Kompleksitas Algoritma

Pendahuluan

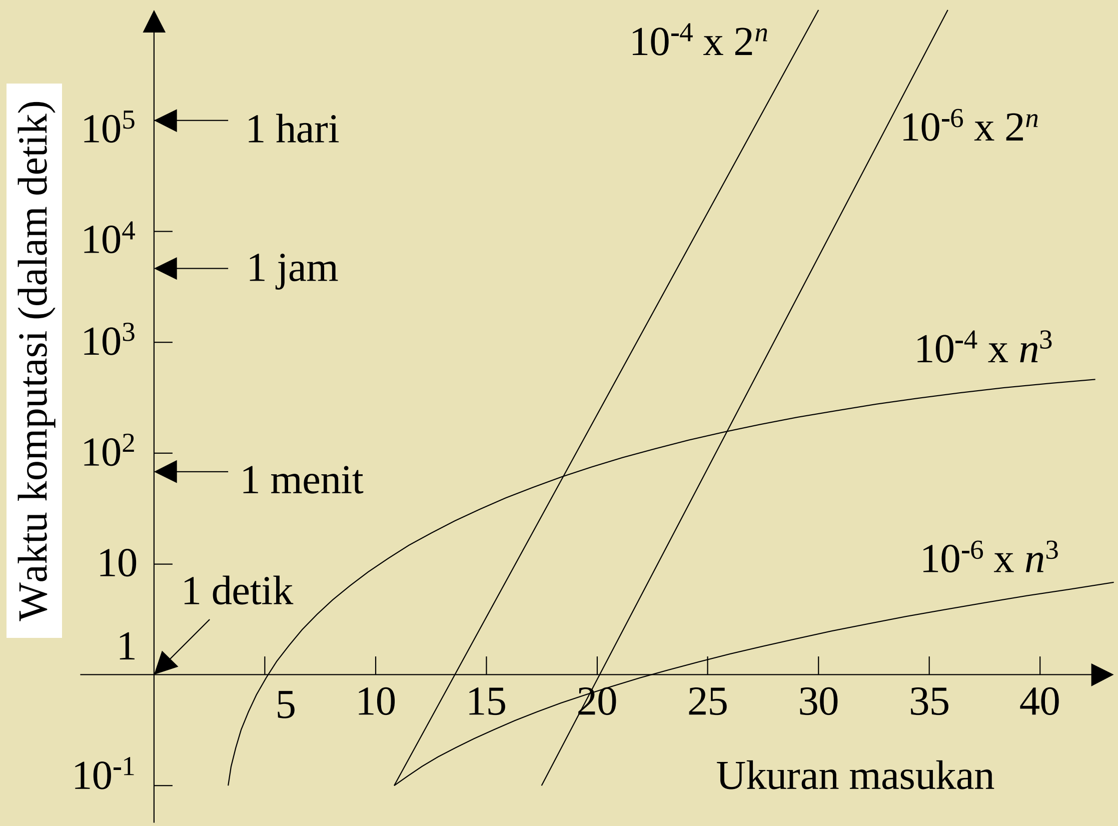
- Sebuah algoritma tidak saja harus benar, tetapi juga harus mangkus (*efisien*).
- Algoritma yang bagus adalah algoritma yang mangkus.
- Kemangkusan algoritma diukur dari berapa jumlah waktu dan ruang (*space*) memori yang dibutuhkan untuk menjalankannya.

- Algoritma yang mangkus ialah algoritma yang meminimumkan kebutuhan waktu dan ruang.

- Kebutuhan waktu dan ruang suatu algoritma bergantung pada ukuran masukan (n), yang menyatakan jumlah data yang diproses.

- Kemangkusan algoritma dapat digunakan untuk menilai algoritma yang bagus.

- Mengapa kita memerlukan algoritma yang mangkus? Lihat grafik di bawah ini.



Model Perhitungan Kebutuhan Waktu/Ruang

- Kita dapat mengukur waktu yang diperlukan oleh sebuah algoritma dengan menghitung banyaknya operasi/instruksi yang dieksekusi.

- Jika kita mengetahui besaran waktu (dalam satuan detik) untuk melaksanakan sebuah operasi tertentu, maka kita dapat menghitung berapa waktu sesungguhnya untuk melaksanakan algoritma tersebut.

