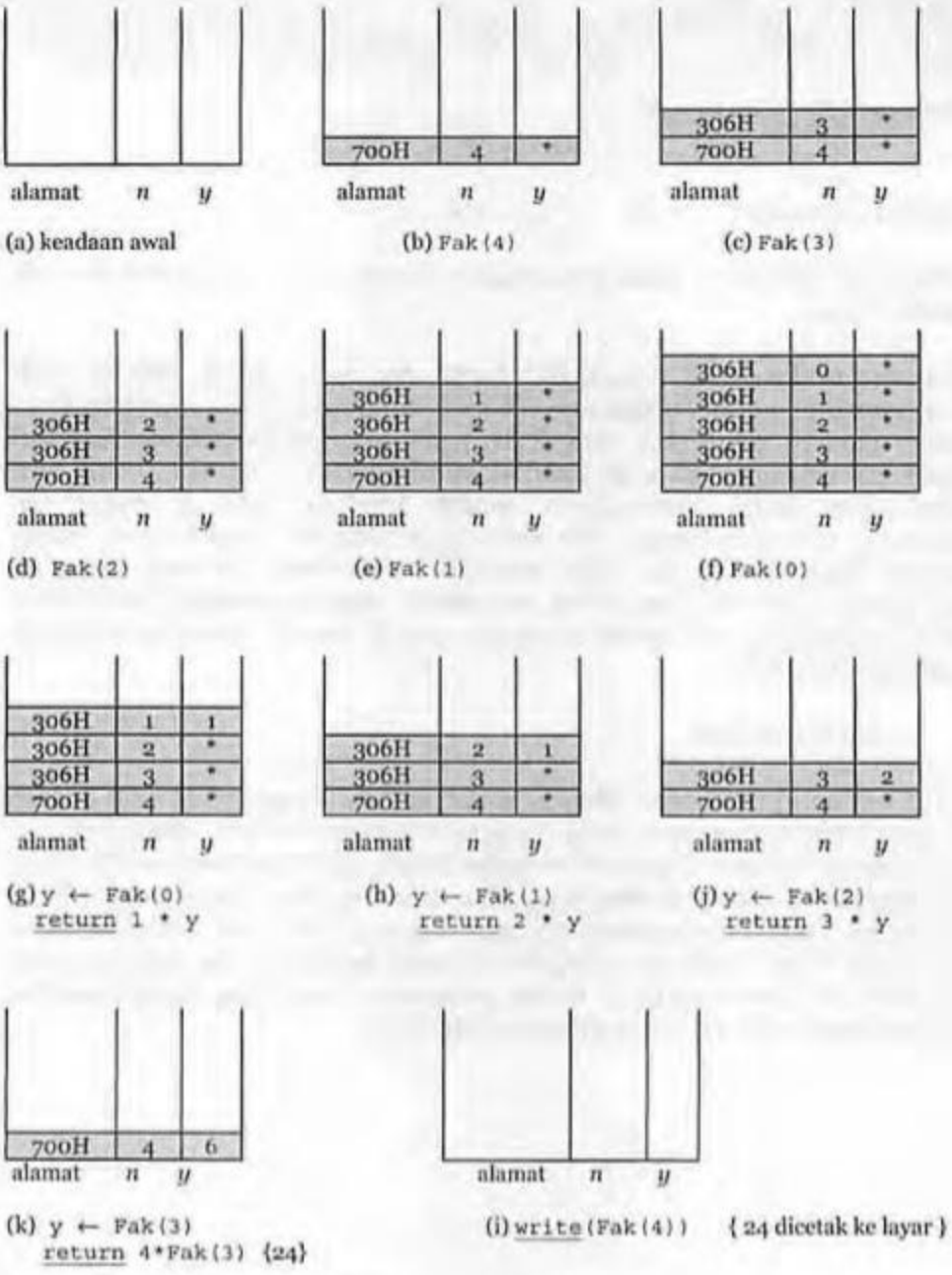
Contoh pemangilan: `700H  write(Fak(4))`

Ilustrasi isi tumpukan pada pemanggilan fungsi `Fak(4)` ditunjukkan pada Gambar 18.12.

Sekarang Anda dapat memahami harga apa yang harus dibayar oleh penggunaan teknik pemrograman rekursif. Program rekursif membutuhkan alokasi memori yang lebih banyak daripada program iteratifnya. Tiap kali terjadi pemanggilan rekursif, memori untuk elemen tumpukan yang baru dialokasikan untuk menyimpan alamat kembali, peubah lokal, dan parameter prosedur/fungsi. Jika kedalaman rekursif sangat besar, ruang memori yang dibutuhkan juga semakin besar. Pada umunya *compiler* membatasi alokasi memori untuk tumpukan. Apabila pemanggilan rekursif terus berlangsung sementara tumpukan sudah penuh, *compiler* biasanya akan memberi pesan

stack overflow

Harga lainnya yang harus dibayar adalah waktu. Program rekursif dapat berjalan lebih lambat daripada program iteratifnya. Ini disebabkan oleh dua kemungkinan. Pertama, *compiler* mungkin buruk dalam mengimplementasikan pemanggilan rekursif. Kebanyakan, jika tidak semua, *compiler PASCAL* menangani rekursif cukup baik sehingga biayanya kecil, kira-kira 5% sampai 10%. Pada kasus terburuk, program rekursif berjalan pada setengah dari kecepatan program iteratifnya. Kedua, program rekursif yang ditulis mungkin tidak mangkus, dan ini harus dihindari [ROH84].



Gambar 18.12 Isi tumpukan pada pemanggilan Fak(4)

18.6 Kapan Tidak Menggunakan Rekursif?

Secara intuitif, algoritma rekursif sangat tepat diterapkan untuk persoalan yang sifat alaminya (*natural*) memang rekursif, misalnya persoalan yang menggunakan struktur data rekursif seperti *list* dan pohon, aplikasi *games*, dan lain-lain. Persoalan seperti itu didefinisikan dalam hubungan rekursif, dan di sini teknik pemrograman rekursif menunjukkan kedayagunaannya.

Terdapat dua alasan mengapa kita harus mengubah algoritma rekursif menjadi algoritma iteratif. Pertama, bahasa pemrograman yang akan digunakan tidak mendukung pemanggilan rekursif. Hal ini terjadi pada bahasa FORTRAN yang tidak memungkinkan rekursifitas, sehingga menghalangi penyelesaian rekursif meskipun cara ini cocok untuk persoalan tertentu. Kedua, dapat terjadi bentuk rekursif tidak mangkus, memiliki kerumitan yang tidak perlu, atau terlalu lambat.

Alasan kedua ditunjukkan pada persoalan menentukan deret Fibonacci berikut:

0, 1, 1, 2, 3, 5, 8, 13, ...

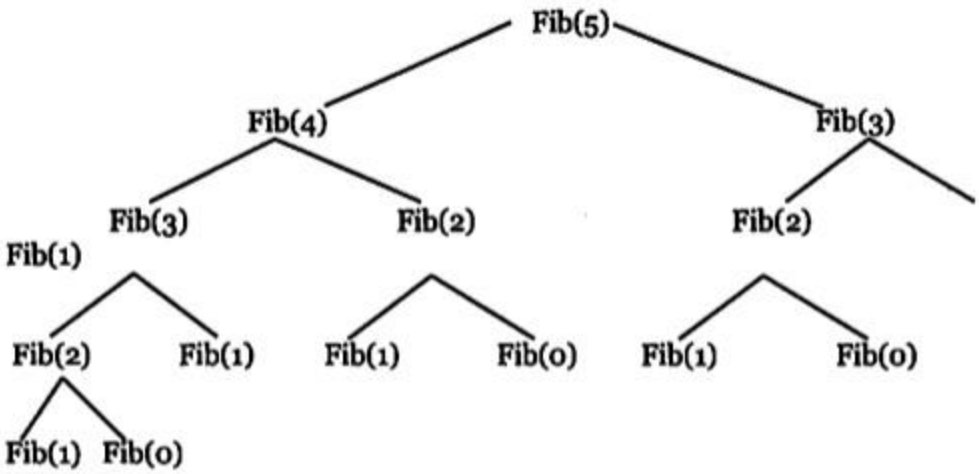
Sebagai konvensi, 0 adalah bilangan Fibonacci ke-0, 1 adalah bilangan Fibonacci ke-1, dan seterusnya. Fungsi rekursif untuk menentukan bilangan Fibonacci ke-n:

