

Kompleksitas Algoritma

Karena kebodohan kita membuat kesalahan,
dan dari kesalahan kita belajar.
(Anonim)

Algoritma adalah urutan logis langkah-langkah penyelesaian masalah secara sistematis. Sebuah algoritma tidak saja harus benar, tetapi juga harus mangkus (*efisien*). Algoritma yang benar sekalipun mungkin tidak berguna untuk jenis dan ukuran masukan tertentu karena waktu yang diperlukan untuk menjalankan algoritma tersebut atau ruang memori yang diperlukan untuk struktur datanya terlalu besar. Misalnya kita ingin menentukan berapa banyak himpunan bagian dari suatu himpunan yang mengandung elemen x . Jika kita tulis algoritamanya, maka algoritma harus menguji semua himpunan bagian dan memeriksa apakah x merupakan anggota himpunan bagian tersebut. Bila kardinalitas himpunan adalah n , maka algoritma harus menguji sebanyak 2^n himpunan bagian. Semakin besar nilai n , waktu yang diperlukan oleh algoritma tersebut tumbuh sangat cepat. Maka, untuk ukuran masukan yang besar algoritma tersebut menjadi tidak mangkus. Masalah kemangkusan (*efficiency*) algoritma merupakan pokok bahasan bab ini.

10.1 Kemangkusan Algoritma

Algoritma yang bagus adalah algoritma yang mangkus. Kemangkusan algoritma diukur dari berapa jumlah waktu dan ruang (*space*) memori yang dibutuhkan untuk menjalankan. Algoritma yang mangkus ialah algoritma yang meminimumkan kebutuhan waktu dan ruang.

Kebutuhan waktu dan ruang suatu algoritma bergantung pada ukuran masukan, yang secara khas adalah jumlah data, yang diproses. Ukuran masukan itu disimbolkan dengan n . Misalnya, bila mengurutkan 1000 buah elemen larik, maka n adalah 1000; menghitung $6!$ maka $n = 6$; dan lain-lain. Waktu/ruang yang dibutuhkan oleh algoritma dinyatakan sebagai fungsi dari n . Bila n meningkat, maka sumberdaya waktu/ruang yang dibutuhkan juga meningkat. Seberapa besar peningkatan sumberdaya itu menentukan apakah algoritmanya mangkus atau tidak.

Kemangkusan algoritma juga berguna dalam membanding-bandingkan algoritma. Sebuah masalah dapat mempunyai lebih dari satu algoritma penyelesaian. Misalnya, untuk mengurutkan elemen larik (*array*) terdapat beberapa algoritma, seperti algoritmapengurutan apung (*bubble sort*), pengurutan sisipan (*insertion sort*), pengurutan seleksi (*selection sort*), pengurutan gabung (*merge sort*), pengurutan cepat (*quick sort*) dan masih banyak lagi algoritma pengurutan lainnya. Jika algoritma-algoritma tersebut akan dipertimbangkan untuk mengurutkan n buah data, pertanyaan yang sering muncul adalah: bagaimana memilih algoritma yang terbaik untuk diimplementasikan? Untuk menjawabnya, kita memerlukan kriteria formal yang digunakan untuk menilai algoritma yang terbaik. Kriteria itu tidak lain adalah kemangkusan algoritma.

10.2 Mengapa Kita Memerlukan Algoritma yang Mangkus?

Mana yang lebih baik: menggunakan algoritma yang waktu eksekusinya cepat dengan komputer standard ataukah menggunakan algoritma yang waktunya tidak cepat tetapi dengan komputer yang cepat? Pertanyaan semacam ini seringkali muncul ketika seseorang akan memecahkan suatu masalah dengan komputer. Sebagai ilustrasi [BRA88], misalkan untuk menyelesaikan sebuah masalah tertentu telah tersedia:

1. algoritma yang waktu eksekusinya dalam orde eksponensial (2^n), dengan n adalah jumlah masukan yang diproses, dan
2. sebuah komputer yang mampu menjalankan program dengan masukan berukuran n dalam waktu $10^{-4} \times 2^n$ detik.

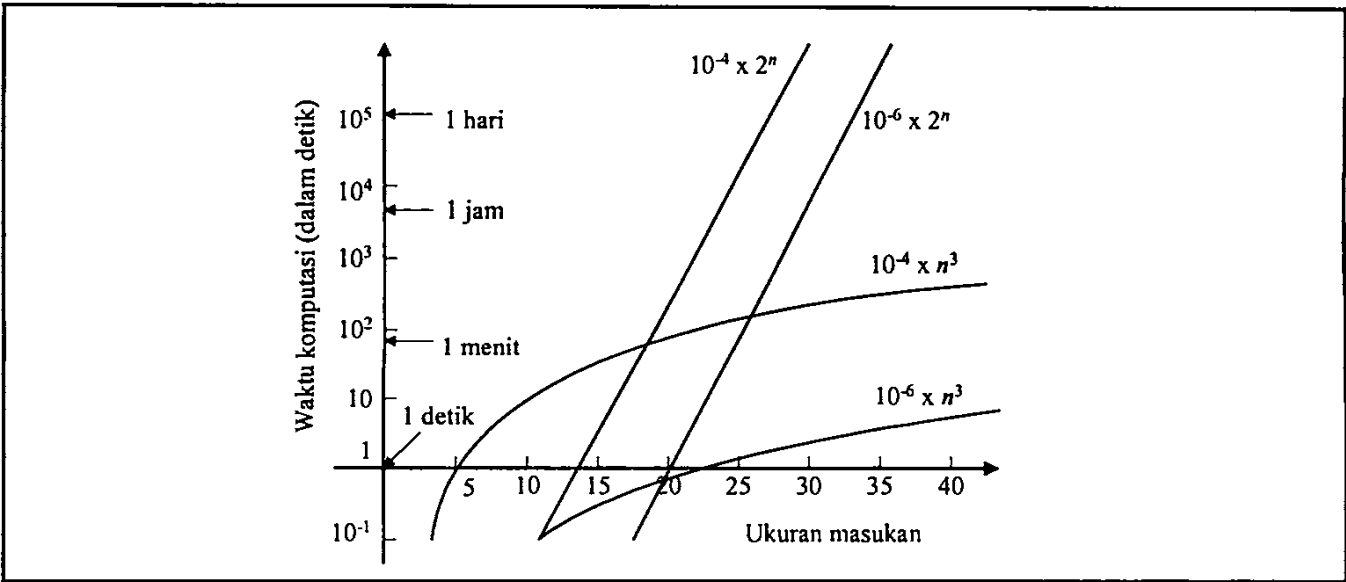
Dengan algoritma dan komputer tersebut, maka dapat dihitung bahwa untuk

- $n = 10$, dibutuhkan waktu eksekusi kira-kira 1/10 detik,
- $n = 20$, dibutuhkan waktu eksekusi kira-kira 2 menit,
- $n = 30$, dibutuhkan waktu eksekusi lebih dari satu hari.

Misalkan kita dapat menjalankan komputer tanpa gangguan selama satu tahun, maka waktu satu tahun itu hanya dapat menyelesaikan persoalan dengan masukan sebanyak 38.

Karena kita perlu menyelesaikan masalah dengan jumlah masukan yang lebih besar, mungkin kita harus membeli mesin baru yang 100 kali lebih cepat daripada komputer semula (menjadi 10^{-6}). Dengan algoritma yang sama, kita sekarang dapat menyelesaikan masalah dengan masukan sebanyak n dalam waktu $10^{-6} \times 2^n$ detik. Bila kita menjalankan mesin baru itu selama satu tahun penuh, kita hanya dapat menyelesaikan persoalan dengan $n = 45$. Secara umum, jika sebelumnya kita dapat menyelesaikan persoalan untuk masukan sebanyak n selama selang waktu tertentu, komputer baru itu akan dapat menyelesaikan persoalan dengan masukan paling banyak $n + 7$ dalam waktu yang sama.

Sebagai gantinya, kita memutuskan untuk menaruh perhatian pada algoritmanya. Misalkan kita menemukan sebuah algoritma baru yang dapat menyelesaikan masalah semula dalam waktu orde kubik (n^3). Bayangkanlah, dengan menggunakan komputer pertama, algoritma baru ini dapat menyelesaikan masalah dengan masukan sebanyak n dalam waktu $10^{-4} \times n^3$ detik. Dalam waktu satu hari kita dapat menyelesaikan masalah dengan jumlah masukan lebih besar dari 900; dan dalam waktu satu tahun komputasi ukuran masukan yang dapat diselesaikan hampir mencapai 6800 lebih. Bahkan dengan komputer yang kedua, jumlah masukan yang dapat diproses selama satu tahun komputasi menjadi lebih banyak lagi, yaitu 31.500 lebih. Hal ini diperlihatkan pada Gambar 10.1 di bawah ini. Jelaslah bahwa algoritma kedua lebih mangkus – yang berarti lebih bagus-dibandingkan dengan algoritma pertama.



Gambar 10.1 Kebutuhan waktu algoritma terhadap ukuran masukan

10.3 Kebutuhan Waktu dan Ruang

Kebutuhan waktu suatu algoritma biasanya dihitung dalam satuan detik, mikrodetik, dan sebagainya, sedangkan ruang memori yang digunakannya dapat dihitung dalam satuan *byte* atau *kilobyte*.

Biasanya orang mengukur kebutuhan waktu sebuah algoritma dengan mengeksekusi langsung algoritma tersebut pada sebuah komputer, lalu dihitung berapa lama durasi waktu yang dibutuhkan untuk menyelesaikan sebuah persoalan dengan n yang berbeda-beda. Keakuratan waktu eksekusi algoritma dapat diperoleh dengan tidak menghitung kebutuhan waktu untuk menampilkan antarmuka program, operasi masukan/keluaran (baca, tulis), dan sebagainya. Jadi, benar-benar yang dihitung adalah kebutuhan waktu untuk bagian algoritma yang inti saja.

Sebagai ilustrasi, tinjau masalah menghitung rata-rata dari n buah data bilangan bulat. Kita mengasumsikan data masukan sudah dibaca dan disimpan di dalam elemen-elemen larik (tabel) a_1, a_2, \dots, a_n . Jadi, kita hanya memperhatikan bagian perhitungan rata-ratanya saja. Bagian perhitungan rata-rata dinyatakan di dalam prosedur *HitungRerata* pada Algoritma 10.1 di bawah ini. Jalankan program yang mengandung prosedur ini pada sebuah komputer. Hitung selisih waktu antara sebelum pemanggilan prosedur dan sesudah pemanggilan prosedur. Selisih kedua waktu ini adalah kebutuhan waktu aktual untuk menghitung rata-rata n buah data.

```
procedure HitungRerata(input  $a_1, a_2, \dots, a_n$  : integer, output  $r$  : real)
{ Menghitung nilai rata-rata dari sekumpulan elemen larik integer  $a_1, a_2, \dots, a_n$ . Nilai rata-rata akan disimpan di dalam peubah  $r$ .
  Masukan:  $a_1, a_2, \dots, a_n$ 
  Keluaran:  $r$  (nilai rata-rata)
}
Deklarasi
   $i$  : integer
  jumlah : real

Algoritma
  jumlah  $\leftarrow$  0
   $i \leftarrow$  1
  while  $i \leq n$  do
    jumlah  $\leftarrow$  jumlah +  $a_i$ 
     $i \leftarrow i + 1$ 
  endwhile
  {  $i > n$  }
   $r \leftarrow$  jumlah/ $n$    ( nilai rata-rata )
```

Algoritma 10.1 Algoritma menghitung nilai rata-rata.

Sayangnya, model perhitungan kebutuhan waktu algoritma seperti di atas ini kurang dapat diterima, karena dua alasan. Alasan pertama, arsitektur komputer yang berbeda menghasilkan waktu yang berbeda pula untuk melaksanakan operasi-operasi dasar (operasi penambahan, perkalian, pembagian, perbandingan, dan sebagainya), sehingga kita tidak mempunyai ukuran kebutuhan waktu yang unik untuk sebuah algoritma. Misalnya, bila anda jalankan sebuah program pada komputer *IBM*, kebutuhan waktunya akan berbeda bila program yang sama dieksekusi pada komputer *Macintosh*.

Alasan pertama ini dapat dijelaskan sebagai berikut. Komputer dengan arsitektur yang berbeda akan berbeda pula perintah (*instruction*) -dalam bahasa mesin tentunya- yang dimilikinya, dan akan berbeda pula kecepatan (*speed*) operasi piranti kerasnya. Dengan demikian, perbedaan di atas akan menghasilkan ukuran waktu (dan kebutuhan ruang memori) yang berbeda-beda pada setiap jenis komputer untuk program yang sama (program adalah realisasi algoritma dalam bahasa tingkat tinggi). Sebagai contoh, komputer tercepat saat ini dapat mengerjakan operasi dasar (seperti penjumlahan, perkalian, pembagian, atau mempertukarkan dua buah bit) dalam waktu 10^{-9} detik, tetapi komputer PC melakukannya dalam 10^{-6} detik, 1000 kali lebih lambat untuk operasi yang sama [ROS99].

Alasan kedua, kebutuhan waktu sebuah algoritma bergantung pada *compiler* bahasa pemrograman yang digunakan. *Compiler* yang berbeda akan menerjemahkan program (dalam bahasa tingkat tinggi) ke dalam kode mesin (*object code* - dalam bahasa tingkat rendah) yang berbeda pula. Sebagai akibatnya, kode mesin yang berbeda akan menggunakan ruang memori dan memerlukan waktu pelaksanaan program yang berbeda pula.

Karena kebutuhan ruang dikaitkan dengan struktur data yang digunakan untuk mengimplementasikan algoritma, sementara topik struktur data di luar bahasan buku ini, maka kebutuhan ruang tidak dibahas di sini.

10.4 Kompleksitas Waktu dan Ruang

Secara teoritis, model abstrak pengukuran waktu/ruang harus independen dari pertimbangan mesin dan *compiler* apapun. Model abstrak seperti itu dapat dipakai untuk membandingkan algoritma yang berbeda. Besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu/ruang ini adalah **kompleksitas algoritma**. Ada dua macam kompleksitas algoritma, yaitu **kompleksitas waktu** dan **kompleksitas ruang**. Kompleksitas waktu diekspresikan sebagai jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n . Kompleksitas ruang diekspresikan sebagai jumlah memori yang digunakan oleh struktur data yang terdapat di dalam algoritma

sebagai fungsi dari ukuran masukan n . Dengan menggunakan besaran kompleksitas waktu/ruang algoritma, kita dapat menentukan *laju* peningkatan waktu (ruang) yang diperlukan algoritma dengan meningkatnya ukuran masukan n .

Terminologi Kompleksitas Waktu/Ruang

Terminologi yang diperlukan dalam membahas kompleksitas waktu dan kompleksitas ruang suatu algoritma adalah:

1. Ukuran besar masukan data untuk suatu algoritma, n .
Sebagai contoh, dalam algoritma pengurutan elemen-elemen larik, n adalah jumlah elemen larik, sedangkan dalam algoritma perkalian matriks n adalah ukuran matriks $n \times n$. Pada beberapa kasus, ukuran masukan lebih tepat menggunakan dua buah besaran, misalnya jika masukan algoritma adalah graf, maka ukuran masukan adalah jumlah simpul dan jumlah sisi.
2. Kompleksitas waktu, $T(n)$, adalah jumlah operasi yang dilakukan untuk melaksanakan algoritma sebagai fungsi dari ukuran masukan n .
3. Kompleksitas ruang, $S(n)$, adalah ruang memori yang dibutuhkan algoritma sebagai fungsi dari ukuran masukan n .