Contoh 1. Menghitung rerata

a_1	a_2	a_3	…	a_n

Larik bilangan bulat

```
procedure HitungRerata(input a1, a2, ..., an : integer, output
r : real)
{ Menghitung nilai rata-rata dari sekumpulan elemen larik integer a1, a2,
..., an.
  Nilai rata-rata akan disimpan di dalam peubah r.
  Masukan: a1, a2, ..., an
  Keluaran: r (nilai rata-rata)
}
Deklarasi
   k : integer
   jumlah : real

Algoritma
   jumlah←0
   k←1
   while k ≤ n do
     jumlah←jumlah + ak
     k←k+1
   endwhile
   { k > n }
   r ← jumlah/n   { nilai rata-rata }
```

(i) Operasi pengisian nilai (`jumlah`←`0`, `k`←`1`, `jumlah`←`jumlah+a`$_k$, `k`←`k+1`, dan `r` ← `jumlah/n`)

Jumlah seluruh operasi pengisian nilai adalah

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

(ii) Operasi penjumlahan (`jumlah+a`$_k$, dan `k+1`)

Jumlah seluruh operasi penjumlahan adalah

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

(iii) Operasi pembagian (`jumlah/n`)

Jumlah seluruh operasi pembagian adalah

$$t_3 = 1$$

Total kebutuhan waktu algoritma `HitungRerata`:

$$t = t_1 + t_2 + t_3 = (3 + 2n)\mathrm{a} + 2nb + c \text{ detik}$$

Model perhitungan kebutuhan waktu seperti di atas kurang berguna, karena:

1. Dalam praktek kita tidak mempunyai informasi berapa waktu sesungguhnya untuk melaksanakan suatu operasi tertentu

2. Komputer dengan arsitektur yang berbeda akan berbeda pula lama waktu untuk setiap jenis operasinya.

- Model abstrak pengukuran waktu/ruang harus independen dari pertimbangan mesin dan *compiler* apapun.

- 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, $T(n)$, diukur dari jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n.

- Kompleksitas ruang, $S(n)$, diukur dari 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.

Kompleksitas Waktu

- Dalam praktek, kompleksitas waktu dihitung berdasarkan jumlah operasi abstrak yang *mendasari* suatu algoritma, dan memisahkan analisisnya dari implementasi.

- **Contoh 2.** Tinjau algoritma menghitung rerata pada Contoh 1. Operasi yang mendasar pada algoritma tersebut adalah operasi penjumlahan elemen-elemen a_k (yaitu $\texttt{jumlah} \leftarrow \texttt{jumlah+a}_\texttt{k}$),

- Kompleksitas waktu HitungRerata adalah $T(n) = n$.

Contoh 3. Algoritma untuk mencari elemen terbesar di dalam sebuah larik (*array*) yang berukuran *n* elemen.

```
procedure CariElemenTerbesar(input a1, a2, ..., an : integer, output
maks : integer)
{ Mencari elemen terbesar dari sekumpulan elemen larik integer a1, a2,
..., an.
  Elemen terbesar akan disimpan di dalam maks.
  Masukan: a1, a2, ..., an
  Keluaran: maks (nilai terbesar)
}
Deklarasi
   k : integer

Algoritma
   maks←a1
   k←2
   while k ≤ n do
     if ak > maks then
        maks←ak
     endif
     k←k+1
   endwhile
   { k > n }
```

Kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi perbandingan elemen larik (`A[i] > maks`).

Kompleksitas waktu `CariElemenTerbesar` : $T(n) = n - 1$.

Kompleksitas waktu dibedakan atas tiga macam :

1. $T_{max}(n)$: kompleksitas waktu untuk kasus terburuk *(worst case)*,
   → kebutuhan waktu maksimum.

2. $T_{min}(n)$: kompleksitas waktu untuk kasus terbaik *(best case)*,
   → kebutuhan waktu minimum.

3. $T_{avg}(n)$: kompleksitas waktu untuk kasus rata-rata *(average case)*
   → kebutuhan waktu secara rata-rata

Contoh 4. Algoritma *sequential search*.

```
procedure PencarianBeruntun(input a1, a2, ..., an : integer, x : integer,
                            output idx : integer)
Deklarasi
  k : integer
  ketemu : boolean   { bernilai true jika x ditemukan atau false jika x
tidak ditemukan }

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

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

Jumlah operasi perbandingan elemen tabel:

1. *Kasus terbaik*: ini terjadi bila $a_1 = x$.

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

2. *Kasus terburuk*: bila $a_n = x$ atau x tidak ditemukan.

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

3. *Kasus rata-rata*: Jika x ditemukan pada posisi ke-j, maka operasi perbandingan (`ak = x`) akan dieksekusi sebanyak j kali.

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

Contoh 5. Algoritma pencarian biner (*bynary search*).