Fib(n) =
(i) n , jika $n = 0$ atau $n = 1$ { basis }
(ii) Fib($n-1$) + Fib($n-2$) , jika $n > 1$ { rekurens }

```
function Fib(input n : integer) → integer
{ menentukan bilangan Fibonacci ke-n

          basis: jika n = 0 atau n = 1, maka Fib(n) = n
          rekurens: jika n > 1 maka Fib(n) = Fib(n-1) + Fib(n-2)
}

DEKLARASI
  -

ALGORITMA:
   if (n = 0) or (n = 1) then
      return n
   else
      return FIB(n-1) + FIB(n-2)
   endif
```

Algoritma 18.36 Fungsi Fibonacci (rekursif)

Perhatikanlah bahwa setiap kali pemanggilan `Fib(n)` dengan $n > 1$ akan menghasilkan dua buah pemanggilan rekursif berikutnya, sehingga total jumlah pemanggilan meningkat secara eksponensial (pemanggilan rekursif dua kali seperti ini disebut dengan rekursif biner. Kita sudah melihat contoh rekursif biner lain pada persoalan menara Hanoi). Untuk $n = 5$, terdapat 15 kali pemanggilan `Fib(n)`. Di sini, algoritma rekursif menjadi tidak mangkus, sebab ada pengulangan perhitungan untuk n yang sama (Gambar 18.13).



Gambar 18.13 Pohon pemanggilan *Fib(n)* untuk n = 5

Dari Gambar 18.13 terlihat ada tiga kali pemanggilan rekursif untuk $n = 2$ dan dua kali untuk $n = 3$. Dengan prosedur iteratif, pengulangan perhitungan untuk n yang sama dapat dihindari dengan cara pemakaian peubah tambahan sedemikian rupa sehingga $x = Fib_i$ dan $y = Fib_{i+1}$.

Fungsi iteratif untuk menghitung deret Fibonacci sebagai berikut:

```
function FIBONACCI(input n : integer) → integer
{ versi iteratif untuk menentukan bilangan Fibonacci ke n }

DEKLARASI
   x, y, z, i : integer

ALGORITMA:
    if n = 0 then
       return 0
    else
       if n = 1 then
          return 1
       else
          x ← 0          { Fib0 }
          y ← 1          { Fib1 }
          i ← 2
```

```
        while i ≤ n do
            z ← x + y    { Fib_i }
            x ← y        { Fib_{i-1} menjadi Fib_{i-2} }
            y ← z        { Fib_i menjadi Fib_{i-1} }
            i ← i + 1
        endwhile
        { i > n }

        return z
    endif
endif
```

Algoritma 18.37 Fungsi Fibonacci (Iteratif)

Contoh ini menunjukkan bahwa kita perlu menghindari penggunaan algoritma rekursif apabila penyelesaian iteratif lebih mudah dikerjakan [WIR76].

18.7 Menghilangkan Rekursifitas

Tidak semua *compiler* bahasa pemrograman menyediakan fasilitas rekursif, misalnya bahasa FORTRAN. Bahasa seperti PASCAL, ADA, C, dan C++ mendukung pemrograman rekursif. Untuk bahasa yang tidak menyediakan fasilitas pemrograman rekursif, kita harus menghilangkan rekursifitas dengan cara mene
rjemahkan program rekursif menjadi program iteratif. Selain karena alasan *compiler*, menghilangkan rekursifitas juga dimaksudkan untuk memperoleh algoritma iteratif yang lebih mangkus daripada algoritma rekursifnya.

Alasan lain penghilangan rekursifitas adalah kendala memori. *Compiler* bahasa pemrograman membatasi besar tumpukan implisit (besar memori yang disediakan untuk tumpukannya, dalam satuan *byte*). Apabila pemanggilan rekursif memakai tumpukan lebih dari batas yang didefinisikan, *compiler* tidak dapat lagi menanganinya (pesan kesalahan: *stack overflow*).

Rekursif di Ekor

Tingkat kesulitan menghilangkan rekursifitas bergantung pada berapa buah pernyataan (*statement*) pemanggilan rekursif di dalam kode prosedur/fungsinya. Kasus yang paling sederhana adalah pemanggilan rekursif di ekor (*tail recursion*), yaitu apabila pernyataan pemanggilan rekursif terletak di akhir prosedur/fungsi (lihat skema prosedur/fungsi rekursif di upabab 9.3, yang dalam hal ini $R2(x)$ dan $S2(x)$ tidak ada). Dengan kata lain, jika pernyataan terakhir di dalam prosedur/fungsi adalah pemanggilan rekursif, gantilah pernyataan terakhir itu dengan pernyataan rekursif sebagai berikut:

a. Kerjakan bagian tanpa-rekursif dengan nilai parameter sekarang dan nilai peubah lokal sekarang.
b. Mutakhirkan parameter dan peubah lokal, dan lompat (*jump*) ke pernyataan a.

Contoh 18.19. Sebagai contoh ilustrasi, tinjau kembali prosedur `CetakList` di bawah ini:

```
procedure CetakList(input L : List)
{ mencetak setiap info elemen list yang dikunjungi
  K.Awal: L berisi alamat elemen pertama list
  K.Akhir: setiap info elemen list dicetak ke layar

   basis :
     jika L = Nil, tidak melakukan apa-apa
   rekurens:
     jika L ≠ Nil
cetak(Info(L))
CetakList(Next(L))
}

DEKLARASI
    -

ALGORITMA:
   if L ≠ Nil then
      write(Info(L))
      CetakList(Next(L))      { rekursif di ekor! }
   endif
```

Versi iteratifnya adalah:

```
procedure CetakList(input L : List)
{ mencetak setiap info elemen list yang dikunjungi
  K.Awal: L berisi alamat elemen pertama list
  K.Akhir: setiap info elemen list dicetak ke layar

  basis :
jika L = Nil, tidak melakukan apa-apa
  rekurens:
jika L ≠ Nil
         - cetak(Info(L))
         - CetakList(Next(L))
}

DEKLARASI
   label S

ALGORITMA:
   S: if L ≠ Nil then
        write(Info(L))
        L ← Next(L)
        goto S
      endif
```

Algoritma 18.38 Mencetak *List* (tanpa Rekursif)

Instruksi *goto* dapat dihilangkan untuk membuat prosedur menjadi lebih terstruktur sebagai berikut:

```
procedure CetakList(input L : List)
{ mencetak setiap info elemen list yang dikunjungi
  K.Awal: L berisi alamat elemen pertama list
   K.Akhir: setiap info elemen list dicetak ke layar

  basis :
  jika L = Nil, tidak melakukan apa-apa
  rekurens:
  jika L ≠ Nil
            - cetak(Info(L))
            - CetakList(Next(L))
}

DEKLARASI
   -

ALGORITMA:
   while L ≠ Nil do
      write(Info(L))
      L ← Next(L)
   endwhile
   { S = Nil }
```

Algoritma 18.39 Mencetak *List* (Iteratif)

Aturan ini cukup umum dan dapat diterapkan tidak hanya pada prosedur rekursif lanjar, tetapi juga pada prosedur rekursif biner. Pada rekursif lanjar, rekursifitas dihilangkan sama sekali, sedangkan pada rekursif biner, hanya satu pernyataan rekursif yang dapat dihilangkan (jika pemanggilan rekursifnya di ekor).

Sayangnya, aturan ini tidak dapat diterapkan untuk pemanggilan rekursif yang tidak di ekor, juga pada pemanggilan rekursif yang parameternya *by reference*, misalnya pada prosedur *SalinList* [ROH84]. Karena itu, kita harus membuat aturan yang lebih umum, yaitu mensimulasikan pemanggilan rekursif dengan menggunakan tumpukan eksplisit.

Menyimulasikan Pemanggilan Rekursif dengan Tumpukan Eksplisit

Yang dimaksud dengan tumpukan eksplisit (*explicit stack*) adalah tumpukan yang didefinisikan oleh pemrogram di dalam programnya. Tumpukan eksplisit menyimulasikan penanganan tumpukan implisit oleh *compiler*. Dengan menerjemahkan algoritma rekursif menjadi algoritma iteratif yang menggunakan tumpukan eksplisit, kita masih dapat mempertahankan sifat "alami" persoalan yang rekursif.