Sebagaimana sudah diungkapkan pada bagian sebelumnya, kita tidak membahas kompleksitas ruang karena kebutuhan ruang dikaitkan dengan struktur data yang digunakan untuk mengimplementasikan algoritma, sementara topik struktur data di luar bahasan buku ini, maka kompleksitas ruang tidak akan kita tinjau. Pertimbangan lain mengapa hanya meninjau kompleksitas waktu adalah tingkat kekritisian memori. Ukuran memori sekarang ini tidak lagi menjadi persoalan kritis, karena komputer sekarang mempunyai ukuran memori yang besar dibandingkan dengan komputer *mainframe* 25 tahun yang lalu. Bahkan, bila memori masih kurang, memori sekunder pun dapat dijadikan sebagai memori tambahan (memori semu, *virtual memory*). Tetapi, ini tidak berarti kita melupakan kompleksitas ruang, hanya saja kompleksitas waktu akan selalu menjadi isu utama dalam merancang suatu algoritma.

Kompleksitas Waktu

Setelah menetapkan ukuran masukan, maka langkah selanjutnya dalam mengukur kompleksitas waktu adalah menghitung banyaknya operasi yang dilakukan oleh algoritma. Di dalam sebuah algoritma mungkin terdapat banyak sekali jenis-jenis operasi, misalnya operasi penjumlahan (termasuk pengurangan), operasi perbandingan, operasi pembagian, operasi pembacaan, pemanggilan prosedur, dan sebagainya.

Contoh 10.1

Sebagai contoh pertama, tinjau kembali Algoritma 10.1. Jenis-jenis operasi yang terdapat di dalam algoritma `HitungRerata` adalah

- operasi pengisian nilai (dengan operator “←”)
- operasi penjumlahan (dengan operator “+”)
- operasi pembagian (dengan operator “/”)

Mari kita hitung kompleksitas waktu algoritma tersebut dengan cara menghitung masing-masing jumlah operasi. Jika operasi tersebut berada di dalam sebuah kalang (*loop*), maka jumlah operasinya bergantung berapa kali kalang tersebut diulang.

(i) Operasi pengisian nilai

Jumlah ← 0,	1 kali
k ← 1,	1 kali
jumlah ← jumlah + a _k ,	n kali
k←k+1,	n kali
r ← jumlah/n),	1 kali

Jumlah seluruh operasi pengisian nilai adalah

$$t_1 = 1 + 1 + n + n + 1 = 3 + 2n$$

(ii) Operasi penjumlahan

jumlah + a _k ,	n kali
k + 1,	n kali

Jumlah seluruh operasi penjumlahan adalah

$$t_2 = n + n = 2n$$

(iii) Operasi pembagian

Jumlah seluruh operasi pembagian adalah

jumlah/n)	1 kali
-----------	--------

Dengan demikian, kompleksitas waktu algoritma dihiutung berdasarjan jumlah operasi aritemetika dan operasi pengisian nilai adalah

$$T(n) = t_1 + t_2 + t_3 = 3 + 2n + 2n + 1 = 4n + 4 \quad \blacksquare$$

Idealnya, kita memang harus menghitung semua operasi yang ada di dalam suatu algoritma, seperti pada Contoh 10.1 di atas. Namun, untuk alasan praktis, kita cukup menghitung jumlah operasi abstrak yang *mendasari* suatu algoritma, dan memisahkan analisisnya dari implementasi [SED92]. Operasi abstrak ini disebut **operasi dasar** (*basic operation*). Sebagai contoh, pada algoritma pencarian, operasi abstrak yang mendasarinya adalah operasi perbandingan *x* dengan elemen-elemen larik. Dengan menghitung berapa kali operasi

perbandingan elemen untuk tiap-tiap nilai n pada dua buah algoritma pencarian, kita memperoleh kemangkusan relatif dari kedua buah algoritma tersebut.

Pada algoritma pengurutan, operasi dasar adalah operasi perbandingan elemen-elemen larik dan operasi pertukaran elemen-elemen. Jadi, kita cukup menghitung berapa kali operasi perbandingan dan berapa kali operasi pertukaran elemen di dalam algoritma pengurutan. Kedua operasi dasar ini dihitung secara terpisah, karena di dalam algoritma pengurutan jumlah operasi perbandingan tidak sama dengan jumlah operasi pertukaran.

Pada algoritma perkalian dua buah matriks $A \times B$ yang masing-masing berukuran $n \times n$, operasi dasar yang bagus untuk dipilih adalah operasi penjumlahan dan perkalian. Jadi, kita cukup menghitung berapa kali operasi penjumlahan dan berapa kali operasi perkalian pada algoritma perkalian dua buah matriks. Kedua operasi dasar ini dihitung secara terpisah.

Pada algoritma evaluasi polinom, $p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$, operasi operasi dasar di dalam algoritmanya juga operasi penjumlahan dan perkalian. Jadi, kita cukup menghitung berapa kali operasi penjumlahan dan berapa kali operasi perkalian pada algoritma evaluasi polinom.

Contoh 10.2

Tinjau kembali algoritma HitungRerata pada Contoh 10.1. Operasi yang mendasar pada algoritma menghitung rata-rata adalah operasi penjumlahan elemen-elemen larik (yaitu operasi di dalam instruksi $\text{jumlah} \leftarrow \text{jumlah} + a_k$), yang mana dilaksanakan sebanyak n kali, yaitu sejumlah pengulangan yang dilakukan. Operasi-operasi lainnya, seperti pembagian) boleh kita abaikan (tidak dihitung). Jika kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi penjumlahan ini, maka kompleksitas waktu HitungRerata adalah $T(n) = n$. ■

Contoh 10.3

Tinjau algoritma untuk mencari elemen terbesar di dalam sebuah larik (*array*) yang berukuran n elemen. Tentukan kompleksitas waktu algoritma, yang dalam hal ini kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi perbandingan elemen-elemen, karena perbandingan adalah operasi dasar yang digunakan di dalam algoritma mencari nilai terbesar (operasi perbandingan $i \leq n$ bukan operasi perbandingan elemen larik, jadi tidak kita hitung).

Penyelesaian:

Operasi perbandingan elemen larik yang dimaksudkan di dalam algoritma adalah $A[i] > \text{maks}$. Operasi ini terdapat di dalam kalang *for*. Jumlah operasi perbandingan elemen ditentukan oleh berapa kali kalang *for* dieksekusi, yaitu $n - 1$ kali. Dengan demikian, kompleksitas waktu algoritma CariMaks adalah $T(n) = n - 1$. ■

```

procedure CariMaks(input  $a_1, a_2, \dots, a_n$  : integer, output maks : integer)
{ Mencari elemen terbesar dari sekumpulan elemen larik integer  $a_1, a_2, \dots, a_n$ . Elemen terbesar akan disimpan di dalam maks.
Masukan:  $a_1, a_2, \dots, a_n$ 
Keluaran: maks (nilai terbesar)
}

```

Deklarasi

i : integer

Algoritma

```

maks  $\leftarrow a_1$ 
 $i \leftarrow 2$ 
while  $i \leq n$  do
  if  $a_i > \text{maks}$  then
    maks  $\leftarrow a_i$ 
  endif
   $i \leftarrow i + 1$ 
endwhile
{  $i > n$  }

```

Algoritma 10.2 Algoritma mencari elemen terbesar

Notasi kompleksitas waktu telah membebaskan kita dari pertimbangan spesifikasi komputer untuk menghitung kebutuhan waktu algoritma. Kita tidak peduli komputer apa yang digunakan untuk menjalankan algoritma, namun yang pasti apapun komputer atau bahasa pemrograman yang digunakan, jumlah komputasi di dalam algoritma tersebut tetap, yaitu $T(n)$.

Andaikan kita mengetahui informasi mengenai waktu yang dibutuhkan untuk melakukan operasi tertentu pada komputer tertentu, kita dapat menghitung kebutuhan waktu aktual yang sesungguhnya untuk sebuah algoritma. Misalnya pada algoritma CariMaks diandaikan satu kali operasi perbandingan di dalam komputer PC membutuhkan waktu 10^{-6} detik, maka untuk masukan sebanyak 1000 elemen, kebutuhan waktu algoritma CariMaks dihitung berdasarkan operasi perbandingan elemen saja, adalah $T(1000) = (1000 - 1) \times 10^{-6} = 0.000999$ detik. Bila algoritma algoritma CariMaks dijalankan pada komputer tercepat saat ini, maka kebutuhan waktu algoritma, dihitung berdasarkan operasi perbandingan elemen saja, adalah $T(1000) = (1000 - 1) \times 10^{-9} = 0.000000999$ detik. Apabila operasi-operasi selain perbandingan juga dihitung, kita akan memperoleh kebutuhan waktu yang lebih presisi. Namun, kita tidak melakukan perhitungan kebutuhan waktu aktual di sini. Kompleksitas waktu $T(n) = n - 1$ sudah cukup memberikan informasi mengenai unjuk kerja algoritma. Dengan kata lain, kita menghitung kebutuhan waktu algoritma secara teoritis, bukan secara praktis.

Hal lain yang harus diperhatikan dalam menghitung kompleksitas waktu adalah parameter yang mencirikan ukuran masukan. Pada algoritma pencarian misalnya,

waktu pencarian tidak hanya bergantung pada ukuran larik (n), tetapi juga bergantung pada nilai elemen yang dicari (x). Sebagai contoh, diketahui larik bilangan bulat yang beranggotakan 128 buah elemen, a_1, a_2, \dots, a_n . Asumsikan elemen-elemen larik sudah terurut. Jika $a_1 = x$, maka waktu pencariannya 128 kali lebih cepat daripada jika $a_{128} = x$ atau jika x tidak terdapat di dalam larik. Demikian pula, jika $a_{64} = x$, maka waktu pencariannya 1/2 kali lebih cepat daripada jika $a_{128} = x$. Karena itu, kompleksitas waktu dibedakan atas tiga macam :

1. $T_{max}(n)$: kompleksitas waktu untuk kasus terburuk (*worst case*), yaitu kebutuhan waktu maksimum yang diperlukan sebuah algoritma sebagai fungsi dari n
2. $T_{min}(n)$: kompleksitas waktu untuk kasus terbaik (*best case*), yaitu kebutuhan waktu minimum yang diperlukan sebuah algoritma sebagai fungsi dari n
3. $T_{avg}(n)$: kompleksitas waktu untuk kasus rata-rata (*average case*) yaitu kebutuhan waktu rata-rata yang diperlukan algoritma sebagai fungsi dari n . Untuk kasus rata-rata ini, biasanya dibuat asumsi bahwa semua barisan masukan bersifat sama. Misalnya, pada persoalan pencarian diandaikan bahwa data yang dicari mempunyai peluang yang sama untuk terletak di dalam larik.

Contoh 10.4

Diberikan larik bilangan bulat a_1, a_2, \dots, a_n yang telah terurut menaik (tidak ada elemen ganda). Hitunglah kompleksitas waktu terbaik, terburuk, dan rata-rata dari algoritma pencarian beruntun (*sequential search*) di bawah ini. Algoritma pencarian beruntun di bawah ini menghasilkan indeks elemen yang bernilai sama dengan x . Jika x tidak ditemukan, indeks 0 akan dihasilkan (catatlah bahwa algoritma pencarian beruntun ini merupakan versi lain dari algoritma pencarian beruntun yang pernah dituliskan pada Algoritma 5.4 di dalam Bab 4).

```

procedure PencarianBeruntun(input  $a_1, a_2, \dots, a_n$  : integer,  $x$  : integer,
                           output  $idx$  : integer)
( Mencari  $x$  di dalam elemen  $a_1, a_2, \dots, a_n$ . Lokasi (indeks elemen) tempat
   $x$  ditemukan diisi ke dalam  $idx$ . Jika  $x$  tidak ditemukan, maka  $idx$  diisi
  dengan 0.
  Masukan:  $a_1, a_2, \dots, a_n$ 
  Keluaran:  $idx$ 
)
Deklarasi
   $i$  : integer
  ketemu : boolean    ( bernilai true jika  $x$  ditemukan atau false jika  $x$ 
                      tidak ditemukan )
Algoritma:
   $i \leftarrow 1$ 

```

```

ketemu ← false
while (i ≤ n) and (not ketemu) do
  if ai = x then
    ketemu ← true
  else
    i ← i + 1
  endif
endwhile
{ i > n or ketemu }

if ketemu then { x ditemukan }
  idx ← i
else
  idx ← 0      { x tidak ditemukan }
endif

```

Algoritma 10.3 Algoritma pencarian beruntun.

Penyelesaian:

Algoritma PencarianBeruntun membandingkan setiap elemen larik dengan x , mulai dari elemen pertama sampai x ditemukan atau sampai elemen terakhir. Jika x ditemukan, maka proses pencarian dihentikan. Kita akan menghitung jumlah operasi perbandingan elemen larik yang terjadi selama pencarian ($a_i = x$). Operasi perbandingan yang lain, seperti $i \leq n$ tidak akan dihitung. Operasi perbandingan elemen-elemen larik adalah operasi abstrak yang mendasari algoritma pencarian.

1. *Kasus terbaik*: ini terjadi bila $a_1 = x$.
Operasi perbandingan elemen ($a_i = x$) hanya dilakukan satu kali, maka

$$T_{\min}(n) = 1$$

2. *Kasus terburuk*: bila $a_n = x$ atau x tidak ditemukan.
Seluruh elemen larik dibandingkan, maka jumlah perbandingan elemen larik ($a_i = x$) adalah

$$T_{\max}(n) = n$$

3. *Kasus rata-rata*: Jika x ditemukan pada posisi ke- j , maka operasi perbandingan ($a_i = x$) dilakukan sebanyak j kali. Jadi, kebutuhan waktu rata-rata algoritma pencarian beruntun adalah

$$T_{\text{avg}}(n) = \frac{(1+2+3+\dots+n)}{n} = \frac{\frac{1}{2}n(1+n)}{n} = \frac{(n+1)}{2}$$

Cara lain yang dapat digunakan dalam menghitung T_{avg} di atas adalah sbb: asumsikan bahwa peluang x terdapat di sembarang lokasi larik adalah sama, artinya, peluang elemen ke- $j = x$ adalah $1/n$, atau kita tulis $P(a_j = x) = 1/n$. Jika $a_j = x$ maka T_j yang dibutuhkan adalah $T_j = j$. Jumlah perbandingan elemen larik secara rata-rata adalah:

$$T_{\text{avg}}(n) = \sum_{j=1}^n T_j P(A[j] = X) = \sum_{j=1}^n T_j \frac{1}{n} = \frac{1}{n} \sum_{j=1}^n T_j = \frac{1}{n} \sum_{j=1}^n j = \frac{1}{n} \left(\frac{n(n+1)}{2} \right) = \frac{n+1}{2}$$

■

Contoh 10.5

Diberikan larik bilangan bulat a_1, a_2, \dots, a_n yang telah terurut menaik (tidak ada elemen ganda). Hitunglah kompleksitas waktu terbaik, terburuk, dan rata-rata dari algoritma pencarian biner (*binary search*) di bawah ini. Algoritma pencarian beruntun di bawah ini menghasilkan indeks elemen yang bernilai sama dengan x . Jika x tidak ditemukan, indeks 0 akan dihasilkan

```

procedure PencarianBiner(input  $a_1, a_2, \dots, a_n$  : integer,  $x$  : integer,
                        output  $\text{idx}$  : integer)
{ Mencari  $x$  di dalam elemen  $a_1, a_2, \dots, a_n$ . Lokasi (indeks elemen) tempat
   $x$  ditemukan diisi ke dalam  $\text{idx}$ . Jika  $x$  tidak ditemukan, maka  $\text{idx}$  diisi
  dengan 0.
  Masukan:  $a_1, a_2, \dots, a_n$ 
  Keluaran:  $\text{idx}$ 
}
Deklarasi
   $i, j, \text{mid}$  : integer
   $\text{ketemu}$  : boolean

Algoritma
   $i \leftarrow 1$ 
   $j \leftarrow n$ 
   $\text{ketemu} \leftarrow \text{false}$ 
  while (not  $\text{ketemu}$ ) and ( $i \leq j$ ) do
     $\text{mid} \leftarrow (i + j) \text{ div } 2$ 
    if  $a_{\text{mid}} = x$  then
       $\text{ketemu} \leftarrow \text{true}$ 
    else
      if  $a_{\text{mid}} < x$  then      { cari di belahan kanan }
         $i \leftarrow \text{mid} + 1$ 
      else                    { cari di belahan kiri }
         $j \leftarrow \text{mid} - 1$ 
      endif
    endif
  endwhile
  { $\text{ketemu}$  or  $i > j$  }

  if  $\text{ketemu}$  then
     $\text{idx} \leftarrow \text{mid}$ 
  else
     $\text{idx} \leftarrow 0$ 
  endif

```

Algoritma 10.4 Algoritma pencarian biner

Penyelesaian:

Algoritma PencarianBiner membagi larik di pertengahan menjadi dua bagian yang berukuran sama ($n/2$ bagian), bagian kiri dan bagian kanan. Jika elemen pertengahan

tidak sama dengan x , keputusan dibuat untuk melakukan pencarian pada bagian kiri atau bagian kanan. Proses bagi-dua dilakukan lagi pada bagian yang dipilih. Perhatikanlah bahwa setiap kali memasuki kalang *while-do* maka ukuran larik yang ditelusuri berkurang menjadi setengah kali ukuran semula: $n, n/2, n/4, \dots$

Kita akan menghitung jumlah operasi perbandingan elemen dengan x yang terjadi selama pencarian ($a_{mid} = x$). Operasi perbandingan yang lain, seperti $i \leq j$ dan $a_{mid} < x$ tidak akan dihitung. Untuk penyederhanaan, asumsikan ukuran larik adalah perangkatan dari 2 (yaitu, $n = 2^k$).