```
procedure PencarianBiner(input a1, a2, ..., an : integer, x : integer,
                         output idx : integer)
Deklarasi
   i, j, mid : integer
   ketemu : boolean

Algoritma
   i←1
   j←n
   ketemu←false
   while (not ketemu) and ( i ≤ j) do
      mid ← (i+j) div 2
      if amid = x then
       ketemu ← true
      else
         if amid < x then      { cari di belahan kanan }
         i←mid + 1
         else                  { cari di belahan kiri }
         j←mid - 1;
         endif
      endif
   endwhile
   {ketemu or i > j }

   if ketemu then
      idx←mid
   else
      idx←0
   endif
```

1. *Kasus terbaik*

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

2. *Kasus terburuk*:

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

Contoh 6. Algoritma algoritma pengurutan seleksi (*selection sort*).

```
procedure Urut(input/output a1, a2, ..., an : integer)
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 aj > aimaks then
         imaks←j
      endif
    endfor
    { pertukarkan aimaks dengan ai }
    temp←ai
    ai←aimaks
    aimaks←temp

  endfor
```

(i) Jumlah operasi perbandingan elemen

Untuk setiap *pass* ke-i,

$i = n$ $\rightarrow$ jumlah perbandingan $= n - 1$

$i = n - 1$ $\rightarrow$ jumlah perbandingan $= n - 2$

$i = n - 2$ $\rightarrow$ jumlah perbandingan $= n - 3$

$\vdots$

$i = 2$ $\rightarrow$ jumlah perbandingan $= 1$

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah

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

Ini adalah kompleksitas waktu untuk kasus terbaik dan terburuk, karena algoritma `Urut` tidak bergantung pada batasan apakah data masukannya sudah terurut atau acak.

(ii) Jumlah operasi pertukaran

Untuk setiap i dari 1 sampai n – 1, terjadi satu kali pertukaran elemen, sehingga jumlah operasi pertukaran seluruhnya adalah

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

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

Kompleksitas Waktu Asimptotik

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

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

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

- $T(n)$ tumbuh seperti n^2 tumbuh saat n bertambah. Kita katakan bahwa $T(n)$ berorde n^2 dan kita tuliskan

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

Notasi "O" disebut notasi "O-Besar" (*Big*-O) yang merupakan notasi **kompleksitas waktu asimptotik**.

DEFINISI. $T(n) = O(f(n))$ (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

$$T(n) \le C(f(n))$$

untuk $n \ge n_0$.

$f(n)$ adalah batas lebih atas (*upper bound*) dari $T(n)$ untuk n yang besar.

Contoh 7. Tunjukkan bahwa $T(n) = 3n + 2 = O(n)$.

Penyelesaian:

$3n + 2 = O(n)$

karena

$3n + 2 \le 3n + 2n = 5n$ untuk semua $n \ge 1$ ($C = 5$ dan $n_0 = 1$).

Contoh 8. Tunjukkan bahwa $T(n) = 2n^2 + 6n + 1 = O(n^2)$.

Penyelesaian:

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

karena

$2n^2 + 6n + 1 \le 2n^2 + 6n^2 + n^2 = 9n^2$ untuk semua $n \ge 1$ (C =9 dan $n_0 = 1$).

atau karena

$2n^2 + 6n + 1 \le n^2 + n^2 + n^2 = 3n^2$ untuk semua $n \ge 6$ (C =3 dan $n_0 = 6$).

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

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

(*a*) $T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n))$

(*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*) $f(n) = O(f(n))$

Contoh 9. Misalkan $T_1(n) = O(n)$ dan $T_2(n) = O(n^2)$, maka

(a) $T_1(n) + T_2(n) = O(\max(n, n^2)) = O(n^2)$

(b) $T_1(n)T_2(n) = O(n.n^2) = O(n^3)$

Contoh 10. $O(5n^2) = O(n^2)$

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

Aturan Untuk Menentukan Kompleksitas Waktu Asimptotik

1. Jika kompleksitas waktu $T(n)$ dari algoritma diketahui,

   Contoh: (i) pada algoritma `cari_elemen_terbesar`

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

   (ii) pada algoritma `pencarian_beruntun`

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

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

   $T_{avg}(n) = (n + 1)/2 = O(n)$,

   (iii) pada algoritma `pencarian_biner`,

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

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

   (iv) pada algoritma `selection_sort`

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

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

Penjelasannya adalah sebagai berikut:

$$T(n) = (n + 2) \log(n^2 + 1) + 5n^2$$
$$= f(n)g(n) + h(n),$$

Kita rinci satu per satu:

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

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

$$\log(n^2 + 1) \le \log(2n^2) = \log 2 + \log n^2$$
$$= \log 2 + 2 \log n \le 3 \log n \text{ untuk } n > 2$$

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

maka

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

2. Menghitung O-Besar untuk setiap instruksi di dalam algoritma dengan panduan di bawah ini, kemudian menerapkan teorema O-Besar.

(a) Pengisian nilai (*assignment*), perbandingan, operasi aritmetik, *read*, *write* membutuhkan waktu $O(1)$.

(b) Pengaksesan elemen larik atau memilih *field* tertentu dari sebuah *record* membutuhkan waktu $O(1)$.

Contoh:

read(x);	$O(1)$
x:=x + a[k];	$O(1) + O(1) + O(1) = O(1)$
writeln(x);	$O(1)$

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

Penjelasan: $O(1) + O(1) + O(1) = O(\max(1,1)) + O(1)$

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

(c) **`if`** `C` **`then`** `S1` **`else`** `S2`; membutuhkan waktu

$$T_C + \max(T_{S1}, T_{S2})$$

<u>Contoh</u>:

```
read(x);                          O(1)
if x mod 2 = 0 then               O(1)
 begin
   x:=x+1;                        O(1)
   writeln(x);                    O(1)
 end
else
  writeln(x);                     O(1)
```

Kompleksitas waktu asimptotik:

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

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

$= O(1) + O(1)$

$= O(1)$

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

<u>Contoh</u>

```
for i:=1 to n do
  jumlah:=jumlah + a[i];      O(1)
```

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

$= O(n \cdot 1)$

$= O(n)$

Contoh: kalang bersarang

```
for i:=1 to n do
   for j:=1 to n do
    a[i,j]:=0;            O(1)
```

Kompleksitas waktu asimptotik:

$$nO(n) = O(n.n) = O(n^2)$$

Contoh: kalang bersarang dengan dua buah instruksi

```
for i:=1 to n do
   for j:=1 to i do
    begin
      a:=a+1;        O(1)
      b:=b-2         O(1)
    end;
```

waktu untuk a:=a+1 : $O(1)$

waktu untuk b:=b-2 : $O(1)$

total waktu untuk badan kalang = $O(1) + O(1) = O(1)$

kalang terluar dieksekusi sebanyak n kali

kalang terdalam dieksekusi sebanyak i kali, $i = 1, 2, \ldots, n$

jumlah pengulangan seluruhnya = $1 + 2 + \ldots + n$

$= n(n + 1)/2$

kompleksitas waktu asimptotik = $n(n + 1)/2 . O(1)$

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

(e) `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.

<u>Contoh</u>: kalang tunggal sebanyak n-1 putaran

```
i:=2;                         O(1)
while i <= n do               O(1)
 begin
   jumlah:=jumlah + a[i];     O(1)
   i:=i+1;                    O(1)
 end;
```

Kompleksitas waktu asimptotiknya adalah

$= O(1) + (n-1)\ \{ O(1) + O(1) + O(1) \}$

$= O(1) + (n-1)\ O(1)$

$= O(1) + O(n-1)$

$= O(1) + O(n)$

$= O(n)$

Contoh: 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^.lalu
{ p = Nil or ketemu }
```

Di sini, pengulangan akan berhenti bila *x* yang dicari ditemukan di dalam senarai. Jika jumlah elemen senarai adalah *n*, maka kompleksitas waktu terburuknya adalah $O(n)$ -yaitu kasus *x* tidak ditemukan.

Pengelompokan Algoritma Berdasarkan Notasi *O*-Besar

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

Urutan spektrum kompleksitas waktu algoritma adalah :

$$\underbrace{O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \ldots}_{\text{algoritma polinomial}} < \underbrace{O(2^n) < O(n!)}_{\text{algoritma eksponensial}}$$

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. Contohnya prosedur `tukar` di bawah ini:

```
procedure tukar(var a:integer; var b:integer);
var
  temp:integer;
begin
  temp:=a;
  a:=b;
  b:=temp;
end;
```

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 tetapan.

$O(n)$ Algoritma yang waktu pelaksanaannya lanjar 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 semual, maka n log n menjadi dua kali semula (tetapi tidak terlalu banyak)

$O(n^2)$ Algoritma yang waktu pelaksanaannya kuadratik hanya praktis digunakan untuk persoalana 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.