Keuntungan lain penyimulasian ini adalah mengatasi kendala batasan memori yang telah ditentukan oleh *compiler* pada tumpukan implisit. Dengan

menggunakan tumpukan eksplisit, kita dapat mendefinisikan besar tumpukan lokal yang diinginkan, atau, pengalokasian elemen tumpukan dapat dibuat dinamis (memakai *list* berkait dengan *pointer*).

Dan akhirnya, pengubahan algoritma rekursif menjadi iteratif dengan menggunakan tumpukan eksplisit dapat meningkatkan pemahaman kita tentang mekanisme pelaksanaan *procedure* rekursif itu sendiri

Pengubahan algoritma rekursif menjadi algoritma iteratif telah menjadi pokok penelitian besar sampai saat ini [ROH84]. Teknik pengubahan yang dibahas di bawah ini dikhususkan untuk rekursif lanjar saja.

Prosedur rekursif

```
procedure C(input x : xtype)
{ prosedur yang memanipulasi nilai x
  K.Awal : x terdefinisi nilainya
  K.Akhir: sekumpulan aksi yang memanipulasi nilai x dilakukan
}

DEKLARASI
   { kamus lokal, jika ada }

ALGORITMA:
  if P(x) then
     M(x)       { basis }
  else
     S1(x)
     C(F(x))    { rekurens }
     S2(x)
  endif
```

Algoritma 18.40 Skema prosedur rekursif

Prosedur iteratif

Definisikan tumpukan eksplisit sebagai berikut, bergantung pada jumlah parameter, tipe parameter, dan peubah lokal. Elemen tumpukan menyimpan parameter prosedur dan peubah lokal. Alamat kembali tidak perlu disimpan karena hanya ada satu pemanggilan rekursif (rekursif lanjar). Misalkan tumpukan diimplementasikan menggunakan larik sebagai berikut:

```
const Nmaks = ...;      { jumlah maksimum elemen tumpukan }
type xType = integer
type stack : record
               <
                elemen : array[1..Nmaks] of xType
                top    : integer
               >
s : stack
```

Algoritma 18.41 Pendefinisian tumpukan

Terdapat dua versi penyimulasian pemanggilan rekursif. Versi pertama meniru persis mekanisme penanganan pemanggilan rekursif yang sudah dijelaskan pada bagian sebelum ini. Versi kedua adalah modifikasi versi pertama.

Versi 1:

```
procedure C(input x : xType);
{ prosedur yang memanipulasi nilai x
  K.Awal : x terdefinisi nilainya
  K.Akhir: sekumpulan aksi yang memanipulasi nilai x dilakukan
}

DEKLARASI
   s : stack

ALGORITMA:
    inisialisasi(s)    { bersihkan tumpukan, TOP(s) = 0}
    push(s, x)         { masukkan parameter pemanggilan pertama kali
  ke dalam tumpukan s }
    while not P(x) do
        S1(x)
        x ← P(x)
        push(s, x)
    endwhile
   { P(x) }

   M(x)
   pop(s,x)
   while not empty(s) do
      x ← s.elemen[s.TOP]
      S2(x)
      pop(s, x)
   endwhile
   { empty(s) }
```

Algoritma 18.42 Menyimulasikan pemanggilan Rekursif (versi 1)

Versi 2:

Versi kedua ini sedikit berbeda daripada versi 1 dan tidak persis sama mengikuti mekanisme pemakaian tumpukan yang telah diterangkan sebelumnya, namun versi kedua ini lebih singkat dan lebih mudah dipahami. Perbedaannya adalah, bahwa pada pemanggilan prosedur rekursif yang pertama kali (dari luar prosedur), parameter *x* tidak langsung dimasukkan (*push*) ke dalam tumpukan. Begitu juga setelah satu pemanggilan rekursif selesai dilaksanakan, elemen puncak tumpukan diambil (*pop*) dan parameter *x* yang diacu untuk `S2(x)` adalah nilai parameter pada elemen diambil itu, bukan nilai parameter di puncak tumpukan sekarang.

```
procedure C(input x : xType);
{ prosedur yang memanipulasi nilai x
  K.Awal : x terdefinisi nilainya
  K.Akhir: sekumpulan aksi yang memanipulasi nilai x dilakukan
```

```
}
DEKLARASI
   s : stack

ALGORITMA:
    inisialisasi(s)     { bersihkan tumpukan, top(s) = 0}
    while not P(x) do
        S1(x)
        push(s,x)
        x ← F(x)
    endwhile
    { P(x) }

    M(x)
    pop(s,x)
    while not empty(s) do
      pop(s,x)
      S2(x)
    endwhile
    { empty(s) }
```

Algoritma 18.43 Menyimulasikan pemanggilan Rekursif (versi 2)

Prosedur `inisialisasi`, `push`, `pop`, dan `kosong` sebagai berikut:

```
procedure inisialisasi(input/output s : stack)
{ menginisialisasi tumpukan s
  K.Awal : s belum terdefinisi harganya
  K.Akhir: seluruh elemen tumpukan diisi nol, s.top = 0
}

DEKLARASI
   i : integer

ALGORITMA:
   for i ← 1 to Nmaks do
      s.elemen[i] ← 0
      s.top ← 0
   endfor


function empty(input s : stack) → boolean
{ true jika tumpukan s kosong, false jika sebaliknya }

DEKLARASI
    .

ALGORITMA:
   return s.top = 0


procedure push(input/output s : stack, input x : xType)
{ memasukkan x ke dalam tumpukan s
  K.Awal : x terdefinisi harganya, tumpukan s belum penuh
   K.Akhir: s.elemen[s.TOP] diisi x
}
```

```
DEKLARASI
  -

ALGORITMA:
   s.top ← s.top + 1
   s.elemen[s.top] ← x


procedure pop(input/output s : stack, output x : xType)
{ mengambil x dari tumpukan s
  K.Awal : tumpukan s tidak kosong
  K.Akhir: x berisi elemen puncak tumpukan
}

DEKLARASI
  -

ALGORITMA:
    x ← s.elemen[s.TOP]
    s.TOP ← s.TOP-1
```

Algoritma 18.44 Prosedur dasar untuk tumpukan

Contoh 18.20. Tinjau kembali prosedur *TulisBiasa* rekursif di bawah ini:

(a) Rekursif

```
procedure TulisBiasa(input n : integer)
{ mencetak digit bilangan bulat dari depan ke belakang
  K.Awal : n terdefinisi
  K.Akhir: digit-digit n dicetak ke layar

    basis: jika n < 10, maka cetak(n)
    rekurens: jika n ≥ 10 maka
                - TulisBiasa(n div 10)
                - cetak(n div 10)
}

DEKLARASI
  -

ALGORITMA:
   if n < 10 then
      write(n)
   else
      TulisBiasa(n div 10);
      write(n mod 10);
   endif
```

Algoritma 18.45 Penulisan *Integer* secara natural

Pengubahannya menjadi algoritma iteratif:

(b) Iteratif

Versi 1:

```
procedure TulisBiasa(input n : integer)
 { mencetak digit bilangan bulat dari depan ke belakang
   K.Awal : n terdefinisi
   K.Akhir: digit-digit n dicetak ke layar

      basis: jika n < 10, maka cetak(n)
      rekurens: jika n > 10 maka
                  - TulisBiasa(n div 10)
                  - cetak(n div 10)
 }

 DEKLARASI
    s : stack

 ALGORITMA:
    inisialisasi(s)
    push(s,n)
    while n ≥ 10 do
       n ← n div 10
       push(s, n)
    endwhile
    { n < 10 }

    write(n)
    pop(s,n)
    while not empty(s) do
        n ← s.elemen[s.TOP]  { nilai n yang diacu }
        write(n mod 10)
        pop(s,n)
    endwhile
    { empty(s) }
```

Algoritma 18.46 Menyimulasikan prosedur penulisan *Integer* Rekursif dengan skema versi 1

Versi 2:

```
procedure TulisBiasa(input n : integer)
 { mencetak digit bilangan bulat dari depan ke belakang
   K.Awal : n terdefinisi
   K.Akhir: digit-digit n dicetak ke layar

      basis: jika n < 10, maka cetak(n)
      rekurens: jika n > 10 maka
                  - TulisBiasa(n div 10)
                  - cetak(n div 10)
 }

 DEKLARASI
    s : stack
```

```
ALGORITMA:
    inisialisasi(s)
    while n ≥ 10 do
       push(s,n)
       n ← n div 10
    endwhile
    { n < 10 }

    write(n)
    while not empty(s) do
       pop(s,n)
       write(n mod 10)
    endwhile
    { empty(s) }
```

Algoritma 18.47 Menyimulasikan prosedur penulisan *Integer* Rekursif dengan skema versi 2

18.8 Rekursif dalam Bahasa PASCAL dan C

Tidak ada perbedaan antara pemanggilan prosedur rekursif dengan prosedur biasa (tanpa rekursif) dalam bahasa *Pascal* dan bahasa *C*. Kedua bahasa pemrograman ini mendukung rekursifitas. Di bawah ini diberikan beberapa contoh translasi notasi algoritmik fungsi/ prosedur rekursif ke dalam bahasa *Pascal* dan bahasa *C*.

