



Kompleksitas Algoritma

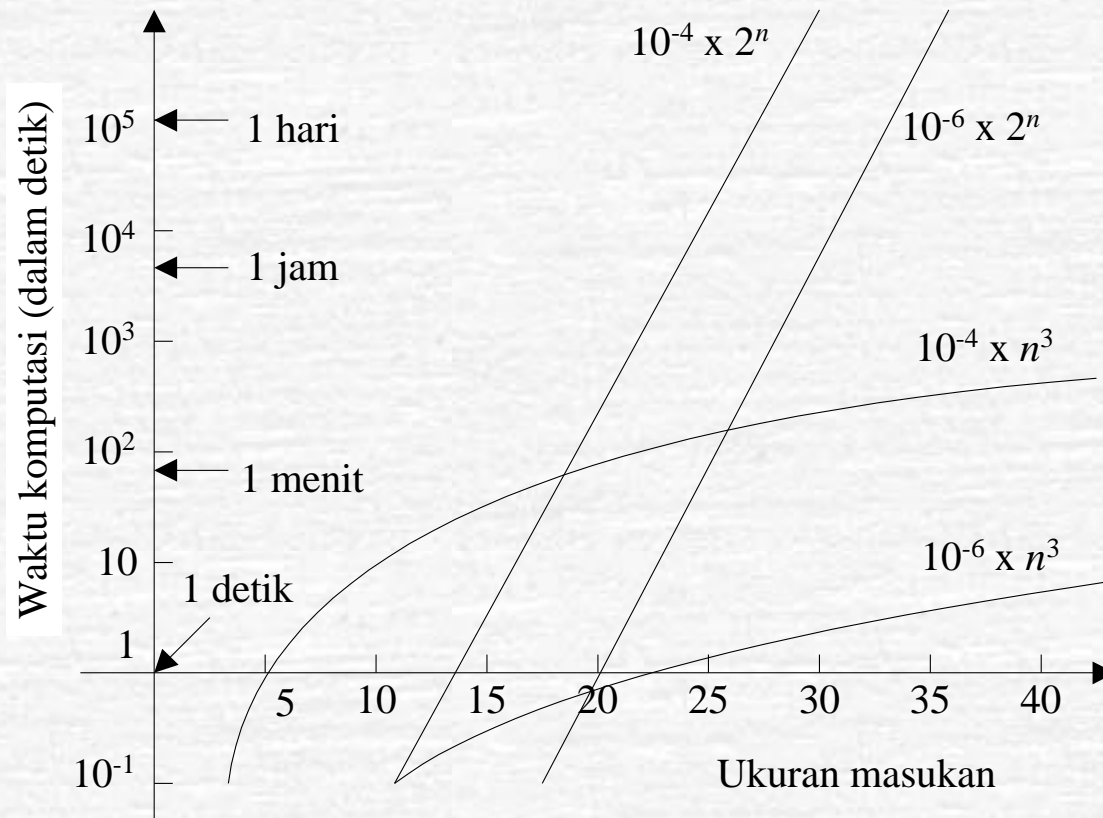
Bahan Kuliah
IF2091 Struktur Diskrit

Pendahuluan

- Sebuah masalah dapat mempunyai banyak algoritma penyelesaian. Contoh: masalah pengurutan (*sort*), ada puluhan algoritma pengurutan
- Sebuah algoritma tidak saja harus benar, tetapi juga harus mangkus (*efisien*).
- Algoritma yang bagus adalah algoritma yang mangkus (*efficient*).
- Kemangkusan algoritma diukur dari waktu (*time*) eksekusi algoritma dan kebutuhan ruang (*space*) memori.

- 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 dari sejumlah algoritma penyelesaian masalah.