1. Kasus terbaik

Kasus terbaik adalah bila x ditemukan pada elemen pertengahan (a_{mid}), dan operasi perbandingan elemen ($a_{mid} = x$) yang dilakukan hanya satu kali. Pada kasus ini

$$T_{min}(n) = 1$$

2. Kasus terburuk:

Pada kasus terburuk, elemen x ditemukan ketika ukuran larik = 1. Pada kasus terburuk ini, ukuran larik setiap kali memasuki kalang *while-do* adalah:

$$n, n/2, n/4, n/8, \dots, 1 \quad (\text{sebanyak } {}^2\log n \text{ kali})$$

artinya, kalang *while-do* dikerjakan sebanyak ${}^2\log n$ kali.

Contoh: $n = 128 \Rightarrow 128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ (sebanyak ${}^2\log 128 = 7$ kali pembagian)

Jumlah operasi perbandingan elemen ($a_{mid} = x$) adalah:

$$T_{max}(n) = {}^2\log n$$

Kompleksitas waktu rata-rata algoritma pencarian bagidua lebih sulit ditentukan. ■

Kadang-kadang penentuan kasus berguna untuk menghitung jumlah operasi yang bergantung pada kondisi tertentu. Misalnya pada potongan algoritma berikut kita ingin menghitung nilai rata-rata elemen larik yang ganjil:

```
k ← 1
jumlah ← 0
for k ← 1 to n do
  if ak mod 2 = 1 then ( ak ganjil )
    jumlah ← jumlah + ak
  endif
endfor
```

Misalkan kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi penjumlahan elemen (yaitu jumlah + a_k). Instruksi $\text{jumlah} \leftarrow \text{jumlah} + a_k$ hanya dikerjakan jika kondisi $a_k \bmod 2 = 1$ benar (yaitu jika a_k ganjil).

Meskipun demikian, kita tetap dapat menghitung kompleksitas waktu algoritma dengan mengasumsikan bahwa pada kasus terburuk semua elemen bernilai ganjil sehingga kondisi $a_k \bmod 2 = 1$ terpenuhi dan instruksi penjumlahan $\text{jumlah} \leftarrow \text{jumlah} + a_k$ dikerjakan sebanyak n kali. Jadi, pada kasus terburuk,

$$T_{\max}(n) = n$$

dan pada kasus terbaik, semua elemen genap sehingga instruksi penjumlahan $\text{jumlah} \leftarrow \text{jumlah} + a_k$ tidak pernah dikerjakan, jadi

$$T_{\min}(n) = 0$$

Contoh 10.6

Hitung jumlah operasi perbandingan elemen larik dan jumlah operasi pertukaran pada algoritma pengurutan seleksi (*selection sort*).

```

procedure UrutSeleksi(input/output  $a_1, a_2, \dots, a_n$  : integer)
{ Mengurutkan elemen-elemen  $a_1, a_2, \dots, a_n$  dengan metode selection sort.
  Masukan:  $a_1, a_2, \dots, a_n$ .
  Keluaran:  $a_1, a_2, \dots, a_n$  (sudah terurut menaik).
}

Deklarasi
  i, j, imaks, temp : integer

Algoritma
  for i ← n downto 2 do { pass sebanyak n - 1 kali }
    imaks ← 1
    for j ← 2 to i do
      if  $a_j > a_{\text{imaks}}$  then
        imaks ← j
      endif
    endfor
    { pertukarkan  $a_{\text{imaks}}$  dengan  $a_i$  }
    temp ←  $a_i$ 
     $a_i \leftarrow a_{\text{imaks}}$ 
     $a_{\text{imaks}} \leftarrow \text{temp}$ 
  endfor

```

Algoritma 10.5 Algoritma pengurutan (*selection sort*)

Penyelesaian:

Algoritma UrutSeleksi terdiri dari $n - 1$ kali *pass*. Pada setiap *pass*, kita mencari elemen terbesar dari elemen-elemen a_1, a_2, \dots, a_n , lalu mempertukarkan elemen terbesar dengan a_n . *Pass* berikutnya akan mencari elemen terbesar dari dari sekumpulan a_1, a_2, \dots, a_{n-1} , begitu seterusnya sampai larik *pass* terakhir sehingga tinggal satu elemen yang pasti sudah terurut. Operasi abstrak yang mendasari algoritma pengurutan adalah operasi

perbandingan elemen larik ($a_j > a_{\text{maks}}$) dan operasi pertukaran (diwakili oleh tiga buah instruksi: $\text{temp} \leftarrow a_n$, $a_n \leftarrow a_{\text{maks}}$, $a_{\text{maks}} \leftarrow \text{temp}$). Kedua operasi ini kita pisahkan perhitungannya sebagai berikut:

(i) Jumlah operasi perbandingan elemen

Untuk setiap *pass* ke- i , $i = n, n - 1, \dots, 2$, operasi perbandingan elemen yang dilakukan adalah sebagai berikut:

$$\begin{aligned} i = n &\rightarrow \text{jumlah operasi perbandingan elemen} = n - 1 \\ i = n - 1 &\rightarrow \text{jumlah operasi perbandingan elemen} = n - 2 \\ i = n - 2 &\rightarrow \text{jumlah operasi perbandingan elemen} = n - 3 \\ &\vdots \\ i = 2 &\rightarrow \text{jumlah operasi perbandingan elemen} = 1 \end{aligned}$$

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah

$$T(n) = (n - 1) + (n - 2) + \dots + 1 = \sum_{k=1}^{n-1} n - k = \frac{n(n-1)}{2}$$

Ini adalah kompleksitas waktu untuk kasus terbaik dan terburuk, karena algoritma UrutSeleksi tidak bergantung pada data masukan apakah sudah terurut atau acak.

(ii) Jumlah operasi pertukaran

Untuk setiap i dari n sampai 2, terjadi satu kali pertukaran elemen, sehingga jumlah operasi pertukaran seluruhnya adalah

$$T(n) = n - 1.$$

Jadi, algoritma pengurutan seleksi membutuhkan $n(n - 1)/2$ buah operasi perbandingan elemen dan $n - 1$ buah operasi pertukaran. ■

Contoh 10.7

Hitung jumlah operasi perkalian pada algoritma yang menghitung $\sum_{j=n, \lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \dots}^1 \sum_{i=1}^j xi$.

Asumsikan n adalah perpangkatan dari 2.

```

procedure Kali(input x:integer, n:integer, output jumlah : integer)
  (Mengalikan x dengan i = 1, 2, ..., j, yang dalam hal ini j = n, n/2, n/4, ..., 1
  Masukan: x dan n (n adalah perpangkatan dua).
  Keluaran: hasil perkalian (disimpan di dalam peubah jumlah).
)
Deklarasi
  i, j, k : integer

Algoritma
  j ← n
  while j > 1 do
    for i ← 1 to j do
      x ← x * i
  
```

```

endfor
j ← d div 2
endwhile
{ j > 1 }
jumlah ← x

```

Algoritma 10.6 Algoritma perkalian

Penyelesaian:

Untuk

$j = n$, jumlah operasi perkalian = n ;
 $j = n/2$, jumlah operasi perkalian = $n/2$;
 $j = n/4$, jumlah operasi perkalian = $n/4$;
 \dots
 $j = 1$, jumlah operasi perkalian = 1

Jumlah operasi perkalian seluruhnya adalah

$$n + n/2 + n/4 + \dots + 2 + 1$$

yang merupakan deret geometri dengan jumlah = $\frac{n(1 - 2^{-\log n})}{1 - \frac{1}{2}} = 2(n - 1)$ ■

10.5 Kompleksitas Waktu Asimptotik

Seringkali kita kurang tertarik dengan kompleksitas waktu yang presisi untuk suatu algoritma, tetapi kita lebih tertarik pada bagaimana waktu terbaik dan waktu terburuk tumbuh bersamaan dengan meningkatnya ukuran masukan. Sebagai contoh, pada Algoritma 10.5, jumlah operasi perbandingan elemen adalah

$$T(n) = n(n - 1)/2$$

Kita mungkin tidak terlalu membutuhkan informasi seberapa tepat jumlah operasi perbandingan elemen pada algoritma pengurutan tersebut. Yang kita butuhkan adalah perkiraan kasar kebutuhan waktu algoritma dan seberapa cepat fungsi kebutuhan waktu itu tumbuh. Hal ini perlu untuk mengetahui kinerja algoritma. Kinerja algoritma baru akan tampak untuk n yang sangat besar, bukan pada n yang kecil. Bila anda menggunakan komputer untuk menjalankan algoritma PencarianBeruntun dan PencarianBiner untuk larik yang berukuran kecil (misalnya $n = 10$), maka perbedaan kecepatan keduanya tidak akan terlihat. Tetapi, bila kedua algoritma tersebut diterapkan untuk larik yang berukuran besar (misalnya $n = 5000$), perbedaan kecepatan keduanya akan terlihat sangat berarti.

Langkah pertama dalam pengukuran kinerja algoritma adalah membuat makna “sebanding”. Gagasannya adalah dengan menghilangkan faktor koefisien di dalam ekspresi $T(n)$. Sebagai contoh, andaikan bahwa kompleksitas waktu terburuk dari sebuah algoritma adalah

$$T(n) = 2n^2 + 6n + 1$$

Untuk n yang besar, pertumbuhan $T(n)$ sebanding dengan n^2 (lihat Tabel 10.1). Pada kasus ini, $T(n)$ tumbuh seperti n^2 tumbuh.

Tabel 10.1 Perbandingan pertumbuhan $T(n)$ dengan n^2

n	$T(n) = 2n^2 + 6n + 1$	n^2
10	261	100
100	2061	1000
1000	2.006.001	1.000.000
10.000	2.000.060.001	1.000.000.000

Jika diperhatikan pada Tabel 10.1 tersebut, suku $6n + 1$ menjadi tidak berarti dibandingkan $2n^2$. Kita dapat mengabaikan suku-suku yang tidak mendominasi perhitungan pada rumus $T(n)$, sehingga kompleksitas waktu $T(n)$ adalah

$$2(n^2) + \text{suku-suku lainnya}$$

Dengan mengabaikan koefisien 2 pada $2n^2$, kita melihat $T(n)$ tumbuh seperti n^2 tumbuh saat n bertambah. Kita katakan bahwa $T(n)$ berorde n^2 dan kita tuliskan

$$T(n) = O(n^2)$$

yang dibaca “ $T(n)$ adalah O dari n^2 ”. Jadi, kita telah mengganti ekspresi seperti $T(n) = 2n^2 + 6n + 1$ dengan ekspresi yang lebih sederhana seperti n^2 yang tumbuh pada kecepatan yang sama dengan $T(n)$. Notasi “ O ” disebut notasi “ O -Besar” (*Big-O*) yang merupakan salah satu dari tiga notasi **kompleksitas waktu asimptotik**. Definisi formal notasi O -Besar dituliskan di bawah ini.

Notasi O -Besar

DEFINISI 10.1. $T(n) = O(f(n))$ (dibaca “ $T(n)$ adalah $O(f(n))$ ” yang artinya $T(n)$ berorde paling besar $f(n)$) bila terdapat konstanta C dan n_0 sedemikian sehingga

$$T(n) \leq C \cdot f(n)$$

untuk $n \geq n_0$.

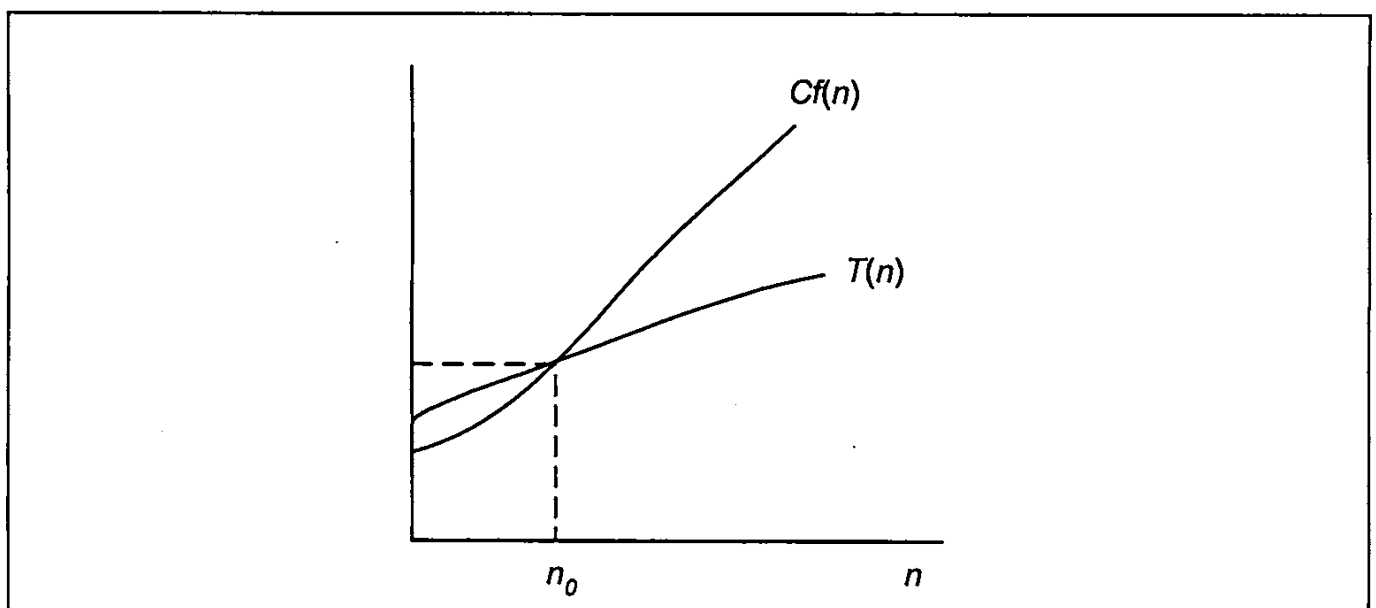
Makna notasi O -besar adalah jika sebuah algoritma mempunyai waktu asimptotik $O(f(n))$, maka jika n dibuat semakin besar, waktu yang dibutuhkananya tidak akan pernah melebihi suatu konstanta C dikali dengan $f(n)$. Jadi, $f(n)$ adalah batas lebih atas (*upper bound*) dari $T(n)$ untuk n yang besar. Kita katakan $T(n)$ berorde paling besar $f(n)$.