Contoh 18.21. Translasi fungsi faktorial (rekursif) dari notasi algoritmik ke notasi bahasa *Pascal* dan bahasa *C*.

ALGORITMIK

```
function Fak(input n : integer) → integer
{ mengembalikan nilai n!. Algoritma rekursif.
  basis    : jika n = 0, maka 0! = 1
  rekurens: jika n > 0, maka n! = n × (n-1)!
}

DEKLARASI
    -

ALGORITMA:
    if n = 0 then
       return 1                    { basis }
    else
       return n * Fak(n - 1)       { rekurens }
    endif
```

PASCAL

```
function Fak(n : integer) : integer;
{ mengembalikan nilai n!. Algoritma rekursif.
  basis    : jika n = 0, maka 0! = 1
  rekurens: jika n > 0, maka n! = n × (n-1)!
}
begin
    if n = 0 then
       Fak := 1                    { basis }
    else
       Fak := n * Fak(n - 1); { rekurens }
    {endif}
end;
```

C

```
int Fak(int n)
/* mengembalikan nilai n!. Algoritma rekursif.
  basis    : jika n = 0, maka 0! = 1
  rekurens: jika n > 0, maka n! = n × (n-1)!
*/
{
   if (n == 0)
     return 1;                   { basis }
   else
     return n * Fak(n - 1);    { rekurens }
    /*endif*/
}
```

Contoh 18.22. Translasi prosedur menara Hanoi (rekursif) dari notasi algoritmik ke notasi bahasa *Pascal* dan bahasa *C*.

ALGORITMIK

```
procedure Hanoi(input n, A, B, C : integer)
{ memindahkan N buah piringan dari A ke B
  K.Awal : n adalah jumlah piringan, harganya sudah terdefinisi
           Pada mulanya seluruh piringan berada di tiang A,
           piringan terbesar berada paling bawah.
  K.Akhir: Seluruh piringan telah pindah ke tiang B, piringan
           paling besar berada paling bawah. Proses perpindahan
           piring dicetak ke layar

           basis:
              jika n = 1, pindahkan piringan dari A ke B
   rekurens:
              jika n > 1,
                  - angkat n-1 buah piringan dari A ke C
                  - pindahkan 1 piringan dari A ke B
                  - angkat n-1 buah piringan dari C ke B
}

DEKLARASI
```

```
ALGORITMA:
     if n = 1 then
        write('pindahkan piringan dari ',A,' ke ', B)
     else
        Hanoi(N-1,A, C, B)
        write('Pindahkan piringan dari ',A,' ke ', B)
        Hanoi(N-1, C, B, A)
     endif
```

PASCAL

```
procedure Hanoi(n, A, B, C : integer);
{ memindahkan n buah piringan dari A ke B
  K.Awal : N adalah jumlah piringan, harganya sudah terdefinisi
           Pada mulanya seluruh piringan berada di tiang A,
           piringan terbesar berada paling bawah.
  K.Akhir: Seluruh piringan telah pindah ke tiang B, piringan
           paling besar berada paling bawah. Proses perpindahan
           piring dicetak ke layar

           basis:
             jika n = 1, pindahkan piringan dari A ke B
           rekurens:
               jika n > 1,
                 - angkat n-1 buah piringan dari A ke C
                 - pindahkan 1 piringan dari A ke B
                 - angkat n-1 buah piringan dari C ke B
}

begin
     if n = 1 then
        writeln('pindahkan piringan dari ',A,' ke ', B)
     else
        begin
           Hanoi(n-1,A, C, B);
           writeln('Pindahkan piringan dari ',A,' ke ', B);
           Hanoi(n-1, C, B, A);
        end;
end;
```

C

```
void Hanoi(int n,int A,int B,int C)
/* memindahkan n buah piringan dari A ke B
  K.Awal : n adalah jumlah piringan, harganya sudah terdefinisi
           Pada mulanya seluruh piringan berada di tiang A,
           piringan terbesar berada paling bawah.
  K.Akhir: Seluruh piringan telah pindah ke tiang B, piringan
           paling besar berada paling bawah. Proses perpindahan
           piring dicetak ke layar

           basis:
             jika n = 1, pindahkan piringan dari A ke B
            rekurens:
               jika n > 1,
```

```
                - angkat N-1 buah piringan dari A ke C
                - pindahkan 1 piringan dari A ke B
                - angkat N-1 buah piringan dari C ke B
*/
{
    if (n == 1)
       printf("pindahkan piringan dari %d ke %d \n", A, B);
    else
       {
         Hanoi(n-1,A, C, B);
         printf("Pindahkan piringan dari %d ke %d \n",A, B);
         Hanoi(n-1, C, B, A);
       }
}
```

Soal Latihan Bab 18

1. Tulislah prosedur/fungsi rekursif untuk persoalan berikut:
   (a) Menghitung $S(n) = 1^3 + 2^3 + 3^3 + ... + (n-1)^3 + n^3$
   (b) Menghitung $a + b$, yang dalam hal ini a dan b adalah bilangan bulat (gunakan terminologi *successor*).
   (c) Menghitung pecahan berikut:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_1 + \cfrac{1}{\cdots + \cfrac{1}{a_n}}}}$$

   yang dalam hal ini nilai-nilai $a_0, a_1, ..., a_n$ bertipe riil.

   (d) Menghapus elemen *list* L yang kuncinya x (x mungkin tidak ada di dalam list).
   (e) Menentukan elemen terkecil dari larik *integer* $A[1..n]$.
   (f) Menggabung dua buah *list*. Salah satu atau kedua *list* mungkin kosong.
   (g) Menentukan elemen terbesar dari *list* L

2. Ubahlah prosedur iteratif ini menjadi prosedur rekursif:

```
procedure lelaran(input  n : integer)

DEKLARASI
    i : integer

ALGORITMA:
    i ← n
    while i ≠ 0 do
```

```
        write(i)
        i ← i - 1
    endwhile
    { i = 0 }
```

3. Fungsi apakah di bawah ini ?

```
function X(input/output a, b : LarikInt, input n:integer) → real
DEKLARASI

ALGORITMA:
   if n = 0 then
      return a[0]*b[0]
   else
      return X(a, b, n-1) + a[n] * b[n]
   endif
```

4. Fungsi apakah di bawah ini ?

```
function func(input n : integer) → integer
{ n > 0 }
DEKLARASI

ALGORITMA:
   if n = 0 then
      return 0
   else
      return n + func(n-1);
   endif
```

5. Modifikasilah fungsi *gcd* (pembagi bersama terbesar) sehingga ia juga dapat digunakan untuk menghitung *gcd*(*x*,*y*) untuk $x < y$.

6. Misalkan panit(n,k) menyatakan jumlah panitia yang terdiri dari k orang yang dapat dibentuk dari n orang. Sebagai contoh, panit(4,3) = 4, karena diberikan empat orang A, B, C, dan D, maka terdapat empat kemungkinan panitia, yaitu ABC, ABD, ACD, dan BCD. Buktikan kesamaan berikut:

   $panit(n,k) = panit(n-1k) + panit(n-1,k-1)$

   Buatlah fungsi rekursifnya untuk $n, k \geq 1$.