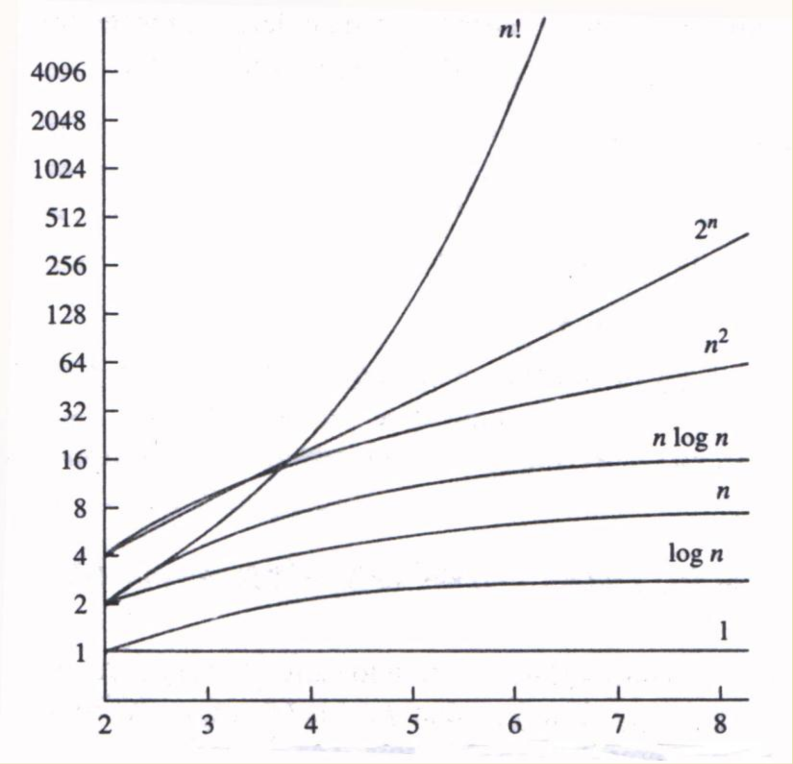
$O(n^3)$ Seperti halnya algoritma kuadratik, algoritma kubik memproses setiap masukan dalam tiga buah kalang bersarang, misalnya algoritma perkalian matriks. Bila n = 100, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, waktu pelaksanan algoritma meningkat menjadi delapan kali semula.

$O(2^n)$ Algoritma yang tergolong kelompok ini mencari solusi persoalan secara "*brute force*", misalnya pada algoritma mencari sirkuit Hamilton. 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* . Bila n = 5, maka waktu pelaksanaan algoritma adalah 120. Bila n dijadikan dua kali semula, maka waktu pelaksanaan algoritma menjadi faktorial dari $2n$.

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)



- Sebuah masalah yang mempunyai algoritma dengan kompleksitas polinomial 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.

- 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. Sebaliknya, sebuah masalah dikatakan *intractable* (sukar dari segi komputasi) jika tidak ada algoritma yang mangkus untuk menyelesaikannya.

- Masalah yang sama sekali tidak memiliki algoritma untuk memecahkannya disebut **masalah tak-terselesaikan** (*unsolved problem*). Sebagai contoh, masalah penghentian (*halting problem*) jika diberikan program dan sejumlah masukan, apakah program tersebut berhenti pada akhirnya.

- Kebanyakan masalah yang tidak 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 terbuktikan, bahwa tidak ada masalah di dalam kelas NP-lengkap yang dapat dipecahkan dalam waktu polinomial.

Notasi Omega-Besar dan Tetha-Besar

Definisi Ω-Besar adalah:

$T(n) = \Omega(\mathrm{g}(n))$ (dibaca "$T(n)$ adalah Omega ($f(n)$)" yang artinya *T(n) berorde paling kecil g(n)*) bila terdapat tetapan C dan n_0 sedemikian sehingga

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

untuk $n \geq n_0$.

Definisi Θ-Besar,

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

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

Jawab:

Karena $2n^2 + 6n + 1 \geq 2n^2$ untuk $n \geq 1$,
maka dengan $C = 2$ 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: Tentukan notasi notasi O, Ω dan Θ untuk $T(n) = 5n^3 + 6n^2 \log n$.

Jawab:

Karena $0 \le 6n^2 \log n \le 6n^3$,

maka $5n^3 + 6n^2 \log n \le 11n^3$ untuk $n \ge 1$

Dengan mengambil $C = 11$, maka

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

Karena $5n^3 + 6n^2 \log n \ge 5n^3$ untuk $n \ge 1$, maka 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 $5n^3 + 6n^2 \log n = \Theta(n^3)$

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

Latihan Soal

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 ≠ B }

Deklarasi
  i, j : integer

Algoritma:
  for i ← 1 to n do
    for j ← 1 to n do
       if Ai,j ≠ Bi,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*.

2. Berapa kali instruksi *assignment* pada potongan program dalam notas Bhasa 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;
```

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

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

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