Gambar 10.2 memperlihatkan tafsiran geometri “ O besar”. Meskipun $T(n)$ pada mulanya berada di atas $Cf(n)$, tapi setelah $n = n_0$ ia selalu berada di bawah $Cf(n)$. Contoh nyatanya adalah $T(n) = n^2 + 5n$, meskipun pada awalnya kurva berada di atas $2n^2$, tetapi untuk $n \geq 5$

$$n^2 + 5n \leq 2n^2$$

Karena itu kita mengambil $C = 2$ dan $n_0 = 5$ untuk memperlihatkan bahwa

$$n^2 + 5n = O(n^2)$$



Gambar 10.2 Ilustrasi “ O besar”

Dari definisi O -Besar jelas menuliskan $T(n) = O(f(n))$ tidak sama dengan $O(f(n)) = T(n)$. Lagi pula, tidaklah bermakna apa-apa menyatakan $O(f(n)) = T(n)$. Penggunaan simbol “=” tidak menguntungkan karena simbol ini sudah umum menyatakan “kesamaan”. Kebingungan yang timbul dari penggunaan simbol ini dapat dihindari dengan membaca simbol “=” sebagai “adalah” dan bukan “sama dengan” [HOR90].

Untuk menunjukkan bahwa $T(n) = O(f(n))$ kita hanya perlu menemukan pasangan C dan n_0 sedemikian sehingga $T(n) \leq C(f(n))$. Tetapi, perlu diingat bahwa pasangan C dan n_0 yang memenuhi definisi di atas tidak unik. Ada banyak C dan

n_0 yang memenuhi definisi ini. Contoh-contoh berikut memperlihatkan cara memperoleh notasi kompleksitas asimptotik untuk bermacam-macam $T(n)$.

Contoh 10.8

Tunjukkan bahwa $T(n) = 2n^2 + 6n + 1 = O(n^2)$.

Penyelesaian:

Kita mengamati bahwa jika $n \geq 1$ maka $n \leq n^2$ dan $1 \leq n^2$ sehingga

$$2n^2 + 6n + 1 \leq 2n^2 + 6n^2 + n^2 = 9n^2 \text{ untuk semua } n \geq 1.$$

Jadi, kita dapat mengambil $C = 9$ dan $n_0 = 1$ untuk memperlihatkan bahwa

$$T(n) = 2n^2 + 6n + 1 = O(n^2).$$

Perhatikan bahwa $C = 9$ dan $n_0 = 1$ bukan satu-satunya pasangan nilai yang dapat kita gunakan untuk memenuhi Definisi 10.1. Kita juga dapat menyatakan bahwa

$$2n^2 + 6n + 1 \leq 6n^2 \text{ untuk semua } n \geq 2$$

Jadi, kita mengambil $C = 6$ dan $n_0 = 2$ untuk memperlihatkan bahwa

$$T(n) = 2n^2 + 6n + 1 = O(n^2).$$

Alternatif lain adalah dengan mengamati bahwa $6n \leq n^2$ untuk $n \geq 6$. Jadi, kita menyatakan bahwa

$$2n^2 + 6n + 1 = O(n^2)$$

karena

$$2n^2 + 6n + 1 \leq n^2 + n^2 + n^2 = 3n^2 \text{ untuk semua } n \geq 6$$

(di sini $C = 3$ dan $n_0 = 6$). ■

Contoh 10.9

Tunjukkan bahwa $T(n) = 5 = O(1)$.

Penyelesaian:

$5 = O(1)$ karena $5 \leq 6 \cdot 1$ untuk $n \geq 1$. Kita juga dapat memperlihatkan bahwa $5 = O(1)$ karena $5 \leq 10 \cdot 1$ untuk $n \geq 1$ ■

Contoh 10.10

Tunjukkan bahwa $T(n) = 3n + 2 = O(n)$.

9. $T(n) = 5n \log n = O(n \log n)$ $\{ 5n \log n \leq 6n \log n \text{ untuk semua } n \geq 1 \}$
10. $T(n) = 2n + 3^2 \log n = O(n)$ $\{ 2n + 3^2 \log n \leq 2n + 3n = 5n \text{ untuk } n \geq 1, \text{ karena } 2^2 \log n < n \text{ untuk } n \geq 1 \}$
11. $T(n) = n = O(n^2)$ $\{ n \leq 1 \cdot n^2 \text{ untuk semua } n \geq 1 \}$
12. $T(n) = n! = O(n^n)$ $\{ n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n \leq n \cdot n \cdot n \cdot \dots \cdot n = n^n \text{ untuk } n \geq 1 \}$
13. $T(n) = \log n! = O(n \log n)$ $\{ \log n! \leq \log n^n = n \log n$
atau dengan cara lain:

$$\log n! = \log(n \times (n-1) \times \dots \times 2 \times 1)$$

$$= \log n + \log(n-1) + \dots + \log 2 + 1 \leq \log n + \log n + \dots$$

$$+ \log n + \log n = n \log n \}$$
14. $T(n) = 21 + 1/n = O(n)$ $\{ 21 + 1/n \leq 22n \text{ untuk semua } n \geq 1 \}$
15. $T(n) = 3n + 2 \neq O(1)$ karena tidak ada C dan n_0 sedemikian sehingga $3n + 2 \leq C \cdot 1$ untuk $n \geq n_0$.
16. $T(n) = 10n^2 + 4n + 2 \neq O(n)$ karena tidak ada C dan n_0 sedemikian sehingga $10n^2 + 4n + 2 \leq C \cdot n$ untuk $n \geq n_0$

Polinomial n derajat m dapat digunakan untuk memperkirakan kompleksitas waktu asimptotik. Suku berorde rendah dalam ekspresi $T(n)$ dapat diabaikan dalam menentukan orde keseluruhan. Jadi, $T(n) = 3n^3 + 6n^2 + n + 8 = O(n^3)$, $T(n) = 2n^2 + 6n + 1 = O(n^2)$, $T(n) = \frac{1}{2} n^2 = O(n^2)$. Hal ini dinyatakan dengan teorema berikut [HOR90]:

TEOREMA 10.1. Bila $T(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ adalah polinom derajat m maka $T(n) = O(n^m)$.

Bukti:

$$\begin{aligned} T(n) &\leq \sum_{i=0}^m a_i n^i \\ &\leq n^m \sum_{i=0}^m a_i n^{i-m} \\ &\leq n^m \sum_{i=0}^m a_i = Cn^m, \text{ untuk } n \geq 1, \text{ yang dalam hal ini } C = \sum_{i=0}^m a_i. \end{aligned}$$

Oleh karena itu

$$T(n) = O(n^m).$$

■

Teorema 10.1 menyatakan bahwa suku yang berderajat lebih tinggi mendominasi suku yang lebih rendah. “Mendominasi” diartikan bahwa “laju pertumbuhannya lebih cepat daripada”; $f(n)$ mendominasi $T(n)$ jika $T(n)$ adalah $O(f(n))$. Hal ini selalu terjadi, artinya, selalu ada suku yang dominan untuk n yang besar. Dalam menentukan kompleksitas waktu asimptotik, perhatian kita selalu tertuju pada suku yang dominan itu.

Besaran dominan lainnya adalah:

- eksponensial mendominasi sembarang perpangkatan (yaitu, $y^n > n^p$, $y > 1$),
- perpangkatan mendominasi $\ln n$ (yaitu $n^p > \ln n$),
- semua logaritma tumbuh pada laju yang sama (yaitu $^a\log(n) = ^b\log(n)$),
- $n \log n$ tumbuh lebih cepat daripada n tetapi lebih lambat daripada n^2 .

Perhatikan bahwa kita juga dapat menyatakan bahwa $2n^2 + 6n + 1 = O(n^3)$ karena $2n^2 + 6n + 1 \leq 9n^3$ untuk semua $n \geq 1$, atau $2n^2 + 6n + 1 \leq 2n^3$ untuk semua $n \geq 4$. Dengan cara yang sama kita juga dapat menyatakan $2n^2 + 6n + 1 = O(n^4)$, dan seterusnya. Jadi, bila $T(n) = O(f(n))$, dan $g(n)$ adalah fungsi yang nilainya lebih besar dari $f(n)$, maka $T(n) = O(g(n))$. Ini berarti, fungsi $f(n)$ pada notasi $T(n) = O(f(n))$ dapat diganti dengan fungsi lain yang lebih besar [ROS99]. Namun, agar notasi O -besar memiliki makna, maka untuk alasan lebih praktis, fungsi f di dalam notasi $T(n) = O(f(n))$ dipilih fungsi yang sekecil mungkin, biasanya dari fungsi acuan seperti 1, $\log n$, n , $n \log n$, n^2 , n^3 , ..., 2^n , $n!$. Jadi, jika suatu algoritma memiliki kebutuhan waktu $T(n) = 2n^2 + 6n + 1$, kompleksitas waktunya ditulis $O(n^2)$, bukan $O(n^3)$, $O(n^4)$, dan seterusnya.

Teorema O-Besar

TEOREMA 10.2. Misalkan $T_1(n) = O(f(n))$ dan $T_2(n) = O(g(n))$, maka

- (a) (i) $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
(ii) $T_1(n) + T_2(n) = O(f(n) + g(n))$
- (b) $T_1(n)T_2(n) = O(f(n))O(g(n)) = O(f(n)g(n))$
- (c) $O(cf(n)) = O(f(n))$, c adalah konstanta
- (d) $f(n) = O(f(n))$

Perhatikan di dalam Teorema 10.2 bahwa bahwa $T_1(n) + T_2(n)$ diberikan dalam dua rumus yang berbeda, hal ini karena notasi O Besar adalah mekanisme untuk menemukan batas lebih atas untuk laju pertumbuhan kompleksitas waktu algoritma dan bukan batas lebih atas yang paling kecil [AZM88]. Keduanya digunakan dalam konteks yang berbeda.

Contoh 10.14

Misalkan $T_1(n) = O(n)$, $T_2(n) = O(n^2)$, dan $T_3(n) = O(mn)$, dengan m adalah peubah maka

- (a) $T_1(n) + T_2(n) = O(\max(n, n^2)) = O(n^2)$ (Teorema 10.2(a)(i))
 - (b) $T_2(n) + T_3(n) = O(n^2 + mn)$ (Teorema 10.2(a)(ii))
 - (c) $T_1(n)T_2(n) = O(n \cdot n^2) = O(n^3)$ (Teorema 10.2(b))
-

Contoh 10.15

- $O(5n^2) = O(n^2)$ (Teorema 10.2(c))
 - $n^2 = O(n^2)$ (Teorema 10.2d))
-

Aturan Menentukan Kompleksitas Waktu Asimptotik

Kompleksitas waktu asimptotik suatu algoritma dapat ditentukan dengan salah satu dari 2 cara di bawah ini:

1. Cara I

Jika kompleksitas waktu $T(n)$ dari algoritma sudah dihitung, maka kompleksitas waktu asimptotiknya dapat langsung ditentukan dengan mengambil suku yang mendominasi fungsi T dan menghilangkan koefisiennya. Cara ini bersuaian dengan Teorema 10.2.

Contoh:

- (i) pada algoritma CariElemenTerbesar, $T(n) = n - 1 = O(n)$
- (ii) pada algoritma PencarianBeruntun,

$$T_{\min}(n) = 1 = O(1)$$

$$T_{\max}(n) = n = O(n)$$

$$T_{\text{avg}}(n) = (n + 1)/2 = O(n), \text{ atau, dengan cara lain sebagai berikut:}$$

- (iii) pada algoritma PencarianBiner,

$$T_{\min}(n) = 1 = O(1)$$

$$T_{\max}(n) = {}^2\log n = O({}^2\log n)$$

- (iv) pada algoritma PengurutanSeleksi,

$$T(n) = \frac{n(n-1)}{2} = O(n^2)$$

(semua notasi O -Besar pada (i), (ii), (iii), dan (iv) di atas didasarkan pada jumlah perbandingan elemen-elemen larik)

(v) pada algoritma Kali,

$$T(n) = 2(n - 2) = O(n) \quad \text{dihitung dari jumlah operasi perkalian}$$

$$(vi) T(n) = (n + 2) \log(n^2 + 1) + 5n^2 = O(n^2)$$

Penjelasannya adalah sebagai berikut:

$$\begin{aligned} T(n) &= (n + 2) \log(n^2 + 1) + 5n^2 \\ &= f(n)g(n) + h(n), \text{ dengan } f(n) = (n + 2), g(n) = \log(n^2 + 1), \\ &\quad \text{dan } h(n) = 5n^2 \end{aligned}$$

Kita rinci satu per satu:

$$\Rightarrow f(n) = (n + 2) = O(n)$$

$$\Rightarrow g(n) = \log(n^2 + 1) = O(\log n), \text{ karena}$$

$$\log(n^2 + 1) \leq \log(2n^2)$$

$$= \log 2 + \log n^2$$

$$= \log 2 + 2 \log n \leq 3 \log n \text{ untuk } n > 2$$

$$\Rightarrow h(n) = 5n^2 = O(n^2)$$

maka

$$\begin{aligned} T(n) &= (n + 2) \log(n^2 + 1) + 5n^2 = O(n)O(\log n) + O(n^2) \\ &= O(n \log n) + O(n^2) && \text{(Teorema 10.2(b))} \\ &= O(\max(n \log n, n^2)) && \text{(Teorema 10.2(a))} \\ &= O(n^2) \end{aligned}$$

2. Cara II

Umumnya menghitung kompleksitas waktu untuk kasus terbaik dan kasus rata-rata sangat sulit dilakukan. Hal ini disebabkan informasi tentang probabilitas setiap permutasi data masukan yang berukuran n tidak diketahui atau tidak dapat ditentukan. Karena itu cukup beralasan kalau kita hanya meninjau kompleksitas waktu terburuk saja. Di bawah ini diberikan beberapa aturan untuk menghitung kompleksitas waktu asimptotik untuk kasus terburuk [AZM88]. Di dalam aturan tersebut kita tidak menghitung jumlah pelaksanaan operasi dasar sebagaimana pada perhitungan $T(n)$, tetapi kita menghitung langsung dengan menggunakan notasi O -Besar:

- (a) Pengisian nilai (*assignment*), perbandingan, operasi aritmetika (+, -, *, /, div, mod), *read*, *write*, pengaksesan elemen larik, memilih *field* tertentu dari sebuah *record*, dan pemanggilan fungsi/prosedur membutuhkan waktu $O(1)$.

Contoh: Tinjau potongan algoritma berikut

<u>read</u> (x)	$O(1)$
$x \leftarrow x + 1$	$O(1) + O(1) + O(1) = O(1)$
<u>write</u> (x)	$O(1)$

Kompleksitas waktu asimptotik algoritma = $O(1) + O(1) + O(1) = O(1)$

Penjelasan:

$$\begin{aligned}
 O(1) + O(1) + O(1) &= O(\max(1,1)) + O(1) && \text{(Teorema 10.2(a)(i))} \\
 &= O(1) + O(1) \\
 &= O(\max(1,1)) = O(1) && \text{(Teorema 10.2(a)(i))}
 \end{aligned}$$

- (b) if C then S1 else S2 membutuhkan waktu $T_C + \max(T_{S1}, T_{S2})$ yang dalam hal ini T_C , T_{S1} , dan T_{S2} adalah kompleksitas waktu C, S1, dan S2.

Contoh: Tinjau potongan algoritma berikut

<u>read</u> (x)	$O(1)$
<u>if</u> x mod 2 = 0 <u>then</u>	$O(1)$
$x \leftarrow x + 1$	$O(1)$
<u>write</u> (x)	$O(1)$
<u>else</u>	
<u>write</u> (x)	$O(1)$
<u>endif</u>	

Kompleksitas waktu asimptotik algoritma:

$$\begin{aligned}
 &= O(1) + O(1) + \max(O(1) + O(1), O(1)) \\
 &= O(1) + \max(O(1), O(1)) \\
 &= O(1) + O(1) \\
 &= O(1)
 \end{aligned}$$

- (c) Kalang for. Kompleksitas waktu kalang for adalah jumlah pengulangan dikali dengan kompleksitas waktu badan (*body*) kalang.