7. Fungsi Ackerman A didefinisikan untuk semua argumen bilangan bulat m dan n sebagai berikut:

   $A(0\ n) = n + 1$
   $A(m, 0) = A(m-1,1), \quad (m > 0)$
   $A(m, n) = A(m-1, A(m, n-1)) \quad (m, n > 0)$

   Tulislah fungsi rekursif untuk menghitung $A(m,n)$.

8. Transformasikan algoritma rekursif yang sudah ditulis di *dalam* buku ini, yaitu
   (a) `function Fak`
   (b) `procedure BalikAngka`
   (c) `procedure Hanoi`
   (d) `procedure CetakList`
   (e) `procedure SalinList`

   menjadi algoritma iteratif dengan:
   (i) skema penghilangan rekursif di ekor
   (ii) menggunakan tumpukan eksplisit.

19

Contoh-contoh Pemecahan Masalah III

Bab 19 ini menyajikan contoh-contoh masalah dan penyelesaiannya, dengan topik masalah yang beragam. Contoh masalah yang disajikan mencakup masalah pencarian, pengurutan, matriks, arsip beruntun, dan algoritma rekursif. Belajar melalui contoh adalah salah satu cara terbaik belajar algoritma dan pemrograman. Melalui contoh kasus, diharapakan anda menemukan pola-pola yang dapat digunakan untuk memecahkan masalah yang sejenis.

Contoh Masalah 19.1: Penyisipan Pada Larik Terurut

Misalkan sebuah larik *integer* yang berisi n buah elemen berisi nilai-nilai *integer* yang sudah terurut menaik. Tuliskan sebuah algoritma untuk menyisipkan elemen baru, x, ke dalam larik tersebut pada posisi yang tepat sehingga larik tetap terurut menaik. Diasumsikan ukuran larik setelah penambahan elemen baru tidak melebihi ukuran maksimum larik (*Nmaks*).

Contoh:

8	10	21	25	27	76
1	2	3	4	5	6

Sisipkan $x = 24$:

8	10	21	25	24	27	76
1	2	3	4	5	6	7

Penyelesaian

Ada dua cara yang dapat digunakan untuk menyisipkan elemen baru. Cara pertama: cari terlebih dahulu posisi yang tepat untuk *x*, misalkan pada posisi ke-*i*, kemudian geser elemen-elemen ke-*i*, *i* + 1, *i* + 2, ..., *n* satu posisi ke kanan, lalu tempatkan *x* pada elemen ke-*i*.

Algoritma penyisipan selengkapnya sebagai berikut:

```
procedure SisipX_1(input/output L : LarikInt, input n : integer,
                   input x : integer)
{ Menyisipkan x ke dalam larik L[1..n] yang sudah terurut menaik
sedemikian sehingga setelah penyisipan larik L tetap terurut menaik.
Asumsi: n + 1 ≤ Nmaks, Nmaks adalah ukuran maksimum larik. }
{ K.Awal : Elemen-elemen larik L sudah terurut menaik; x sudah
  terdefinisi nilainya. }
{ K.Akhir : Elemen-elemen larik L tetap terurut. }

DEKLARASI
i : integer      { indeks larik }
  ketemu : boolean { untuk menyatakan posisi penyisipan ditemukan }

ALGORITMA:
    { cari posisi yang tepat untuk x di dalam L[1..n], yaitu elemen
      L[i] sedemikian sehingga x < L[i] }
    i ← 1
    ketemu ← false
    while (i ≤ n) and (not ketemu) do
      if x < L[i] then
        ketemu ← true
      else
        i ← i + 1
      endif
    endwhile
    { i > n or ketemu }

    if ketemu then
      { geser elemen L[i], L[i + 1], ..., L[n] satu posisi ke kanan }
      for j ← n downto i do
        L[j + 1] ← L[j]
      endfor

      { sisipkan x ke dalam L[i] }
      L[i] ← x
    else
      { sisipkan x pada posisi L[n + 1] }
      L[n + 1] ← x
    endif
```

Algoritma 19.1 Penyisipan elemen x ke dalam larik yang sudah terurut menaik (cara 1)

Cara kedua lebih mangkus daripada cara pertama, yang dalam hal ini kita melakukan pergeseran ketika mencari posisi yang tepat untuk *x*. Cara kedua ini diilhami oleh metode pengurutan sisip. Proses pembandingan dilakukan

secara mundur, dimulai dari elemen terakhir sampai posisi penyisipan ditemukan.

Algoritma penyisipan selengkapnya untuk cara kedua sebagai berikut:

```
procedure SisipX_2(input/output L : LarikInt, input/output n : integer,
                   input x : integer)
{ Menyisipkan x ke dalam larik L[1..n] yang sudah terurut menaik
sedemikian sehingga setelah penyisipan larik L tetap terurut menaik.
  Asumsi: n + 1 ≤ Nmaks. Nmaks adalah ukuran maksimum larik. }
{ K.Awal : Elemen-elemen larik L sudah terurut menaik; x sudah
  terdefinisi nilainya. }
{ K.Akhir : Elemen-elemen larik L tetap terurut. }

DEKLARASI
i : integer      { indeka larik }
ketemu : boolean { untuk menyatakan posisi penyisipan ditemukan }

ALGORITMA:
    { cari posisi yang tepat untuk x di dalam L[1..n] sambil
      menggeser }
    i ← n
    ketemu ← false
    while (i ≥ 1) and (not ketemu) do
      if x < L[i] then
         L[i + 1] ← L[i]       { geser }
         i ← i - 1
      else
         ketemu ← true
      endif
    endwhile
    { i < 1 or ketemu }

    L[i + 1] ← x  { sisipkan x pada tempat yang sesuai }
```

Algoritma 19.2 Penyisipan elemen x ke dalam larik yang sudah terurut menaik (cara 2)

Contoh Masalah 19.2: Penyisipan terurut selama pembacaan data

Gunakan Algoritma 19.2 di atas untuk membuat program utama yang mengurutkan n buah data *integer* yang di-*entry* dari papan ketik. Dalam hal ini, setiap kali data dibaca, data tersebut dicarikan posisinya yang tepat di dalam larik sehingga larik tetap terurut menaik. Proses pembacaan data berakhir bila data yang dimasukkan adalah 9999.

Penyelesaian

Untuk menggunakan Algoritma 19.2, larik harus berisi minimal 1 elemen karena proses penyisipan memerlukan pembandingan dengan elemen yang sudah terdapat di dalam larik.

```
PROGRAM Baca_dan_Urut
{ Membaca data dari papan ketik dan menyisipkannya ke dalam larik L
sedemikian sehingga larik L terurut menaik. }

DEKLARASI
   const Nmaks = 1000  { jumlah maksimum elemen larik }
   type LarikInt = array [1..Nmaks] of integer

   L : LarikInt
   i, n, x : unteger

   procedure SisipX_2(input/output L : LarikInt,
                      input/output n : integer, input x : integer)
  { Menyisipkan x ke dalam larik L[1..n] yang sudah terurut menaik
    sedemikian sehingga setelah penyisipan larik L tetap terurut menaik.
    Asumsi: n + 1 ≤ Nmaks. Nmaks adalah ukuran maksimum larik. }

ALGORITMA:
   read(n)    { baca ukuran larik; n ≤ Nmaks }
   read(x)    { baca data pertama }
   L[1] ← x
   for i ← 2 to n do
     read(x)
     SisipX_2(L, i - 1, x)  { sisipkan x di dalam L[1..i-1] }
   endfor
```

Algoritma 19.3 Program penyisipan terurut

Contoh Masalah 19.3: Pengurutan Arsip Beruntun (1)

Arsip dapat menyimpan data yang lebih banyak daripada disimpan di dalam memori utama. Mengurutkan arsip beruntun memerlukan teknik khusus. Agar proses pengurutan berlangsung cepat, maka pengurutan data tidak dilakukan langsung di dalam berkas, tetapi isi arsip dibaca ke dalam beberapa buah larik. Selanjutnya larik diurut dan hasil pengurutan disimpan kembali ke dalam arsip.