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



Model Perhitungan Kebutuhan Waktu

- ☛ Menghitung kebutuhan waktu algoritma dengan mengukur waktu sesungguhnya (dalam satuan detik) ketika algoritma dieksekusi oleh komputer bukan cara yang tepat.
- ☛ Alasan:
 1. Setiap komputer dengan arsitektur berbeda mempunyai bahasa mesin yang berbeda → waktu setiap operasi antara satu komputer dengan komputer lain tidak sama.
 2. *Compiler* bahasa pemrograman yang berbeda menghasilkan kode mesin yang berbeda → waktu setiap operasi antara compiler dengan compiler lain tidak sama.

- 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 .

- Ukuran masukan (n): jumlah data yang diproses oleh sebuah algoritma.
- Contoh: algoritma pengurutan 1000 elemen larik, maka $n = 1000$.
- Contoh: algoritma *TSP* pada sebuah graf lengkap dengan 100 simpul, maka $n = 100$.
- Contoh: algoritma perkalian 2 buah matriks berukuran 50×50 , maka $n = 50$.
- Dalam praktek perhitungan kompleksitas, ukuran masukan dinyatakan sebagai variabel n saja.

Kompleksitas Waktu

- Jumlah tahapan komputasi dihitung dari berapa kali suatu operasi dilaksanakan di dalam sebuah algoritma sebagai fungsi ukuran masukan (n)..
- Di dalam sebuah algoritma terdapat bermacam jenis operasi:
 - Operasi baca/tulis
 - Operasi aritmetika (+, -, *, /)
 - Operasi pengisian nilai (*assignment*)
 - Operasi pengaksesan elemen larik
 - Operasi pemanggilan fungsi/prosedur
 - dll
- Dalam praktek, kita hanya menghitung jumlah operasi khas (tipikal) yang *mendasari* suatu algoritma.

Contoh operasi khas di dalam algoritma

- Algoritma pencarian di dalam larik
Operasi khas: perbandingan elemen larik
- Algoritma pengurutan
Operasi khas: perbandingan elemen, pertukaran elemen
- Algoritma penjumlahan 2 buah matriks
Operasi khas: penjumlahan
- Algoritma perkalian 2 buah matriks
Operasi khas: perkalian dan penjumlahan

- **Contoh 1.** Tinjau algoritma menghitung rerata sebuah larik (*array*).

```
sum ← 0
for i ← 1 to n do
    sum ← sum + a[i]
endfor
rata_rata ← sum/n
```

- Operasi yang mendasar pada algoritma tersebut adalah operasi penjumlahan elemen-elemen a_i (yaitu $\text{sum} \leftarrow \text{sum} + a[i]$) yang dilakukan sebanyak n kali.
- Kompleksitas waktu: $T(n) = n$.

Contoh 2. Algoritma untuk mencari elemen terbesar di dalam sebuah larik (*array*) yang berukuran n elemen.

```
procedure CariElemenTerbesar(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

k : integer

Algoritma

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

Kompleksitas waktu algoritma dihitung berdasarkan jumlah operasi perbandingan elemen larik ($A[i] > \text{maks}$).

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 3. Algoritma *sequential search*.

```
procedure PencarianBeruntun(input  $a_1, a_2, \dots, a_n : \text{integer}$ ,  $x : \text{integer}$ ,  
                           output  $\text{idx} : \text{integer}$ )
```

Deklarasi

```
   $k : \text{integer}$   
   $\text{ketemu} : \text{boolean}$     { bernilai true jika  $x$  ditemukan atau false jika  $x$   
    tidak ditemukan }
```

Algoritma:

```
   $k \leftarrow 1$   
   $\text{ketemu} \leftarrow \text{false}$   
  while ( $k \leq n$ ) and (not  $\text{ketemu}$ ) do  
    if  $a_k = x$  then  
       $\text{ketemu} \leftarrow \text{true}$   
    else  
       $k \leftarrow k + 1$   
    endif  
  endwhile  
  {  $k > n$  or  $\text{ketemu}$  }  
  
  if  $\text{ketemu}$  then    {  $x$  ditemukan }  
     $\text{idx} \leftarrow k$   
  else  
     $\text{idx} \leftarrow 0$       {  $x$  tidak ditemukan }  
  endif
```

Jumlah operasi perbandingan elemen tabel:

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

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

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

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

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

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

Cara lain: asumsikan bahwa $P(a_j = x) = 1/n$. Jika $a_j = x$ maka T_j yang dibutuhkan adalah $T_j = j$. Jumlah perbandingan elemen larik rata-rata:

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

Contoh 4. Algoritma pencarian biner (*binary 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 5. Algoritma pengurutan seleksi (*selection sort*).

```
procedure Urut(input/output  $a_1, a_2, \dots, a_n$  : integer)
```

Deklarasi

```
     $i, j, \text{imaks}, \text{temp}$  : integer
```

Algoritma

```
    for  $i \leftarrow n$  downto 2 do      { pass sebanyak  $n - 1$  kali }
```

```
         $\text{imaks} \leftarrow 1$ 
```

```
        for  $j \leftarrow 2$  to  $i$  do
```

```
            if  $a_j > a_{\text{imaks}}$  then
```

```
                 $\text{imaks} \leftarrow j$ 
```

```
            endif
```

```
        endfor
```

```
        { pertukarkan  $a_{\text{imaks}}$  dengan  $a_i$  }
```

```
         $\text{temp} \leftarrow a_i$ 
```

```
         $a_i \leftarrow a_{\text{imaks}}$ 
```

```
         $a_{\text{imaks}} \leftarrow \text{temp}$ 
```

```
    endfor
```

(i) Jumlah operasi perbandingan elemen

Untuk setiap *pass* ke-*i*,

$i = n \rightarrow \text{jumlah perbandingan} = n - 1$

$i = n - 1 \rightarrow \text{jumlah perbandingan} = n - 2$

$i = n - 2 \rightarrow \text{jumlah perbandingan} = n - 3$

\vdots

$i = 2 \rightarrow \text{jumlah perbandingan} = 1$

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah

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

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 seleksi membutuhkan $n(n - 1)/2$ buah operasi perbandingan elemen dan $n - 1$ buah operasi pertukaran.

Latihan

Contoh 6. Hitung kompleksitas waktu algoritma berikut berdasarkan jumlah operasi kali.