Contoh: Tinjau potongan algoritma berikut

<u>for</u> i \leftarrow 1 <u>to</u> n <u>do</u>	
jumlah \leftarrow jumlah + a_i	$O(1)$
<u>endfor</u>	

Kompleksitas waktu asimptotik = $n \cdot O(1) = O(n \cdot 1) = O(n)$

atau dengan pendekatan lain:

$$\begin{aligned} \text{Kompleksitas waktu asimptotik} &= n \cdot O(1) \\ &= O(n) O(1) && \text{Teorema 10.2(d)} \\ &= O(n) && \text{Teorema 10.2(c)} \end{aligned}$$

Contoh: Tinjau potongan algoritma berikut

```

for i ← 1 to n do
  for j ← 1 to n do
    aij ← 0      O(1)
  endfor
endfor

```

Kalang terdalam mempunyai kompleksitas waktu $O(n)$. Kalang terluar dikerjakan sebanyak n kali, sehingga kompleksitas waktu asimptotik seluruhnya adalah

$$n \cdot O(n) = O(n \cdot n) = O(n^2)$$

Contoh: Tinjau potongan algoritma berikut yang memiliki kalang bersarang dengan dua buah instruksi di dalamnya

```

for i ← 1 to n do
  for j ← 1 to i do
    a ← a + 1      O(1)
    b ← b - 2      O(1)
  endfor
endfor

```

$$\text{waktu untuk } a \leftarrow a+1 = O(1)$$

$$\text{waktu untuk } b \leftarrow b - 2 = O(1)$$

$$\text{total waktu untuk badan kalang} = O(1) + O(1) = O(1)$$

$$\text{waktu untuk kalang terdalam} = i \cdot O(1) = O(i) \cdot O(1) = O(i \cdot 1) = O(i)$$

$$\text{waktu untuk kalang terluar} = \sum_{i=1}^n O(i)$$

$$= O\left(\sum_{i=1}^n i\right) \quad (\text{Teorema 10.2(a)(ii)})$$

$$= O\left(\frac{n(n+1)}{2}\right)$$

$$= O\left(\frac{n^2}{2} + \frac{n}{2}\right)$$

$$= O\left(\frac{n^2}{2}\right) \quad (\text{Teorema 10.2(a)(i)})$$

$$= O(n^2) \quad (\text{Teorema 10.2(c)})$$

Jadi, kompleksitas waktu algoritma adalah $O(n^2)$.

- (d) while C do S; dan repeat S until C; Untuk kedua buah kalang, kompleksitas waktunya adalah jumlah pengulangan dikali dengan kompleksitas waktu badan C dan S. Masalah yang muncul adalah bila jumlah pengulangan tidak dapat ditentukan karena pengulangan dilakukan bergantung pada kondisi yang harus dipenuhi .

Contoh: Tinjau potongan algoritma berikut

```

i ← 2                                O(1)
while i ≤ n do                        O(1)
    jumlah ← jumlah + ai            O(1)
    i ← i + 1                        O(1)
endwhile

```

Kalang while dieksekusi sebanyak $n - 1$ kali, sehingga kompleksitas waktu asimptotik algoritma adalah

$$\begin{aligned}
 &= O(1) + (n - 1) \{ O(1) + O(1) + O(1) \} \\
 &= O(1) + (n - 1) O(1) \\
 &= O(1) + O(n - 1) \\
 &= O(1) + O(n) \\
 &= O(n)
 \end{aligned}$$

Jadi, kompleksitas waktu algoritma adalah $O(n)$.

Contoh: Tinjau potongan algoritma berikut yang mempunyai kalang yang tidak dapat ditentukan panjangnya:

```

ketemu ← false
while (p ≠ Nil) and (not ketemu) do
    if p↑.kunci = x then
        ketemu ← true
    else
        p ← p↑.Next
    endif
endwhile
{ p = Nil or ketemu }

```

Di sini, pengulangan akan berhenti bila x yang dicari ditemukan di dalam senarai atau seluruh elemen senarai sudah dibandingkan. Jika jumlah elemen

senarai adalah n , maka kompleksitas waktu terburuknya adalah $O(n)$ -yaitu kasus x tidak ditemukan. Pada kebanyakan kasus, tiap elemen senarai mempunyai peluang yang sama mengandung nilai x . Tetapi, ingatlah kembali bahwa kompleksitas waktu asimptotik menyatakan waktu terpanjang yang dibutuhkan untuk eksekusi algoritma sebagai fungsi dari n . Jadi, kita andaikan x terdapat pada elemen terakhir atau x tidak ditemukan. Dengan kata lain, kompleksitas waktu kalang tersebut adalah $O(n)$.

(e) case A

A = a_1 : S_1

A = a_2 : S_2

...

A = a_n : S_n

endcase

membutuhkan waktu $\max(T_{S1}, T_{S2}, \dots, T_{Sn})$ yang dalam hal ini $T_{S1}, T_{S2}, \dots, T_{Sn}$ dan T_{S2} adalah kompleksitas waktu $S1, S2, \dots, Sn$

Contoh: Tinjau potongan algoritma berikut

```

read(n)                                O(1)
case n
  n mod 2 = 0 : for i ← 1 to n do
                  write(i*i)           O(1)
                endfor
  x mod 2 = 1 : write('ganjil')         O(1)
endcase

```

Kompleksitas waktu asimptotik algoritma:

$$\begin{aligned}
 &= O(1) + \max(O(n), O(1)) \\
 &= O(1) + O(n) \\
 &= O(n)
 \end{aligned}$$

(f) Untuk fungsi/prosedur rekursif, digunakan teknik perhitungan kompleksitas dengan *relasi rekurens*.

Contoh: Tinjau potongan algoritma rekursif berikut

```

function faktorial(input n : integer) → integer
{ mengembalikan nilai n!, n tidak negatif }

Algoritma
  if n = 0 then
    return 1
  else
    return n * faktorial(n-1)
  endif

```

Kompleksitas waktu untuk algoritma faktorial rekursif di atas dihitung berdasarkan jumlah operasi perkalian dalam relasi rekurens (*recurrence relation*) berikut:

- (a) Untuk kasus basis, tidak ada operasi perkalian (0),
- (b) Untuk kasus rekurens, kompleksitas waktu diukur dari jumlah perkalian (1) ditambah kompleksitas waktu untuk $\text{faktorial}(n - 1)$.

Jadi,

$$T(n) = \begin{cases} 0, & n = 0 \\ 1 + T(n - 1), & n > 0 \end{cases}$$

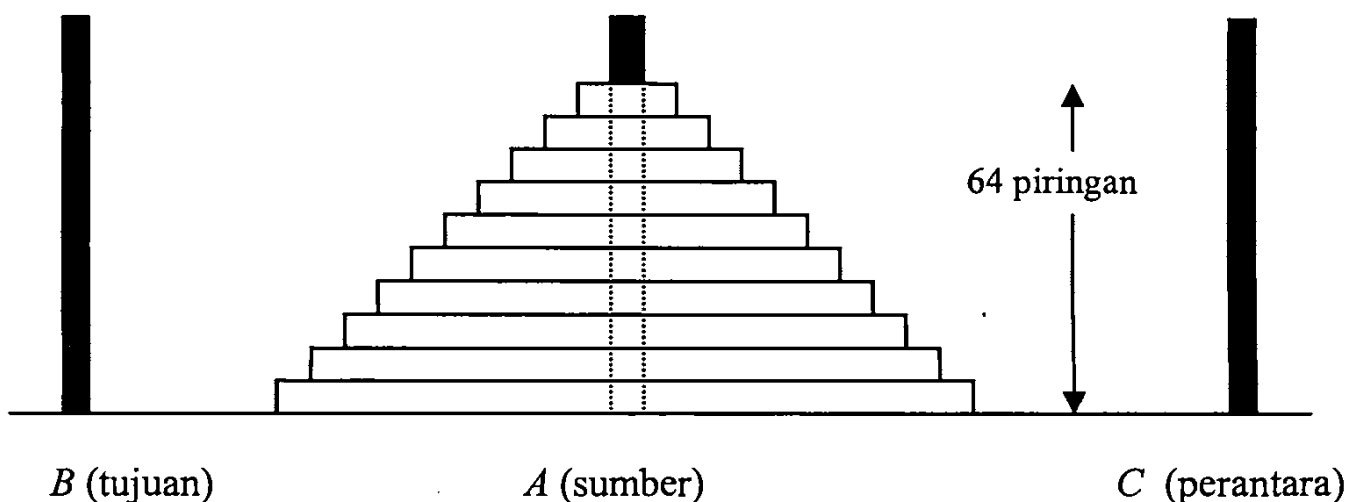
dengan a dan b adalah konstanta. $T(n)$ dapat dipecahkan sebagai berikut:

$$\begin{aligned} T(n) &= 1 + T(n - 1) \\ &= 1 + 1 + T(n - 2) = 2 + T(n - 2) \\ &= 2 + 1 + T(n - 3) = 3 + T(n - 3) \\ &\vdots \\ &= n + T(0) \\ &= n + 0 \\ &= O(n) \end{aligned}$$

Jadi, kompleksitas algoritma faktorial adalah $O(n)$.

Contoh 10.16

(Menara Hanoi). Contoh ini merupakan legenda klasik pendeta Budha. Di kota Hanoi, Vietnam, terdapat tiga buah tiang tegak setinggi 5 meter dan 64 buah piringan (*disk*) dari berbagai ukuran (Gambar 10.3). Tiap piringan mempunyai lubang di tengahnya yang memungkinkannya untuk dimasukkan ke dalam tiang.



Gambar 10.3 Menara Hanoi

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!

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 (disk) dari A ke B , n adalah jumlah piringan. Pada mulanya seluruh piringan berada di tiang A , piringan terbesar berada paling bawah. Setelah proses pemindahan, seluruh piringan pindah ke tiang B , piringan paling besar berada paling bawah.)

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 10.7 Algoritma Menara Hanoi

Kompleksitas waktu dari algoritma Menara Hanoi diukur dari jumlah perpindahan piringan dari satu tiang ke tiang lain. Untuk $n = 1$ piringan (basis), hanya ada satu piringan yang dipindahkan, sedangkan untuk $n > 1$ piringan (rekurens), satu piringan dipindahkan ditambah dengan jumlah piringan yang dipindahkan pada pemanggilan prosedur Hanoi untuk $n - 1$ piringan lain. Kompleksitas waktu algoritma adalah

$$T(n) = \begin{cases} 1 & , n = 1 \\ 1 + 2T(n - 1) & , n > 1 \end{cases}$$

Bagian rekurens dipecahkan sebagai berikut:

$$\begin{aligned} T(n) &= 1 + 2T(n - 1) \\ &= 1 + 2(1 + 2T(n - 2)) = 1 + 2 + 2^2T(n - 2) \\ &= 1 + 2 + 2^2(1 + 2T(n - 3)) = 1 + 2 + 2^2 + 2^3T(n - 3) \\ &\vdots \\ &= (1 + 2 + 2^2 + \dots + 2^{n-2}) + 2^{n-1}T(1) \\ &= 1 + 2 + 2^2 + \dots + 2^{n-2} + 2^{n-1} \cdot 1 = 1 + 2 + 2^2 + \dots + 2^{n-2} + 2^{n-1} \\ &= 2^n - 1 = O(2^n) \end{aligned}$$

Jadi, persoalan menara Hanoi mempunyai kompleksitas ekponensial, $O(2^n)$. Perhatikanlah bahwa $T(n) = 2^n - 1$ adalah jumlah seluruh perpindahan piringan dari satu tiang ke tiang lainnya. Jika untuk memindahkan satu piringan dibutuhkan waktu satu detik, maka waktu yang dibutuhkan untuk memindahkan seluruh piringan adalah $2^{64} - 1$ detik = 18.446.744.073.709.551.615 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 dari tiang *A* ke tiang *B* ada benarnya, karena 600 milyar tahun adalah waktu yang sangat lama, dunia saat ini mungkin sudah tidak mungkin bertahan lagi dalam jangka waktu itu, dan akhirnya hancur (kiamat)! ■

Contoh 10.17

Misalkan kita memiliki dua buah larik *a* dan *b*, masing-masing dengan elemen a_1, a_2, \dots, a_n , dan b_1, b_2, \dots, b_n , yang keduanya berukuran masing-masing *n* elemen dan setiap larik sudah terurut menaik. Kita ingin membentuk sebuah larik baru, *c*, yang merupakan gabungan dari dua buah larik tersebut. Penggabungan dilakukan dengan cara membandingkan satu elemen pada larik *a* dengan satu elemen pada larik *b*. Jika elemen pada *a* lebih kecil dari elemen pada *b*, maka salin elemen dari *a* ke *c*. Elemen berikutnya pada *a* maju satu elemen, sedangkan elemen *b* tetap. Hal yang sama juga berlaku bila elemen dari *b* lebih kecil dari elemen *a*, maka salin elemen dari *b* ke *c*. Larik *b* maju satu elemen, larik pertama tetap. Dengan cara seperti ini, akan ada larik yang elemennya sudah duluan habis disalin, sedangkan larik yang lain masih tersisa. Elemen larik yang tersisa disalin ke larik *c*.

Contoh ilustrasi penggabungan:

<i>a</i>	<i>b</i>		<i>c</i>
1 13 24	2 15 27	1 < 2 → 1	1
1 13 24	2 15 27	2 < 13 → 2	1 2
1 13 24	2 15 27	13 < 15 → 13	1 2 13
1 13 24	2 15 27	15 < 24 → 15	1 2 13 15
1 13 24	2 15 27	24 < 27 → 24	1 2 13 15 24
1 13 24	2 15 27	27 →	1 2 13 15 24 27

Algoritmanya adalah sebagai berikut:

```

procedure GabungLarikTerurut(input a1, a2, ..., an,
                             b1, b2, ..., bn : integer,
                             output c1, c2, ..., c2n : integer)
{Penggabungan dua buah larik terurut, a dan b, menghasilkan larik baru,
 c, yang terurut menaik.
Masukan: a1, a2, ..., an dan b1, b2, ..., bn
Keluaran: c1, c2, ..., c2n
}

Deklarasi
    k1, k2, k3 : integer

Algoritma
    k1 ← 1
    k2 ← 1
    k3 ← 1
    while (k1 ≤ n) and (k2 ≤ n) do
        if ak1 ≤ bk2 then
            ck3 ← ak1
            k1 ← k1 + 1
        else
            ck3 ← bk2
            k2 ← k2 + 1
        endif
        k3 ← k3 + 1
    endwhile
    { k1 > n or k2 > n }

    { salin sisa larik a, jika ada }
    while (k1 ≤ n) do
        ck3 ← ak1
        k1 ← k1 + 1
    
```



```

    k3 ← k3 + 1
  endwhile
  { k1 > n }

  { salin sisa larik b, jika ada }
  while (k2 ≤ n) do
    ck3 ← bk2
    k2 ← k2 + 1
    k3 ← k3 + 1;
  endwhile
  { k2 > n }

```

Algoritma 10.7 Penggabungan dua buah larik terurut

Berapa kompleksitas asimptotik algoritma GabungLarikTerurut di atas?

Penyelesaian:

Algoritma GabungLarikTerurut mempunyai tiga buah kalang *while-do*. Empat buah instruksi pengisian nilai sebelum kalang *while-do* yang pertama mempunyai kompleksitas $O(1)$.