Misalkan arsip beruntun *F* berisi data mahasiswa yang deklarasi arsipnya sebagai berikut:

```
DEKLARASI
   type DataMhs : record <NIM : integer, Nama : string, IP : real>
   type ArsipMhs : File of DataMhs

   F : ArsipMhs
```

Algoritma 19.4 Deklarasi arsip data mahasiswa

Larik *M* digunakan untuk menyimpan data dari arsip dan dideklarasikan sebagai berikut:

```
DEKLARASI
  const Nmaks = 1000
  type TabMhs : array[1..Nmaks] of DataMhs

  M : TabMhs
```

Algoritma19.5 Deklarasi larik yang bertipe data mahasiswa

Buatlah algoritma yang membaca isi arsip ke dalam larik *M*, mengurutkan larik berdasarkan NIM yang terurut menaik dengan metode pengurutan seleksi-maksimum, dan menyimpan hasil pengurutan ke dalam arsip semula.

Penyelesaian:

Mula-mula, baca arsip dan simpan seluruh datanya ke dalam larik *M*:

```
procedure SalinArsipMhs(input F : ArsipMhs, output  M : TabMhs,
                        output n : integer)
{ Menyalin seluruh isi arsip F ke dalam larik M. }
{ K.Awal : sembarang }
{ K.Akhir: M berisi salinan seluruh rekaman dari arsip F;
  n berisi jumlah seluruh data di dalam arsip (n menjadi ukuran larik
  efektif }

DEKLARASI
  RekMhs    : DataMhs

ALGORITMA:
  Open(F, 1)
  n ← 0
  while not EOF(F) do
    Fread(F, RekMhs)    { baca RekMhs dari arsip F }
    n ← n + 1
    M[n] ← RekMhs       { masukkan RekMhs ke dalam M[n] }
  endwhile
  { EOF(F) }

  Close(F)
```

Algoritma.19.6 Menyalin isi arsip F ke dalam larik M

Setelah isi arsip disalin ke dalam larik *M*, selanjutnya larik diurut menaik berdasarkan *NIM* dengan metode pengurutan seleksi-maksimum:

```
procedure SortDataMhs(input/output M : TabMhs, input n : integer)
{ Mengurutkan elemen larik M[1..n] sehingga tersusun menaik
berdasarkan NIM dengan metode pengurutan seleksi-maksimum. }
{ K.Awal : Elemen-elemen larik M sudah terdefinisi. }
{ K.Akhir: Elemen larik M terurut menaik sedemikian sehingga
           M[1] ≤ M[2] ≤ ... ≤ M[n] }

DEKLARASI
i     : integer       { pencacah pass }
j     : integer       { pencacah untuk mencari nilai maksimum }
```

```
imaks : integer          { indeks yang berisi nilai maksimum sementara }

    procedure Tukar (input/output a : Mahasiswa,
                     input/output b : Mahasiswa)
        { Mempertukarkan elemen a dan b }

ALGORITMA:
for i ← n downto 2 do     { jumlah pass sebanyak n - 1 }

   imaks ← 1
   for j ← 2 to i do
     if M[j].NIM > M[imaks].NIM then
        imaks ← j
     endif
   endfor

   {pertukarkan M[imaks] dengan M[i]}
   Tukar(M[imaks], M[i])

endfor
```

Algoritma 19.7 Pengurutan larik dengan metode pengurutan seleksi-maksimum

Catatan:
Sebelum mengurutkan larik, periksa apakah larik berisi minimal satu buah elemen. Jika arsip kosong, maka pembacaan arsip hanya menghasilkan larik yang tidak terdefinisi elemen-elemennya (atau larik "kosong"). Kondisi larik kosong dapat diperiksa melalui nilai n yang dihasilkan dari prosedur BacaArsipMhs di atas:

```
SalinArsipMhs(F, M, n)
if n = 0 then
   write('Larik kosong')
else
   SortDataMhs(M, n)
endif
```

Setelah larik terurut, selanjutnya arsip *F* dibuka untuk penulisan, dan seluruh isi larik disalin kembali ke dalam arsip:

```
procedure SalinLarikMhs(input M : TabMhs, input n : integer,
                        output F : ArsipMhs )
{ Menyalin seluruh elemen larik M ke dalam arsip F. }
{ K.Awal : Larik M[1..n] sudah berisi elemen-elemen yang sudah terurut
  berdasarkan NIM; }
{ K.Akhir: F berisi salinan seluruh rekaman dari larik M. }

DEKLARASI
  i  : integer

ALGORITMA:
  Open(F, 2)
  for i ← 1 to n do
    Fwrite(F, M[i])   { salin M[i] ke dalam arsip F }
```

```
endfor

Close(F)
```

Algoritma 19.8 Menyalin larik M ke dalam arsip F

Keseluruhan proses pengurutan arsip dapat dinyatakan dengan pemanggilan prosedur-prosedur di atas sebagai berikut:

```
Open(F, 1)    { buka arsip F untuk dibaca }
SalinArsipMhs(F, M, n)
Close(F)      { tutup arsip F }

if n = 0 then
   write('Larik kosong')
else
   SortDataMhs(M, n)
endif

Open(F, 2)    { buka arsip F untuk penulisan }
SalinLarikMhs(M, n, F)
Close(F)      { tutup arsip F }
```

Contoh Masalah 19.4: Pengurutan Arsip Beruntun (2)

Contoh Masalah 19.3 di atas dikembangkan sedemikian sehingga isi arsip dibagi menjadi dua bagian, masing-masing bagian disimpan ke dalam dua buah larik. Setiap larik diurut menaik berdasarkan NIM, kemudian hasil pengurutan kedua larik tersebut digabung dengan metode penggabungan larik terurut yang sudah pernah diberikan di dalam bab 16. Hasil penggabungan disimpan ke dalam arsip semula. Sebagai contoh, misalkan arsip berisi 2000 data. Teknik pengurutannya sebagai berikut: Baca 1000 buah data arsip pertama ke dalam larik pertama, 1000 data kedua ke dalam larik kedua. Urutkan larik ini, lalu gabungkan hasil pengurutan ke dalam ke arsip semula.

Buatlah algoritma pengurutan arsip dengan teknik ini. Asumsikan arsip tidak kosong dan jumlah rekaman (data) di dalam arsip di atas lebih dari 1000 buah dan tidak melebihi 2000.

Penyelesaian:

Mula-mula, baca arsip dengan ke dalam larik $M1[1..n1]$ dan $M2[1..n2]$ sebagai berikut:

```
procedure SalinArsipMhs2(input F : ArsipMhs, output  M1, M2 : TabMhs,
                         output n1, n2 : integer)
{ Menyalin seluruh isi arsip F ke dalam larik M1[1..n1] dan M2[1..n2].
}
{ K.Awal : sembarang }
{ K.Akhir: M1 dan M2 berisi salinan seluruh rekaman dari arsip F;
  n1 berisi jumlah data di dalam M1 dan n2 berisi jumlah data di dalam
```

```
  M2.}

DEKLARASI
  RekMhs    : DataMhs

ALGORITMA:
  Open(F, 1)
  n1 ← 0
  while (not EOF(F)) and (n1 ≤ 1000) do
   Fread(F, RekMhs)    { baca RekMhs dari arsip F }
   n1 ← n1 + 1
   M[n1] ← RekMhs      { masukkan RekMhs ke dalam M1[n1] }
  endwhile,
  { EOF(F) }

  if not EOF(F) then    { isi arsip belum habis dibaca }
     n2 ← 0
     while not EOF(F) do
       Fread(F, RekMhs)    { baca RekMhs dari arsip F }
       n2 ← n2 + 1
       M[n2] ← RekMhs      { masukkan RekMhs ke dalam M[n2] }

     endwhile
     { EOF(F) }
  endif
```

Algoritma 19.9 Menyalin isi arsip F ke dalam larik M1 dan M2

Setelah isi arsip disalin ke dalam larik *M*1 dan *M*2, selanjutnya setiap larik diurut menaik berdasarkan *NIM* dengan metode pengurutan seleksi-maksimum (panggil prosedur SorDataMhs yang sudah didefinisikan di dalam Algoritma 19.7):

```
SortDataMhs(M1, n1)
SortDataMhs(M2, n2)
```

Bagian akhir dari proses pengurutan ini adalah menggabungkan kedua buah larik terurut ini ke dalam arsip sehingga data di dalam arsip terurut menaik:

```
procedure  GabungArsip2(input M1, M2 : TabMhs, input n1, n2 : integer,
                        output F : ArsipMhs)

{ Menggabungkan dua buah larik, M1 dan M2, yang sudah terurut menaik
ke dalam sebuah arsip, F, yang berisi data terurut. }
{ K.Awal : Larik M1[1..n1] dan M2[1..n2] sudah terurut menaik. }
{ K.Akhir: arsip F berisi hasil penggabungan larik M1 dan M2. }

DEKLARASI
   i, j : integer


ALGORITMA:
   i ← 1 { mulai dari elemen pertama larik M1 }
   j ← 1 { mulai dari elemen pertama larik M2 }
```

```
while (i ≤ n1) and (j ≤ n2) do
    if (M1[i].NIM ≤ M2[j].NIM) then
      Fwrite(F, M1[i]
      i ← i + 1  { larik M1 maju satu rekaman }
    else
      Fwrite(F, M2[j]
      j ← j + 1  { larik M2 maju satu rekaman }
    endif
endwhile
{ i > n1 or j > n2 }

{ salin rekaman yang tersisa, dari larik M1 atau M2 }

 while (i ≤ n1) do   { jika yang tersisa adalah larik M1 }
    Fwrite(F, M1[i]
    i ← i + 1  { larik M1 maju satu rekaman }
 endwhile
 { i >n1 }

 { simpan hasil pengurutan ke dalam arsip F }
 Open(F, 2)    { buka arsip F untuk penulisan }
 while (j ≤ n2) do   { jika yang tersisa adalah larik M2 }
    Fwrite(F, M2[j]
    j ← j + 1  { larik M2 maju satu rekaman }
 endwhile
 { j > n2 }

 Close(F)
```

Algoritma 19.10 Penggabungan dua larik terurut menjadi satu arsip terurut

Contoh Masalah 19.5: Menghitung frekuensi kemunculan nilai di dalam matriks

Di dalam bidang pengolahan citra digital (*image processing*), citra digital yang berukuran $m \times n$ *pixel* (tinggi m *pixel* dan lebar m *pixel*) direpresentasikan dengan matrik yang berukuran $m \times n$ (m buah baris dan n buah kolom). Setiap elemen matriks berisi **nilai keabuan** (*grey level*) *pixel*. Nilai keabuan ini rentangnya adalah dari 0 sampai x. Pada citra hitam-putih, nilai 0 menyatakan *pixel* tersebut berwarna hitam, nilai x menyatakan putih, sedangkan antara 0 dan x menyatakan pergeseran warna dari hitam menuju putih. Salah satu proses penting dalam pengolahan citra digital adalah menghitung frekuensi kemunculan nilai-nilai keabuan di dalam citra.

Sebagai contoh, misalkan matriks di bawah ini menyatakan citra digital yang berukuran 8 × 8 *pixel* dengan nilai keabuan dari 0 sampai 15 (ada 16 buah derajat keabuan):

$$\begin{bmatrix} 3 & 7 & 7 & 8 & 10 & 12 & 14 & 10 \\ 2 & 0 & 0 & 0 & 1 & 8 & 15 & 15 \\ 14 & 6 & 5 & 9 & 8 & 10 & 9 & 12 \\ 12 & 12 & 11 & 8 & 8 & 10 & 11 & 1 \\ 0 & 2 & 3 & 4 & 5 & 13 & 10 & 14 \\ 4 & 5 & 0 & 0 & 1 & 0 & 2 & 2 \\ 15 & 13 & 11 & 10 & 9 & 9 & 8 & 7 \\ 2 & 1 & 0 & 10 & 11 & 14 & 13 & 12 \end{bmatrix}$$

Tabulasi perhitungan frekuensi kemunculan nilai-nilai keabuan *pixel* ditunjukan pada tabel berikut:

i	f_i
0	8
1	4
2	5
3	2
4	2
5	3
6	1
7	3
8	6
9	3
10	7
11	4
12	5
13	3
14	4
15	3

Setelah frekuensi kemunculan nilai keabuan dihitung, langkah selanjutnya adalah membuat histogramnya, yaitu grafik yang menggambarkan penyebaran nilai-nilai keabuan di dalam citra, seperti contoh di bawah ini (satu karakter asterik menyatakan satu unit):

```
0    ********
1    ****
2    *****
3    **
4    **
5    ***
6    *
```

```
7       ***
8       ******
9       ***
10      *******
11      ****
12      *****
13      ***
14      ****
15      ***
```

Persoalan:
Misalkan diberikan sebuah citra *C* yang berukuran $m \times n$ dan mempunyai 256 derajat keabuan (dari 0 sampai 255). Buatlah program (nyatakan sebagai prosedur saja) dalam bahasa *Pascal* dan bahasa *C* untuk membuat histogram citra.

Penyelesaian:

PASCAL:

Deklarasi matriks citra sebagai berikut:

```
const
  NbarisMaks = 1000;    { jumlah baris maksimum }
  NkolomMaks = 1000;    { jumlah kolom maksimum }
type
  citra = array [1..NbarisMaks, 1..NKolomMaks] of byte;
  { elemen matriks bertipe byte karena rentang nilainya hanya 0 - 255
}
```

Algoritma 6.11 Deklarasi matriks citra (*Pascal*)

Prosedur untuk menghitung frekuensi kemunculan nilai-nilai keabuan dan menggambarkan histogram sebagai berikut:

```
procedure BuatHistogram(C : citra; m, n : integer);
{ Membuat histogram untuk citra C yang berukuran m x n.
  K. Awal: C sudah terdefinisi elemen-elemennya.
  K. Akhir: Gambar histogram tercetak di layar.
}
var
   i, j : integer;
   frek : array[0..255] of integer;

begin
{ inisialisasi frek[0..255] dengan 0 }
    for i := 0 to 255 do
       frek[i] := 0;
    {endfor}

    { hitung frekuensi kemunculan setiap nilai keabuan }
    for i:= 1 to m do
```

```
      for j := 0 to n do
         frek[C[i,j]] := frek[C[i,j]] + 1;
      {endfor}
   {endfor}

   { gambarkan histogramnya }
   for i := 0 to 255 do
    begin
      write(i, '   ');
      for j := 1 to frek[i] do
        write('*');
     {endfor}
     writeln;  { pindah ke baris baru }
    end;
  {endfor}

end;
```

Algoritma 6.12 Membuat histogram dari sebuah citra yang mempunyai 256 nilai keabuan (*Pascal*).

C:

Deklarasi matriks citra sebagai berikut:

```
#define NbarisMaks 1000   /* jumlah baris maksimum */
#define NkolomMaks 1000   /* jumlah kolom maksimum */

typedef unsigned char citra[NbarisMaks + 1][NkolomMaks + 1];
/* elemen matriks bertipe unsigned char karena rentang nilainya hanya
  0 - 255 */
```

Algoritma 6.13 Deklarasi matriks citra (bahasa *C*)

Prosedur untuk menghitung frekuensi kemunculan nilai-nilai keabuan dan menggambarkan histogram sebagai berikut:

```
void BuatHistogram(citra C, int m, int n)
/* Membuat histogram untuk citra C yang berukuran m x n.
  K. Awal: C sudah terdefinisi elemen-elemennya.
  K. Akhir: Gambar histogram tercetak di layar.
*/
{
   int i, j;
   int frek[256];

   for(i = 0; i <= 255; i++)
      frek[i] = 0;
   /* endfor */

   for(i = 1; i <= m; i++)
       for(j = 0; j <= n; j++)
          frek[C[i][j]] = frek[C[i][j]] + 1;
       /*endfor*/
   /*endfor*/

   /* gambarkan histogramnya */
```

```
    for(i = 0; i <= 255; i++)
    {
       printf("%d    ", i);
       for (j = 1; j <= frek[i]; j++) do
          printf("*");
       /*endfor*/
       printf("/n");  /* pindah ke baris baru */
    }
    {endfor}

}
```

Algoritma 6.14 Membuat histogram dari sebuah citra yang mempunyai 256 nilai keabuan (bahasa C).