Kita akan menghitung kompleksitas kalang *while-do* yang pertama. Pada kasus terburuk, yaitu pada kasus seluruh elemen larik *a* lebih kecil dari seluruh elemen larik *b* (atau sebaliknya), kalang *while-do* ini akan dijalankan sebanyak n kali. Instruksi *if-then-else* di dalam badan kalang *while-do* mempunyai kompleksitas $O(1)$. Dengan demikian, kompleksitas kalang *while-do* pertama adalah

$$n \cdot O(1) = O(n)$$

Salah satu dari kalang *while-do* kedua atau kalang *while-do* ketiga akan dieksekusi, yaitu untuk menyalin elemen larik yang tersisa. Kita akan menghitung kompleksitas waktu kalang kedua/ketiga. Pada kasus terburuk tersebut di atas, semua elemen larik *a* atau semua elemen larik *b* disalin ke larik *c*. Ini berarti kalang *while-do* kedua/ketiga akan dijalankan sebanyak n kali. Instruksi penyalinan di dalam badan kalang itu membutuhkan waktu $O(1)$. Dengan demikian, kompleksitas kalang *while-do* kedua/ketiga adalah

$$n \cdot O(1) = O(n)$$

Kompleksitas waktu asimptotik seluruhnya adalah

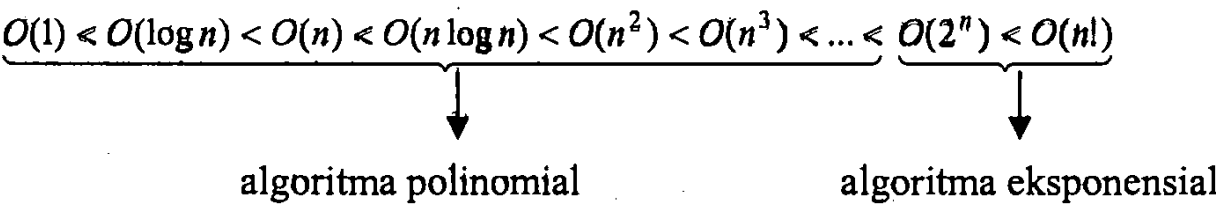
$$O(1) + O(n) + O(n) = O(1) + O(\max(n, n)) = O(\max(1, n)) = O(n)$$

Jadi, kompleksitas asimptotik algoritma GabungLarikTerurut adalah $O(n)$. ■

Pengelompokan Algoritma Berdasarkan Notasi O-Besar

Tiap-tiap algoritma mempunyai kompleksitas waktu asimptotik masing-masing. Kompleksitas waktu asimptotik ini dapat digunakan untuk mengelompokkan algoritma (Tabel 10.2).

Urutan spektrum kompleksitas waktu algoritma adalah :



Tabel 10.2 Kelompok algoritma berdasar kan kompleksitas waktu asimptotiknya

Kelompok Algoritma	Nama
$O(1)$	konstan
$O(\log n)$	logaritmik
$O(n)$	linier
$O(n \log n)$	$n \log n$
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
$O(n!)$	faktorial

Urutan spektrum kompleksitas waktu di atas ditafsirkan sebagai berikut: jika $O(f(n))$ terletak sebelum $O(g(n))$, maka itu berarti $f(n) \leq g(n)$ untuk semua bilangan bulat n . Jadi, jika algoritma A dan B mempunyai waktu pelaksanaan masing-masing $O(f(n))$ dan $O(g(n))$ dan $O(f(n))$ terletak sebelum $O(g(n))$, maka algoritma A dikatakan lebih *mangkus* daripada algoritma B untuk ukuran masukan yang besar.

Penjelasan masing-masing kelompok algoritma adalah sebagai berikut [SED92]:

$O(1)$ Kompleksitas $O(1)$ berarti waktu pelaksanaan algoritma adalah tetap, tidak bergantung pada ukuran masukan. Dengan kata lain, jumlah operasi abstrak adalah tetap dan total kebutuhan waktunya dibatasi oleh sebuah konstanta. Bila n dijadikan dua kali semula, misalnya, waktu pelaksanaan algoritmanya hanya bertambah sejumlah konstanta. Algoritma yang termasuk kelompok ini adalah algoritma yang kebanyakan instruksinya dilaksanakan satu kali atau paling banyak beberapa kali. Contohnya prosedur tukar di bawah ini:

```

procedure Tukar(input/output a, b :integer)
{ Mempertukarkan nilai a dan b. Setelah pertukaran, a berisi nilai b
  dan b berisi nilai a semula.
  Masukan: a dan b
  Keluaran: a dan b
}

```

Deklarasi

```
temp : integer
```

Algoritma

```
temp ← a
```

```
a ← b
```

```
b ← temp
```

Algoritma 10.8 Pertukaran dua buah nilai

Di sini jumlah operasi penugasan (*assignment*) ada tiga buah dan tiap operasi dilakukan satu kali. Jadi, $T(n) = 3 = O(1)$.

$O(\log n)$ Kompleksitas waktu logaritmik berarti laju pertumbuhan waktunya berjalan lebih lambat daripada pertumbuhan n . Algoritma yang termasuk kelompok ini adalah algoritma yang memecahkan persoalan besar dengan mentransformasikannya menjadi beberapa persoalan yang lebih kecil yang berukuran sama (misalnya algoritma pencarian_biner). Di sini basis algoritma tidak terlalu penting sebab bila n dinaikkan dua kali semula, misalnya, $\log n$ meningkat sebesar sejumlah konstanta. Fungsi $\log n$ hanya meningkat menjadi dua kali semula jika n dinaikkan sebesar n^2 kali semula (basis dua).

$O(n)$ Algoritma yang waktu pelaksanaannya linier umumnya terdapat pada kasus yang setiap elemen masukannya dikenai proses yang sama, misalnya algoritma pencarian_beruntun. Bila n dijadikan dua kali semula, maka waktu pelaksanaan algoritma juga dua kali semula.

$O(n \log n)$ Waktu pelaksanaan yang $n \log n$ terdapat pada algoritma yang memecahkan persoalan menjadi beberapa persoalan yang lebih kecil, menyelesaikan tiap persoalan secara independen, dan menggabung solusi masing-masing persoalan. Algoritma yang diselesaikan dengan teknik bagi dan gabung mempunyai kompleksitas asimptotik jenis ini. Bila $n = 1000$, maka $n \log n$ mungkin 20.000. Bila n dijadikan dua kali semula, maka $n \log n$ menjadi dua kali semula (tetapi tidak terlalu banyak)

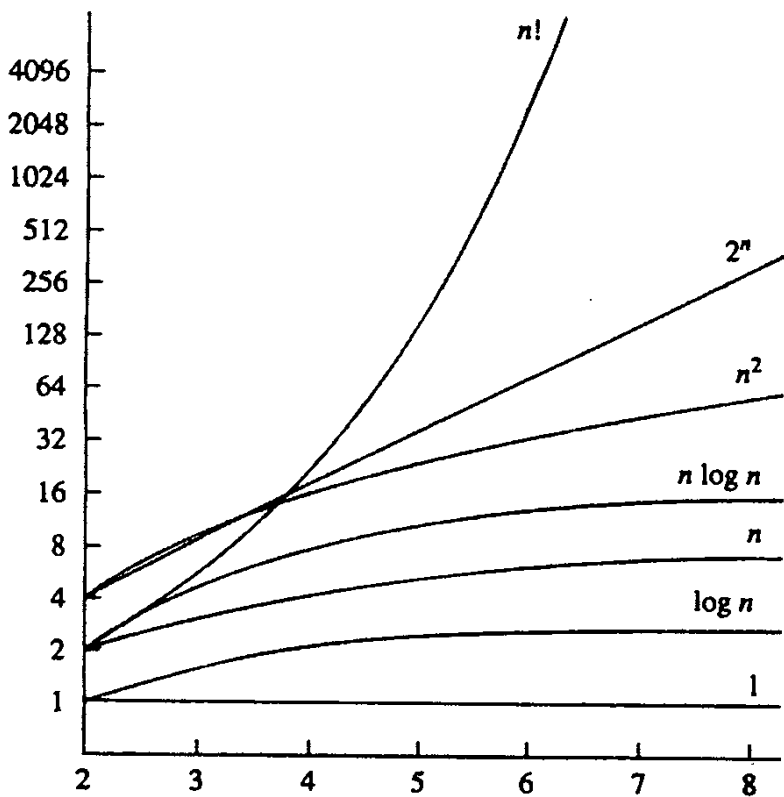
- $O(n^2)$ Algoritma yang waktu pelaksanaannya kuadratik hanya praktis digunakan untuk persoalan yang berukuran kecil. Umumnya algoritma yang termasuk kelompok ini memproses setiap masukan dalam dua buah kalang bersarang, misalnya pada algoritma `urut_maks`. Bila $n = 1000$, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, maka waktu pelaksanaan algoritma meningkat menjadi empat kali semula.
- $O(n^3)$ Seperti halnya algoritma kuadratik, algoritma kubik memproses setiap masukan dalam tiga buah kalang bersarang, misalnya algoritma perkalian matriks. Bila $n = 100$, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, waktu pelaksanaan algoritma meningkat menjadi delapan kali semula.
- $O(2^n)$ Algoritma yang tergolong kelompok ini mencari solusi persoalan secara "*brute force*", misalnya pada algoritma mencari sirkuit Hamilton (lihat Bab Graf). Bila $n = 20$, waktu pelaksanaan algoritma adalah 1.000.000. Bila n dijadikan dua kali semula, waktu pelaksanaan menjadi kuadrat kali semula!
- $O(n!)$ Seperti halnya pada algoritma eksponensial, algoritma jenis ini memproses setiap masukan dan menghubungkannya dengan $n - 1$ masukan lainnya, misalnya algoritma Persoalan Pedagang Keliling (*Travelling Salesperson Problem* - lihat bab 9). Bila $n = 5$, maka waktu pelaksanaan algoritma adalah 120. Bila n dijadikan dua kali semula, maka waktu pelaksanaan algoritma menjadi faktorial dari $2n$.

Enam buah notasi orde yang pertama, $O(1)$ sampai $O(n^3)$, adalah orde algoritma yang penting. Di sini kebutuhan waktunya dibatasi oleh polinomial, dan algoritmanya dinamakan **algoritma polinomial**. Algoritma yang kompleksitas waktu asimptotiknya $O(2^n)$ dinamakan **algoritma eksponensial**, karena bila n diperbesar, kebutuhan waktunya meningkat dengan tajam. Algoritma yang kebutuhan waktunya $O(2^n)$ tidak dapat digolongkan sebagai algoritma polinomial karena tidak ada bilangan bulat m sedemikian sehingga polinom n^m membatasi 2^n , atau $2^n \neq O(n^m)$ untuk sembarang bilangan bulat m . Algoritma yang kebutuhan waktunya $O(n!)$ juga digolongkan sebagai algoritma eksponensial karena pertumbuhan waktunya mempunyai kemiripan dengan fungsi 2^n .

Tabel 10.3 dan Gambar 10.4 di bawah ini memperlihatkan bagaimana kebutuhan waktu untuk ketujuh fungsi kompleksitas waktu tumbuh [HOR90] (diandaikan konstanta atau koefisien di depan persamaannya sama dengan satu).

Tabel 10.3 Nilai masing-masing fungsi untuk setiap bermacam-macam nilai n

$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
0	1	0	1	1	2	1
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	9	24	64	512	256	362880
4	16	64	256	4096	65536	20922789888000
5	32	160	1024	32768	4294967296	(terlalu besar untuk ditulis)



Gambar 10.4 Laju pertumbuhan fungsi kompleksitas waktu [ROS99].

Tabel 10.4 memperlihatkan waktu eksekusi algoritma yang kompleksitas waktunya adalah fungsi-fungsi acuan, yaitu $f(n) = n$, $f(n) = \log n$, $f(n) = n \log n$, $f(n) = n^2$, $f(n) = n^3$, dan $f(n) = 2^n$ [NEA96]. Asumsi yang digunakan adalah satu kali operasi dasar pada setiap algoritma membutuhkan waktu 1 nanodetik (10^{-9} detik). Tabel tersebut memperlihatkan hasil yang mengejutkan. Orang mungkin berharap bahwa selama algoritmanya bukan algoritma eksponensial, ia merupakan algoritma yang memadai. Tetapi, algoritma kuadratik membutuhkan waktu 31,7 tahun untuk memproses masukan sebanyak 1 miliar, sedangkan algoritma $O(n \log n)$ membutuhkan waktu hanya 29,9 detik untuk memproses masukan sebanyak itu. Sebuah algoritma sebaiknya $O(n \log n)$ atau kita mengsumsikan bahwa algoritma dapat memproses masukan yang berukuran sangat besar dalam waktu yang dapat ditoleransi. Hal ini tidaklah berarti bahwa

algoritma yang kompleksitas waktunya lebih tinggi tidak berguna. Algoritma dengan kebutuhan waktu kuadratik, kubik, dan yang lebih tinggi sering berguna memproses masukan pada banyak aplikasi [NEA96].

Tabel 10.4 Waktu eksekusi algoritma dengan berbagai macam kompleksitas waktu [NEA96]

n	$f(n) = \log n$	$f(n) = n$	$f(n) = n \log n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0,003 μ s	0,01 μ s	0,033 μ s	0,1 μ s	1 μ s	1 μ s
20	0,004 μ s	0,02 μ s	0,086 μ s	0,4 μ s	8 μ s	1 ms
30	0,005 μ s	0,03 μ s	0,147 μ s	0,9 μ s	27 μ s	1 s
40	0,005 μ s	0,04 μ s	0,213 μ s	1,6 μ s	64 μ s	18,3 menit
50	0,006 μ s	0,05 μ s	0,282 μ s	2,5 μ s	125 μ s	13 hari
10^2	0,007 μ s	0,10 μ s	0,664 μ s	10 μ s	1 ms	$4 \cdot 10^{13}$ thn
10^3	0,010 μ s	1,00 μ s	9,966 μ s	1 ms	1 s	
10^4	0,013 μ s	10 μ s	130 μ s	100 ms	16,7 menit	
10^5	0,017 μ s	0,10 ms	1,67 ms	10 s	11,6 hari	
10^6	0,020 μ s	1 ms	19,93 ms	16,7 menit	31,7 tahun	
10^7	0,023 μ s	0,01 s	0,23 s	1,16 hari	31,709 th	
10^8	0,027 μ s	0,10 s	2,66 s	115,7 hari	$1,17 \cdot 10^7$ th	
10^9	0,030 μ s	1 s	29,90 s	31,7 tahun		

Keterangan:

1 μ s = 10^{-6} second

1 ms = 10^{-3} second

Sekarang, tinjau secara khusus algoritma yang kebutuhan waktunya eksponensial. Algoritma eksponensial tumbuh sangat cepat dengan bertambahnya nilai n . Algoritma eksponensial hanya bagus untuk nilai n yang sangat kecil, bahkan bila kita menurunkan nilai koefisien di depannya, pengurangan kebutuhan waktunya tidak mengalami banyak kemajuan. Agar lebih jelas mengapa perubahan koefisien atau konstanta, daripada perubahan orde, menghasilkan sedikit perbaikan dalam kebutuhan waktu pelaksanaan program, kita dapat melihatnya pada contoh di berikut ini.

Misalkan orde dua buah algoritma adalah $O(n^2 \cdot 2^n)$ dan $O(n \cdot 2^n)$. Kedua algoritma adalah eksponensial, tetapi fungsi kompleksitas algoritma pertama memiliki faktor tambahan n dibandingkan dengan yang kedua. Koefisien di depan persamaannya diandaikan sama dengan satu. Untuk bermacam-macam n , nilai setiap fungsi ditunjukkan pada Tabel 10.5. Dengan menggunakan andaian bahwa satu buah operasi memakan waktu 0.001 detik, kita menemukan bahwa untuk $n = 30$ kebutuhan waktu untuk algoritma pertama 8.9 jam dan algoritma kedua 11 hari. Meskipun faktor tambahan n membuat perbedaan yang berarti, ciri eksponensial mendominasi perhitungan dan menyiratkan bahwa kedua algoritma membutuhkan waktu yang lama. Jika kita dapat mempercepat algoritma kedua dengan faktor $1/10$, sehingga menjadi $(1/10) n^2 \cdot 2^n$, maka untuk ukuran masukan $n > 10$

algoritma pertama masih lebih cepat. Selanjutnya, untuk $n = 30$ waktu yang dibutuhkan algoritma pertama masih lebih besar dari 24 jam. Kesimpulan yang kita buat dari contoh ini adalah: algoritma eksponensial membutuhkan waktu yang besar. Pengubahan koefisien maupun pemakaian komputer yang lebih cepat tidak memberi perbaikan waktu komputasi yang berarti. Jalan alternatifnya ialah menurunkan algoritma dengan orde yang lebih baik [HOR77].