Contoh Masalah 19.6: Menggambar mistar [SED92]

Misalkan Anda diminta menggambar tanda batang pada setiap sentimeter sebuah mistar (penggaris), seperti gambar berikut:

Tanda batang pada titik tengah mistar paling panjang, lalu pada setiap ¼ selangnya lebih pendek, dan lebih pendek lagi pada 1/8 selang, begitu seterusnya. Jika diinginkan resolusinya $1/2^n$, buatlah program dalam bahasa *Pascal* dan bahasa *C* untuk menggambar tanda-tanda batang pada mistar di atas dengan meletakkan tanda batang pada setiap titik antara 0 dan 2^n, titik ujung tidak termasuk. Disediakan prosedur `mark(x, h)` untuk menggambarkan tanda batang setinggi h pada posisi x. Tanda batang di tengah mistar tingginya n satuan, tanda batang di kiri dan kanannya tingginya $n - 1$ satuan, begitu seterusnya.

Penyelesaian:

Persoalan ini lebih alami diselesaikan secara rekursif. Ujung kiri mistar adalah 0 dan ujung kanan mistar adalah 2^n (n sembarang). Mula-mula tanda batang dilukis di tengah mistar, lalu selang mistar dibagi menjadi 2 bagian yang berukuran sama. Lukislah tanda batang yang lebih pendek pada tiap bagian dengan menggunakan prosedur yang sama (rekursif). Kondisi berhenti rekursi adalah bila tinggi batang sama dengan nol!

Prosedur penggambaran tanda batang dalam bahasa *Pascal* dan bahasa *C* sebagai berikut:

```
procedure LukisBatang(ki, ka, h : integer);
{ Melukis tanda batang pada mistar dari ujung kiri sampai ujung kanan.
  K. Awal: ki adalah posisi ujung kiri mistar, ka adalah posisi ujung
  kanan mistar; kedua nilai ini sudah terdefinisi.
  K. Akhir: Gambar tanda-tanda batang tercetak di layar.
}
var
   x : integer;

begin
if h > 0 then
 begin
   x := (ki + ka) div 2;
   mark(x, h);     /* Prosedur untuk menggambar garis setinggi h }
   LukisBatang(kiri, x, h - 1);
   LukisBatang(x, kanan, h - 1);
 end;
 end;
```

Algoritma 19.15 Melukis tanda batang pada mistar secara rekursif (*Pascal*)

```
void LukisBatang(int ki, int ka, int h)
/* Melukis tanda batang pada mistar dari ujung kiri sampai ujung kanan.
  K. Awal: ki adalah posisi ujung kiri mistar, ka adalah posisi ujung
  kanan mistar; kedua nilai ini sudah terdefinisi.
  K. Akhir: Gambar tanda-tanda batang tercetak di layar.
*/
{
   int x;

   if (h > 0)
    {
       x = (ki + ka)/2;
       mark(x, h);
       LukisBatang(kiri, x, h - 1);
       LukisBatang(x, kanan, h - 1);
    }
 }
```

Algoritma 19.16 Melukis tanda batang pada mistar secara rekursif (*C*)

Contoh pemanggilan prosedur:

```
LukisBatang(0, 64, 6)
```

Artinya melukis tanda-tanda batang dari posisi 0 sampai 64 dengan tinggi batang di tengah mistar adalah 6 satuan.

Contoh Masalah 19.7: Membuat permutasi n buah huruf

Buatlah algoritma rekursif dalam notasi bahasa *Pascal* untuk membuat permutasi (susunan berbeda dari pengaturan elemen-elemen objek) dari

string yang terdiri dari *n* buah huruf. Misalkan untuk *string* 'ABC', semua permutasinya adalah:

ABC
ACB
BAC
BCA
CAB
CBA

Penyelesaian:

Misalkan *string* yang panjangya *n* karakter dinyatakan sebagai larik karakter. Persoalan ini lebih alami diselesaikan secara rekursif. Dapat kita lihat bahwa permutasi dari *n* buah huruf adalah sebuah huruf disambung dengan permutasi *n* – 1 buah huruf sisanya. Dalam membuat permutasi ini, kita memerlukan pertukaran huruf agar susunan huruf-huruf dalam setiap permutasi tersebut selalu berbeda.

Deklarasi larik karakter dan algoritma pembangkitan permutasi sebagai berikut:

```
const n = …;
type LarikKar = array[1..n] of char;

procedure BangkitkanPermutasi(C : LarikKar; i : integer)
{ Membangkitkan semua permutasi dari huruf-huruf di dalam larik
C[1..i].
  K. Awal: C sudah terdefinisi elemen-elemennya.
  K. Akhir: semua permutasi dicetak ke layar.
}
var
    i, j : integer;

begin
    if i = n then  { basis }
      begin
        for j := 1 to n do
          write(C[j]);
          {endfor}
          writeln;
      end
    else           { rekurens }
    begin
      for j := i to n do
        begin
          Tukar(C[j], C[i]);
          BangkitkanPermutasi(C, i + 1);
        end;
    end;
end;
```

Algoritma 19.17 Membangkitkan semua permutasi dari n buah huruf

Pemanggilan prosedur pertama kali:

```
BangkitkanPermutasi(C, 1)
```

Isi prosedur Tukar sebagai berikut:

```
procedure Tukar(var a, b : integer)
{ Mempertukarkan nilai a dan b .
  K. Awal: a dan b sudah terdefinisi nilainya.
  K. Akhir: a berisi nilai b, b berisi nilai a semula.
}
var
    temp : char;

begin
    temp := a;
    a := b;
    b := temp;
end;
```

Algoritma 19.18 Prosedur Tukar

Daftar Pustaka

[AHO82] Aho, Alfred V., Hopcroft, John E., Ullman, Jeffrey D., *Data Structures and Algorithms*, Addison-Wesley Publishing Company, 1982.

[AHO87] Aho, Alfred dkk, *Data Structures and Algorithms*, Addison-Wesley Publishing Company, 1987.

[AZM88] Azmoodeh, Manoochehr., *Abstract Data Types and Algorithms*, MacMillan Education LTD, 1988.

[COR89] Cormen, Thomas H., *Introduction to Algorithms*,The MIT Press, 1989

[COR90] Cormen, Thomas H., Leirserson, Charles E., Rivest, Ronald L., *Introduction to Algorithms*, McGraw-Hill Book Company, 1990.

[GOL88] Goldschlager, Les & Lister, Anfrew, *Computer Science, A Modern Introduction, Edisi kedua*, Prentice Hall, 1988.

[KER88] Kernighan, Brian W. & Ritchie, Dennis M., *The Ansi C Programming Language*, Prentice Hall, 1988.

[KNU73] Knuth, Donald E., *The Art of Computer Programming Volume 1*, Addison-Wesley Company, Inc, 1973.

[KUS91] Kusuma, Markus Robijanto, *Belajar Turbo C dengan Cepat dan Mudah*, PT Elex Media Komputindo, 1991.

[LEV03] Levitin, Anany, *Introduction to The Design and Analysis of Algorithms*, Addison-Wesley, 2003.

[LIE96] Liem, Inggriani, *Diktat Kuliah Algoritma dan Pemrograman Prosedural*, Jurusan Teknik Informatika ITB, 1996.

[LIU85] Liu, C.L., *Element of Discrete Mathematics*, McGraw-Hill, 1985.

[MUN99] Munir, Rinaldi, *Diktat Kuliah Pemrograman I, Program D3 Informatika Pos – ITB*, 1999.

[NEA96] Neapolitan, Richard R., *Foundations of Algorithms*, D.C. Heath and Company, 1996.

[NIE93] Nievergelt, Jurg., Hinrichs, Klaus H., *Algorithms & Data Structures with Application to Graphichs and Geometry*, Prentice-Hall, 1993.

[PAR95] Parsons, Thomas W., *Introduction to Algorithms in Pascal*, Johns Wiley and Sons, Inc, 1995.

[ROS99] Rosen, Kenneth H., *Discrete Mathematics and Its Application, Edisi Keempat*, McGraw-Hill, 1999.

[ROH84] Rohl, J.S., *Recursion via Pascal*, Cambridge University Press, 1984.

[SED92] Sedgewick, Robert, *Algorithms in C++*, Addison-Wesley Publishing, 1992.

[TEN86] Tenenbaum, Aaron M., Augenstein, Moshe J., *Data Structures Using Pascal, Prentice-Hall*, 1986.

[WIR76] Wirth, Nicklaus, *Algorithm + Data Structure = Program*, Prentice-hall, 1976.