Tabel 10.5 Nilai fungsi $n^2 \cdot 2^n$ dan $n \cdot 2^n$ untuk setiap bermacam-macam nilai n

n	$n^2 \cdot 2^n$	$n \cdot 2^n$
5	160	800
10	10240	102400
15	491520	7372800
20	20971520	419430400
30	3.2×10^{10}	9.6×10^{11}

Beberapa catatan lain yang ditambahkan di sini adalah:

1. Perlu diingat sekali lagi, bahwa kompleksitas waktu asimptotik $-O(f(n))$ -hanyalah ukuran *kasar* kebutuhan waktu pelaksanaan sebuah algoritma untuk n yang besar, jadi bukan ukuran waktu sebenarnya. Notasi O -Besar mengekspresikan berapa waktu yang dibutuhkan untuk menyelesaikan masalah dengan meningkatnya ukuran masukan. Jika kita ingin mengukur kebutuhan waktu yang lebih presisi, kita harus juga membandingkan koefisien dan konstanta dalam persamaan $T(n)$ -nya. Sebagai misal, meskipun $T(n) = 1000n \log n$ dan $T(n) = (n \log n)/10$ sama-sama berorde $O(n \log n)$, tetapi $T(n) = (n \log n)/10$ lebih cepat untuk n yang besar. Contoh lainnya, andaikan algoritma A membutuhkan waktu $T(n) = 300n$ dan algoritma B membutuhkan waktu $T(n) = 5n^2$. Untuk ukuran masukan berukuran $n = 5$, algoritma A membutuhkan 1500 satuan waktu dan algoritma B membutuhkan 125 satuan waktu. Jadi, untuk masukan yang berukuran kecil algoritma B lebih cepat daripada algoritma A . Tentu saja algoritma A lebih cepat daripada algoritma B untuk n yang besar.
2. Kompleksitas waktu asimptotik dapat digunakan untuk membandingkan dua buah algoritma. Misalkan ada sebuah persoalan yang sama diselesaikan dengan dua buah algoritma yang berbeda. Algoritma I mempunyai kompleksitas waktu $O(n)$, dan algoritma II mempunyai kompleksitas $O(n^2)$. Manakah yang lebih cepat, algoritma I atau algoritma II untuk n yang besar? Mudah dilihat bahwa untuk n yang cukup besar, waktu untuk algoritma II tumbuh lebih cepat daripada waktu algoritma I. Sedangkan untuk menghitung kebutuhan waktu yang sebenarnya, kita juga harus melihat konstanta yang mendahuluinya.

Contoh (a): (i) Algoritma I $\rightarrow T(n) = 2n = O(n)$
 Algoritma II $\rightarrow T(n) = n^2 = O(n^2)$

Terlihat, untuk $n > 2$ algoritma I yang berorde $O(n)$ paling cepat waktu pelaksanaannya dibandingkan dengan algoritma II yang berorde $O(n^2)$. Jadi, algoritma I cepat untuk $n > 2$.

Contoh (b): (ii) Aalgoritma I $\rightarrow T(n) = 10^4 n = O(n)$
 Aalgoritma II $\rightarrow T(n) = n^2 = O(n^2)$

Terlihat, untuk $n < 10^4$ algoritma II paling cepat. Tetapi untuk $n > 10^4$ algoritma I yang paling cepat.

3. Sebuah masalah yang mempunyai algoritma dengan kompleksitas polinomial pada kasus-terburuk dianggap mempunyai algoritma yang “bagus”; artinya masalah tersebut mempunyai algoritma yang mangkus, dengan catatan polinomial tersebut berderajat rendah. Jika polinomnya berderajat tinggi, waktu yang dibutuhkan untuk mengeksekusi algoritma tersebut panjang. Untunglah pada kebanyakan kasus, fungsi polinomnya mempunyai derajat yang rendah.
4. Suatu masalah dikatakan *tractable* (mudah dari segi komputasi) jika ia dapat diselesaikan dengan algoritma yang memiliki kompleksitas polinomial kasus terburuk (artinya dengan algoritma yang mangkus), karena algoritma akan menghasilkan solusi dalam waktu yang lebih pendek [ROS99]. Sebaliknya, sebuah masalah dikatakan *intractable* (sukar dari segi komputasi) jika tidak ada algoritma yang mangkus untuk menyelesaikannya. Untuk menunjukkan bahwa suatu masalah *tractable* kita cukup menunjukkan ada algoritma yang mangkus untuk menyelesaikan masalah itu. Sebagai contoh, masalah menentukan nilai maksimum, masalah pencarian, masalah pengurutan, adalah masalah yang *tractable* karena algoritmanya mempunyai kompleksitas waktu polinom untuk kasus terburuk. Untuk menunjukkan bahwa suatu algoritma *intractable*, kita harus menunjukkan bahwa tidak ada algoritma yang mangkus untuk menyelesaikannya. Masalah TSP (*Travelling Salesperson Problem*) – lihat Bab Graf- adalah contoh masalah yang belum dapat diselesaikan dengan algoritma polinomial, karena itu *intractable*. Namun orang belum mampu membuktikan bahwa algoritma yang mangkus untuk menyelesaikan masalah itu tidak mungkin ada. Algoritma yang ada untuk menyelesaikan masalah *intractable*, seperti TSP itu, mempunyai kompleksitas eksponensial.
5. Masalah yang sama sekali tidak memiliki algoritma untuk memecahkannya disebut **masalah tak-terselesaikan** (*unsolved problem*). Sebagai contoh, masalah penghentian (*halting problem*) adalah masalah tak-terselesaikan. Masalah penghentian berbunyi: jika diberikan program dan sejumlah masukan, apakah program tersebut berhenti pada akhirnya [JOH90]?

6. Kebanyakan masalah yang dapat dipecahkan dipercaya tidak memiliki algoritma penyelesaian dalam kompleksitas waktu polinomial untuk kasus terburuk, karena itu dianggap *intractable*. Tetapi, jika solusi masalah tersebut ditemukan, maka solusinya dapat diperiksa dalam waktu polinomial. Masalah yang solusinya dapat diperiksa dalam waktu polinomial dikatakan termasuk ke dalam **kelas NP** (*non-deterministic polynomial*). Masalah yang *tractable* termasuk ke dalam **kelas P** (*polynomial*). Jenis kelas masalah lain adalah kelas **NP-lengkap** (*NP-complete*). Kelas masalah NP-lengkap memiliki sifat bahwa jika ada sembarang masalah di dalam kelas ini dapat dipecahkan dalam waktu polinomial, berarti semua masalah di dalam kelas tersebut dapat dipecahkan dalam waktu polinomial. Atau, jika kita dapat membuktikan bahwa salah satu dari masalah di dalam kelas itu *intractable*, berarti kita telah membuktikan bahwa semua masalah di dalam kelas tersebut *intractable*. Meskipun banyak penelitian telah dilakukan, tidak ada algoritma dalam waktu polinomial yang dapat memecahkan masalah di dalam kelas NP-lengkap. Secara umum diterima, meskipun tidak terbukti, bahwa tidak ada masalah di dalam kelas NP-lengkap yang dapat dipecahkan dalam waktu polinomial [ROS99].
7. Meskipun algoritma eksponensial secara asimptotik lebih buruk daripada algoritma polinomial, tetapi untuk n yang kecil algoritma eksponensial menunjukkan kinerja yang lebih baik. Contohnya, $2^n \leq 100n^2$ untuk $n = 1, 2, \dots, 14$. Jadi, untuk n yang kecil lebih tepat menggunakan algoritma eksponensial.
9. Penelitian untuk menemukan algoritma dengan kompleksitas waktu yang kecil masih terus berlangsung sampai saat ini, dan ini merupakan tantangan bagi para ilmuwan komputer, terutama para analis algoritma. Penemuan yang cukup bersejarah adalah algoritma **Transformasi Fourier Cepat** (*Fast Fourier Transform*) -atau TFC- yang ditulis oleh J.W Cooley dan J.W Tukey pada tahun 1965. Sebelumnya, algoritma Transformasi Fourier Farik (*discrete*) -atau *TFD*- yang biasa berorde $O(n^2)$, tetapi TFC memperkecilnya menjadi $O(n \log n)$. Dengan algoritma TFF, waktu komputasi transformasi Fourier berkurang cukup tajam. Transformasi Fourier banyak digunakan pada bidang pengolahan sinyal.

Notasi Omega-Besar dan Tetha-Besar

Notasi *O*-Besar menyediakan batas lebih atas (*upper bound*) untuk fungsi $T(n)$, tetapi tidak memberikan batas lebih bawah (*lower bound*). Untuk itu, kita mendefinisikan batas lebih bawah (*lower bound*) untuk kompleksitas waktu, yang dilambangkan dengan *Omega-Besar* (*Big-Ω*).

Definisi Ω -Besar adalah:

DEFINISI 10.2. $T(n) = \Omega(g(n))$ (dibaca " $T(n)$ adalah omega ($f(n)$ ") yang artinya $T(n)$ berorde paling kecil $g(n)$) bila terdapat konstanta C dan n_0 sedemikian sehingga

$$T(n) \geq C \cdot f(n)$$

untuk $n \geq n_0$.

Kita juga mendefinisikan Θ -Besar,

DEFINISI 10.3. $T(n) = \Theta(h(n))$ (dibaca " $T(n)$ adalah tetha $h(n)$ ") yang artinya $T(n)$ berorde sama dengan $h(n)$ jika $T(n) = O(h(n))$ dan $T(n) = \Omega(h(n))$.

Contoh 10.18

Tentukan notasi Ω dan Θ untuk $T(n) = 2n^2 + 6n + 1$.

Penyelesaian:

Karena $2n^2 + 6n + 1 \geq 2n^2$ untuk $n \geq 1$, maka dengan mengambil $C = 2$ kita memperoleh

$$2n^2 + 6n + 1 = \Omega(n^2)$$

Karena $2n^2 + 5n + 1 = O(n^2)$ dan $2n^2 + 6n + 1 = \Omega(n^2)$, maka $2n^2 + 6n + 1 = \Theta(n^2)$. ■

Contoh 10.19

Tentukan notasi notasi O , Ω dan Θ untuk $T(n) = 5n^3 + 6n^2 \log n$.

Penyelesaian:

Karena $0 \leq 6n^2 \log n \leq 6n^3$, maka $5n^3 + 6n^2 \log n \leq 11n^3$ untuk $n \geq 1$. Dengan mengambil $C = 11$, maka

$$5n^3 + 6n^2 \log n = O(n^3)$$

Karena $5n^3 + 6n^2 \log n \geq 5n^3$ untuk $n \geq 1$, maka dengan mengambil $C = 5$ kita memperoleh

$$5n^3 + 6n^2 \log n = \Omega(n^3)$$

Karena $5n^3 + 6n^2 \log n = O(n^3)$ dan $5n^3 + 6n^2 \log n = \Omega(n^3)$, maka $5n^3 + 6n^2 \log n = \Theta(n^3)$. ■

Contoh 10.19

Tentukan notasi notasi O , Ω dan Θ untuk $T(n) = 1 + 2 + \dots + n$.

Penyelesaian:

$$1 + 2 + \dots + n = O(n^2) \text{ karena } 1 + 2 + \dots + n \leq n + n + \dots + n = n^2 \text{ untuk } n \geq 1.$$

$$1 + 2 + \dots + n = \Omega(n) \text{ karena } 1 + 2 + \dots + n \geq 1 + 1 + \dots + 1 = n \text{ untuk } n \geq 1.$$

Kita tidak dapat menurunkan notasi Θ -Besar untuk $1 + 2 + \dots + n$ karena batas bawah n tidak sama dengan batas atas n^2 . Satu cara untuk mendapatkan batas bawah yang sama dengan batas atas adalah dengan menghilangkan setengah pertama dari suku-suku deret. Dengan menjumlahkan hanya suku-suku yang lebih besar dari $\lceil n/2 \rceil$, kita mendapatkan

$$\begin{aligned} 1 + 2 + \dots + n &\geq \lceil n/2 \rceil + \dots + (n-1) + n \\ &\geq \lceil n/2 \rceil + \dots + \lceil n/2 \rceil + \lceil n/2 \rceil \\ &= \lceil (n+1)/2 \rceil \lceil n/2 \rceil \\ &\geq (n/2)(n/2) \\ &= n^2/4 \end{aligned}$$

Kita menyimpulkan bahwa

$$1 + 2 + \dots + n = \Omega(n^2)$$

Oleh karena itu,

$$1 + 2 + \dots + n = \Theta(n^2)$$

■

Sebuah fakta yang berguna untuk menentukan orde kompleksitas adalah dari suku tertinggi di dalam polinomial. Sebagai contoh, bila $T(n) = 3n^4 + 6n^3 + 18n + 2$, maka $T(n)$ adalah berorde n^4 (yaitu $O(n^4)$, $\Omega(n^4)$, dan $\Theta(n^4)$). Ini dikatakan dengan teorema berikut:

TEOREMA 10.3. Bila $T(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$ adalah polinom derajat m maka $T(n)$ adalah berorde n^m .

Contoh 10.20

Tentukan notasi O , Ω dan Θ untuk algoritma pengurutan seleksi (*selection sort*).

Penyelesaian:

Kompleksitas waktu algoritma pengurutan seleksi sudah dihitung pada Contoh 10.6, yaitu

$$T(n) = \frac{n(n-1)}{2} = n^2/2 - n/2. \text{ Menurut Teorema 10.3, algoritma ini berorde } n^2. \text{ Jadi,}$$

kompleksitas asimtotik algoritma pengurutan seleksi adalah $O(n^2)$, $\Omega(n^2)$, dan $\Theta(n^2)$. ■

10.6 Ragam Soal dan Penyelesaian

Contoh 10.21

Di bawah ini adalah algoritma (dalam notasi Pascal-like) untuk menguji apakah dua buah matriks, A dan B , yang masing-masing berukuran $n \times n$, sama.

```
function samaMatriks(A, B : matriks; n : integer) → boolean  
( true jika A dan B sama; sebaliknya false jika  $A \neq B$  )
```

Deklarasi

i, j : integer

Algoritma:

```
for i ← 1 to n do  
  for j ← 1 to n do  
    if  $A_{i,j} \neq B_{i,j}$  then  
      return false  
    endif  
  endfor  
endfor  
return true
```

- (a) Apa kasus terbaik dan terburuk untuk algoritma di atas?
- (b) Tentukan kompleksitas waktu terbaik dan terburuk dalam notasi O .

Penyelesaian:

- (a) Kasus terbaik adalah jika elemen pertama dari kedua buah matriks, yaitu A_{11} dan B_{11} tidak sama sehingga operasi perbandingan elemen-elemen matriks cukup dilakukan sekali saja (fungsi mengembalikan nilai *false*).

Kasus terburuk adalah jika kedua matriks sama, yaitu seluruh elemen matriks A dan B pada posisi yang bersesuaian sama (A_{ij} dan B_{ij} untuk semua i dan j).

- (b) Operasi dasar yang dihitung adalah operasi perbandingan elemen-elemen kedua buah matriks. Kompleksitas waktu terbaik adalah $O(1)$ karena hanya ada satu kali operasi perbandingan elemen. Pada kasus terburuk, seluruh elemen matriks dibandingkan sehingga jumlah operasi perbandingan elemen matriks adalah sebanyak $T(n) = n \cdot n = n^2$ kali. Jadi, kompleksitas waktu untuk kasus terburuk adalah $O(n^2)$. ■

Contoh 10.22

Berapa kali instruksi *assignment* pada potongan program dalam notasi Bahasa Pascal di bawah ini dieksekusi? Tentukan juga notasi O-besar.

```
for i := 1 to n do  
  for j := 1 to n do  
    for k := 1 to j do  
      x := x + 1;
```

Penyelesaian:

Kalang terluar (kalang i) dieksekusi n kali. Untuk setiap nilai i , kalang terluar kedua (kalang j) dieksekusi n kali. Untuk setiap nilai j , kalang terdalam (kalang k) dieksekusi j kali, $j = 1, 2, 3, \dots, n$, sehingga jumlah pengulangan seluruhnya adalah

$$T(n) = n(1 + 2 + 3 + \dots + n) = n^2(n+1)/2$$

dalam notasi O -besar:

$$T(n) = n^2(n+1)/2 \cdot O(1) = O(n^3)$$

Contoh 10.23

Untuk soal (a) dan (b) berikut, tentukan C , $f(n)$, n_0 , dan notasi O -besar sedemikian sehingga $T(n) = O(f(n))$ jika $T(n) \leq C \cdot f(n)$ untuk semua $n \geq n_0$:

(a) $T(n) = 2 + 4 + 6 + \dots + 2n$

(b) $T(n) = (n+1)(n+3)/(n+2)$

Penyelesaian:

(a) $2 + 4 + 6 + \dots + 2n = 2(1 + 2 + 3 + \dots + n)$
 $\leq 2(n + n + n + \dots + n) = 2(n^2)$ untuk $n \geq 1$

sehingga $C = 2$, $f(n) = n^2$, $n_0 = 1$, didapat $T(n) = O(n^2)$

(b) $(n+1)(n+3)/(n+2) = (n^2 + n + 3)/(n+2)$
 $\leq n + n = 2n$ untuk $n \geq 2$

sehingga $C = 2$, $f(n) = n$, $n_0 = 2$, didapat $T(n) = O(n)$

atau dengan pendekatan lain:

perhatikan $(n+1)/(n+2)$ akan mendekati 1 untuk n yang besar
jadi $(n+1)(n+3)/(n+2)$ akan tumbuh sebagaimana $(n+3)$ tumbuh
kompleksitas $(n+3) = O(n)$ karena $n+3 \leq n+3n = 4n$ untuk $n \geq 1$

sehingga $C = 4$, $f(n) = n$, $n_0 = 1$, dan didapat kompleksitas asimptotik yang sama, $T(n) = O(n)$. ■

Contoh 10.24

[AZM88] Tentukan kompleksitas waktu potongan algoritma di bawah ini dalam notasi O -Besar.

```
ukuran ← m
i ← 1
while i < n do
  i ← i + 1
  ProcA(i)      { prosedur ProcA membutuhkan waktu O(n) }
  if ukuran > 0 then
    s ← sebuah nilai di dalam rentang 1 .. ukuran
    ukuran ← ukuran - s
```

```

for j ← 1 to s do
  ProcB(s)      { prosedur ProcB membutuhkan waktu O(n) }
endfor
endif
endwhile

```

Penyelesaian:

Kalang *while* dieksekusi sebanyak n kali, karena i dimulai dari 1 dan nilai i selalu bertambah 1 di dalam kalang. Jadi, ProcA dieksekusi sebanyak n kali, sehingga total kebutuhan waktu untuk ProcA adalah $(n - 1) \times O(n)$. Kita tidak dapat menentukan jumlah pengulangan untuk kalang *for* (di dalam kalang *while*) sebagai fungsi dari i . Tetapi, jika s_i adalah nilai s di dalam kalang *while* untuk suatu nilai i , maka jumlah pengulangan untuk ProcB adalah sebanyak $\sum_i s_i$. Jumlah nilai s_i adalah $\sum_i s_i = m$,

karena ukuran diinisialisasi dengan m dan pada setiap pengulangan kalang *while* ukuran dikurangi dengan s_i . Total kebutuhan waktu untuk ProcB adalah $m \times O(n)$. Instruksi yang lain seperti *assignment*, perbandingan ($i < n$ dan ukuran > 0), dan sebagainya membutuhkan waktu $O(1)$. Dengan demikian, kebutuhan waktu potongan algoritma di atas adalah:

$$(n - 1) \times O(n) + m \times O(n) + O(1) = O(n^2) + O(mn) + O(1) = O(n^2 + mn) \quad \blacksquare$$

Soal Latihan

1. Untuk $T(n) = 2 + 4 + 8 + 16 + \dots + 2^n$, tentukan C , $f(n)$, n_0 , dan notasi O -besar sedemikian sehingga $T(n) = O(f(n))$ jika $T(n) \leq C \cdot f(n)$ untuk semua $n \geq n_0$.
2. Untuk tiap fungsi berikut, tentukan $f(n)$ sedemikian fungsi tersebut adalah $O(f(n))$:
 - (a) $17n^3 + 1.7n^2 + n$
 - (b) $2n^2 + n \log n$
 - (c) $3n + n \log n$
 - (d) $2^n + n^9$
 - (e) $(n + \sqrt{n})(\log n + n + 3)$
3. Tentukan kompleksitas waktu asimptotik kode program berikut:

```
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      wij ← wij or wik and wkj
    endfor
  endfor
endfor
```
5. Bandingkan fungsi n^2 dan $2^n/4$ untuk bermacam-macam nilai n . Tentukan kapan fungsi kedua lebih besar daripada fungsi pertama?
6. Perhatikan bahwa $T(n) = 1^k + 2^k + \dots + n^k = O(n^{k+1})$
7. Temukan $f(n)$, C , dan n_0 sedemikian sehingga $T(n)$ di bawah ini adalah $O(f(n))$. Untuk fungsi f di dalam $O(f(n))$, gunakan fungsi f yang sederhana dengan pangkat sekecil mungkin:
 - (a) $T(n) = 5n^3 + (\log n)^4$
 - (b) $T(n) = (n^4 + 3 \log n)/(n^4 + 2)$
 - (c) $T(n) = n \log(n^2 + 1) + n^2 \log n$
8. Selesaikan tiga soal di bawah ini:
 - (a) Perhatikan bahwa $n^2 + 5n + 13$ adalah $O(n^3)$ tetapi n^3 bukan $O(n^2 + 5n + 13)$
 - (b) Perhatikan bahwa 2^n adalah $O(3^n)$ tetapi 3^n bukan $O(2^n)$
9. Perhatikan bahwa $n! = O(n^n)$

10. Perhatikan bahwa kesamaan berikut tidak benar:

- (a) $10n^2 + 9 = O(n)$
- (b) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$

11. Tulislah notasi Tetha-Besar untuk setiap fungsi di bawah ini:

- (a) $T(n) = 6n^3 + 12n^2 + 1$
- (b) $T(n) = 3n^2 + 2n \log n$

12. Berapa kali pernyataan $x \leftarrow x + 1$ pada algoritma di bawah ini dilaksanakan? Nyatakan kompleksitas waktu asimptotiknya dalam notasi O -Besar, Θ -Besar, dan Ω -Besar.

(i) $i \leftarrow 2$
while $i < n$ do
 $i \leftarrow i * i$
 $x \leftarrow x + 1$
end

(ii) for $i \leftarrow 1$ to $2n$ do
 $x \leftarrow x + 1$
endfor

(iii) $i \leftarrow 1$
while $i < 2n$ do
 $x \leftarrow x + 1$
 $i \leftarrow i + 2$
end

(iv) for $i \leftarrow 1$ to $2n$ do
for $j \leftarrow 1$ to n do
 $x \leftarrow x + 1$
endfor
endfor

(v) $i \leftarrow n$
while $i \geq 1$ do
 $x \leftarrow x + 1$
 $i \leftarrow i \text{ div } 2$
endwhile

(iv) for $i \leftarrow 1$ to n do
for $j \leftarrow 1$ to $i \text{ div } 2$ do
 $x \leftarrow x + 1$
endfor
endfor

13. Tulislah algoritma untuk menentukan nilai maksimum sekaligus nilai minimum larik *integer*. Ukuran larik adalah n elemen. Berapa kompleksitas waktunya? Berapa kompleksitas waktu asimptotiknya dalam notasi O -Besar, Θ -Besar, dan Ω -Besar.

14. Tulislah algoritma untuk menjumlahkan dua buah matriks yang masing-masing berukuran $n \times n$. Berapa kompleksitas waktunya? Berapa kompleksitas waktu asimptotiknya dalam notasi O -Besar, Θ -Besar, dan Ω -Besar?

15. Tulislah algoritma untuk menyalin (*copy*) sebuah larik ke larik yang lain. Ukuran larik adalah n elemen. Berapa kompleksitas waktunya? Berapa kompleksitas waktu asimptotiknya dalam notasi O -Besar, Θ -Besar, dan Ω -Besar.

16. Pecahkan relasi rekurens berikut:

$$T(n) = \begin{cases} a, & n = 0 \\ b + nT(n-1), & n > 0 \end{cases}$$

17. Menghitung perpangkatan a^n , $a \in R$ dan n adalah bilangan bulat (asumsi: n adalah perpangkatan dari 2, atau $n = 2^k$), dapat dilakukan dengan dua buah algoritma di bawah ini (dalam notasi Bahasa Pascal):

(i) Algoritma pertama (iteratif)

$$a^n = a \cdot a \cdot a \dots a$$

(sebanyak n kali)

```
function p1(a:real;n:integer):real;
{ menghitung a^n = a*a*a* ...*a }
var
  k : integer;
  p : real;
begin
  p:=a;
  for k:=2 to n do
    p:=p*a;
  {endfor}
  p1:=p;
end;
```

(ii) Algoritma kedua (rekursif)

$$\begin{aligned} a^n &= 1 && \text{jika } n = 0 \\ &= a^{n/2} \cdot a^{n/2} && \text{jika } n > 0 \text{ dan } n \text{ genap} \\ &= a \cdot a^{n/2} \cdot a^{n/2} && \text{jika } n > 0 \text{ dan } n \text{ ganjil} \end{aligned}$$

```
function p2(a:real;n:integer):real;
{menghitung a^n=a^(n/2)*a^(n/2) }
begin
  if n=0 then
    p2:=1
  else
    if odd(n) then
      p2:=sqr(p2(a,n div 2))*a;
    else
      p2:=sqr(p2(a,n div 2))
    {endif}
  end;
```

Keterangan: sqr adalah fungsi kuadrat

(a) Hitung kompleksitas waktu $T(n)$ dan waktu asimptotik $O(f(n))$ masing-masing algoritma pertama dan kedua berdasarkan jumlah operasi perkalian.

(b) Algoritma manakah yang lebih mangkus/cepat untuk n yang besar?

18. Diberikan algoritma pengurutan *bubble-sort* seperti berikut ini:

```

procedure BubbleSort(input/output  $a_1, a_2, \dots, a_n$ ; integer)
{ Mengurut tabel integer TabInt[1..n] dengan metode pengurutan bubble-
sort
Masukan:  $a_1, a_2, \dots, a_n$ 
Keluaran:  $a_1, a_2, \dots, a_n$  (terurut menaik)
}

```

Deklarasi

```

k : integer      ( indeks untuk traversal tabel )
pass : integer   ( tahapan pengurutan )
temp : integer   ( peubah bantu untuk pertukaran elemen tabel )

```

Algoritma

```

for pass ← 1 to n - 1 do
  for k ← n downto pass + 1 do
    if  $a_k < a_{k-1}$  then
      { pertukarkan  $a_k$  dengan  $a_{k-1}$  }
      temp ←  $a_k$ 
       $a_k$  ←  $a_{k-1}$ 
       $a_{k-1}$  ← temp
    endif
  endfor
endfor

```

- Hitung berapa jumlah operasi perbandingan elemen-elemen tabel.
- Berapa kali maksimum operasi pertukaran elemen-elemen tabel dilakukan?
- Hitung kembali kompleksitas waktu asimptotik algoritma pengurutan *bubble-sort*.

19. Sepotong algoritma disajikan di bawah ini:

```

for i ← 1 to n do
  for j ← 1 to i do
    for k ← 1 to j do
       $x \leftarrow x + 1$ 
    endfor
  endfor
endfor

```

- Jika $T(n)$ dihitung dari operasi penjumlahan pada pernyataan $x \leftarrow x + 1$, tentukan $T(n)$.
- Nyatakan $T(n)$ dalam notasi O -besar, Ω -besar, dan Θ -besar.

20. Pertanyaan yang sama dengan soal nomor 16 untuk potongan algoritma berikut ini:

```

j ← n
while j ≥ 1 do
  for i ← 1 to j do
     $x \leftarrow x + 1$ 
  endfor
  j ← j div 2
endwhile

```

21. Algoritma di bawah ini menghitung nilai polinom untuk $x = t$:

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

```
function p(input x: real) → real
{ Mengembalikan nilai p(x) }
```

Deklarasi

```
j, k : integer
jumlah, suku : real
```

Algoritma

```
jumlah ← a0
for j ← 1 to n do
  { hitung ajxj }
  suku ← aj
  for k ← 1 to j do
    suku ← suku * x
  endfor
  jumlah ← jumlah + suku
endfor
return jumlah
```

Hitunglah berapa operasi perkalian dan berapa operasi penjumlahan yang dilakukan oleh algoritma di atas? Jumlahkan kedua hitungan tersebut, lalu tentukan juga kompleksitas waktu asimptotik algoritma tersebut dalam notasi *O*-Besar.

Algoritma mengevaluasi polinom yang lebih baik dapat dibuat dengan metode Horner berikut:

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \dots + x(a_{n-1} + a_nx)))) \dots)$$

```
function p2(input x : real) → real
{ Mengembalikan nilai p(x) dengan metode Horner }
```

Deklarasi

```
k : integer
b1, b2, ..., bn : real
```

Algoritma

```
bn ← an
for k ← n - 1 downto 0 do
  bk ← ak + bk+1 * x
endfor
return b0
```

O -Besar. Manakah yang terbaik, algoritma p atau p^2 ?

Daftar Pustaka¹

- [AND79] Anderson, Robert B., *Proving Programs Correct*, John Wiley & Sons, 1979.
- [BRA88] Brassad, Gilles & Paul Bratley, *Algorithmics, Theory and Practice*, Prentice Hall, 1988
- [AZM88] Azmoodeh, Manoochehr, *Abstract Data Types and Algorithms*, Macmillan Education, 1988
- [DEO74] Deo, Narshing, *Graph Theory with Applications To Engineering and Computer Science*, Prentice-Hall International, 1974
- [DOE85] Doerr, Alan & Kenneth Levasseur, *Applied Discrete Structures for Computer Science*, SRA Associates, 1985
- [DUL94] Dulimarta, Hans Sudiana, *Catatan Kuliah Teori Graph*, Teknik Informatika ITB, 1994
- [GIB85] Gibbons, Alan, *Algorithmic Graph Theory*, Cambridge University Press, 1985

¹ Disusun menurut urutan abjad nama kedua dari pengarang buku

- [HOR76] Horowitz, Ellis & Sartaj Sahni, *Fundamental of Data Structures*, Pitman Publishing Limited, 1976.
- [HOR78] Horowitz, Ellis & Sartaj Sahni, *Fundamental of Computer Algorithms*, Pitman Publishing Limited, 1978.
- [JOH97] Johnsonbaugh, Richard, *Discrete Mathematics*, 4th, Pentice Hall, 1997
- [LIU85] Liu, C.L, *Element of Discrete Mathematics*, McGraw-Hill, Inc, 1985.
- [MAN82] Mano, M. Moris, *Computer System Architecture 2nd*, Prentice-Hall International, 1982
- [NEA96] Neapolitan, Richard E. & Kumarss Naimipour, *Foundations of Algorithms*, D. C. Heath and Company, 1996.
- [ROS99] Rosen, Kenneth H., *Discrete Mathematics and Its Applications*, 4th, McGraw-Hill International 1999.
- [RHE94] Rhee, Man Young, *Cryptography and Secure Communications*, McGraw-Hill, 1994.
- [SAN93] Santoso, Judhi S. *Catatan Kuliah Teori Graph dan Aplikasinya*, Teknik Informatika ITB, 1993.
- [SCH96] Schneier, Bruce, *Aplied Cryptography 2nd*, John Wiley & Sons, 1996
- [SED92] Sedgewick, Robert, *Algorithms in C++*, Addison-Wesley Publishing Company, 1992.
- [LIP92] Lipschutz, Seymour & Marc Lars Lipson, *2000 Solved Problems in Discrete Mathematics*, McGraw-Hill, 1992.