```
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
```

Jawaban

Untuk

$j = n$, jumlah operasi perkalian = n

$j = n/2$, jumlah operasi perkalian = $n/2$

$j = n/4$, jumlah operasi perkalian = $n/4$

...

$j = 1$, jumlah operasi perkalian = 1

Jumlah operasi perkalian seluruhnya adalah

= $n + n/2 + n/4 + \dots + 2 + 1 \rightarrow$ deret geometri

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

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

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

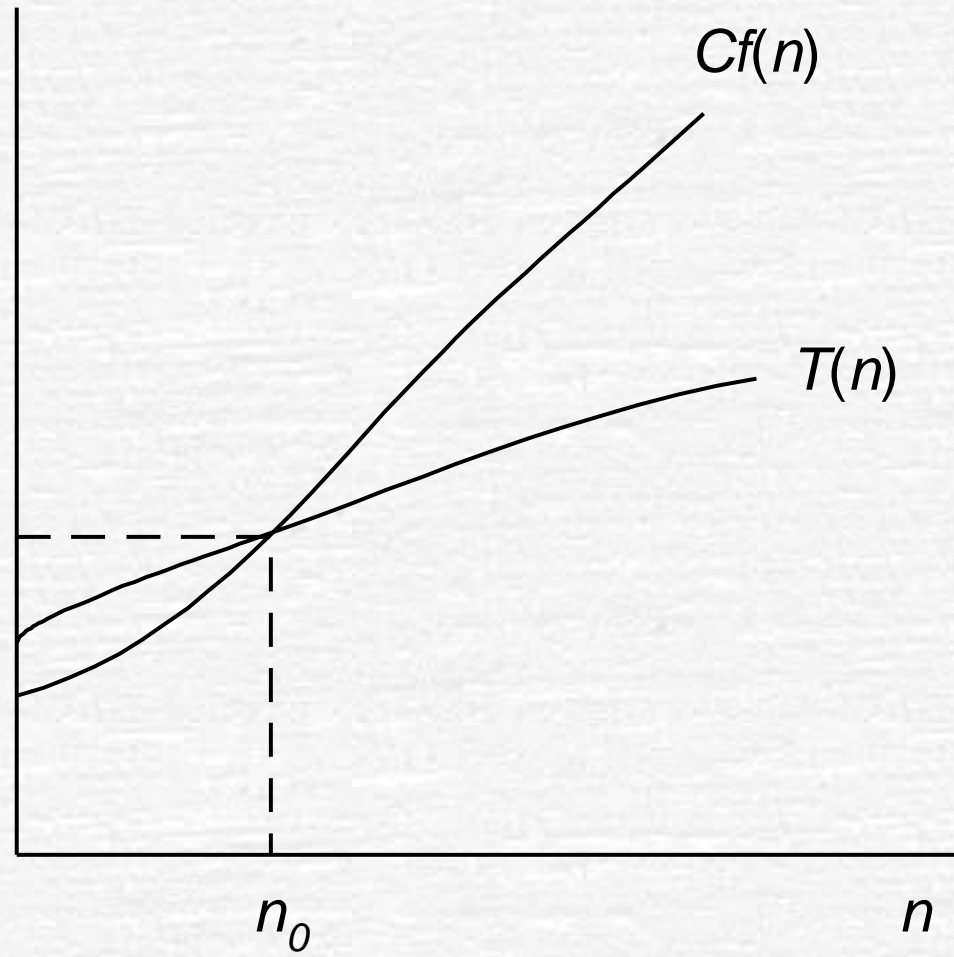
Notasi “ O ” disebut notasi “ O -Besar” (*Big-O*) yang merupakan notasi **kompleksitas waktu asimptotik**.

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

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

untuk $n \geq n_0$.

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



Contoh 7. Tunjukkan bahwa $T(n) = 2n^2 + 6n + 1 = O(n^2)$.

Penyelesaian:

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

karena

$$2n^2 + 6n + 1 \leq 2n^2 + 6n^2 + n^2 = 9n^2 \text{ untuk semua } n \geq 1 \text{ (} C=9 \text{ dan } n_0 = 1 \text{)}.$$

atau karena

$$2n^2 + 6n + 1 \leq n^2 + n^2 + n^2 = 3n^2 \text{ untuk semua } n \geq 6 \text{ (} C=3 \text{ dan } n_0 = 6 \text{)}.$$

Contoh 8. Tunjukkan bahwa $T(n) = 3n + 2 = O(n)$.

Penyelesaian:

$$3n + 2 = O(n)$$

karena

$$3n + 2 \leq 3n + 2n = 5n \text{ untuk semua } n \geq 1 \text{ (} C = 5 \text{ dan } n_0 = 1 \text{)}.$$

Contoh-contoh Lain

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

☛ Penyelesaian:

☛ $5 = O(1)$ karena $5 \leq 6 \cdot 1$ untuk $n \geq 1$.

($C = 6$ dan $n_0 = 1$)

☛ Kita juga dapat memperlihatkan bahwa

$5 = O(1)$ karena $5 \leq 10 \cdot 1$ untuk $n \geq 1$

2. Tunjukkan bahwa kompleksitas waktu algoritma pengurutan seleksi (*selection sort*) adalah $T(n) = n(n - 1)/2 = O(n^2)$.

☞ Penyelesaian:

☞ $n(n - 1)/2 = O(n^2)$ karena

$$n(n - 1)/2 \leq n^2/2 + n^2/2 = n^2$$

untuk semua $n \geq 1$ ($C = 1$ dan $n_0 = 1$).

3. Tunjukkan $T(n) = 6 \cdot 2^n + 2n^2 = O(2^n)$

☞ Penyelesaian:

☞ $6 \cdot 2^n + 2n^2 = O(2^n)$ karena

$$6 \cdot 2^n + 2n^2 \leq 6 \cdot 2^n + 2 \cdot 2^n = 8 \cdot 2^n$$

untuk semua $n \geq 1$ ($C = 8$ dan $n_0 = 1$).

4. Tunjukkan $T(n) = 1 + 2 + \dots + n = O(n^2)$

Penyelesaian:

$$1 + 2 + \dots + n \leq n + n + \dots + n = n^2 \text{ untuk } n \geq 1$$

5. Tunjukkan $T(n) = n! = O(n^n)$

Penyelesaian:

$$n! = 1 \cdot 2 \cdot \dots \cdot n \leq n \cdot n \cdot \dots \cdot n = n^n \text{ untuk } n \geq 1$$

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

☛ Jadi, cukup melihat suku (*term*) yang mempunyai pangkat terbesar.

☛ Contoh:

$$T(n) = 5 = 5n^0 = O(n^0) = O(1)$$

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

$$T(n) = 3n^3 + 2n^2 + 10 = O(n^3)$$

Teorema tersebut digeneralisasi untuk suku dominan lainnya:

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

Contoh: $T(n) = 2^n + 2n^2 = O(2^n)$.

$$T(n) = 2n \log(n) + 3n = O(n \log(n))$$

$$T(n) = \log(n^3) = 3 \log(n) = O(\log(n))$$

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

Perhatikan....(1)

- Tunjukkan bahwa $T(n) = 5n^2 = O(n^3)$, tetapi $T(n) = n^3 \neq O(n^2)$.
- Penyelesaian:
- $5n^2 = O(n^3)$ karena $5n^2 \leq n^3$ untuk semua $n \geq 5$.
- Tetapi, $T(n) = n^3 \neq O(n^2)$ karena tidak ada konstanta C dan n_0 sedemikian sehingga $n^3 \leq Cn^2 \Leftrightarrow n \leq C$ untuk semua n_0 karena n dapat berupa sembarang bilangan yang besar.

Perhatikan ...(2)

- Defenisi: $T(n) = O(f(n))$ jika terdapat C dan n_0 sedemikian sehingga $T(n) \leq C.f(n)$ untuk $n \geq n_0$
→ tidak menyiratkan seberapa atas fungsi f itu.
- Jadi, menyatakan bahwa
 - $T(n) = 2n^2 = O(n^2) \rightarrow$ benar
 - $T(n) = 2n^2 = O(n^3) \rightarrow$ juga benar
 - $T(n) = 2n^2 = O(n^4) \rightarrow$ juga benar
- Namun, untuk alasan praktis kita memilih fungsi yang sekecil mungkin agar $O(f(n))$ memiliki makna
- Jadi, kita menulis $2n^2 = O(n^2)$, bukan $O(n^3)$ atau $O(n^4)$

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

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

$$(b) \quad T_1(n)T_2(n) = O(f(n))O(g(n)) = O(f(n)g(n))$$

$$(c) \quad O(cf(n)) = O(f(n)), c \text{ adalah konstanta}$$

$$(d) \quad f(n) = O(f(n))$$

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

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

$$(b) \quad T_1(n)T_2(n) = O(n \cdot n^2) = O(n^3)$$

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

Pengelompokan Algoritma Berdasarkan Notasi O -Besar

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

Urutan spektrum kompleksitas waktu algoritma adalah :

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

algoritma polinomial

algoritma eksponensial

Penjelasan masing-masing kelompok algoritma adalah sebagai berikut:

$O(1)$ 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 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.

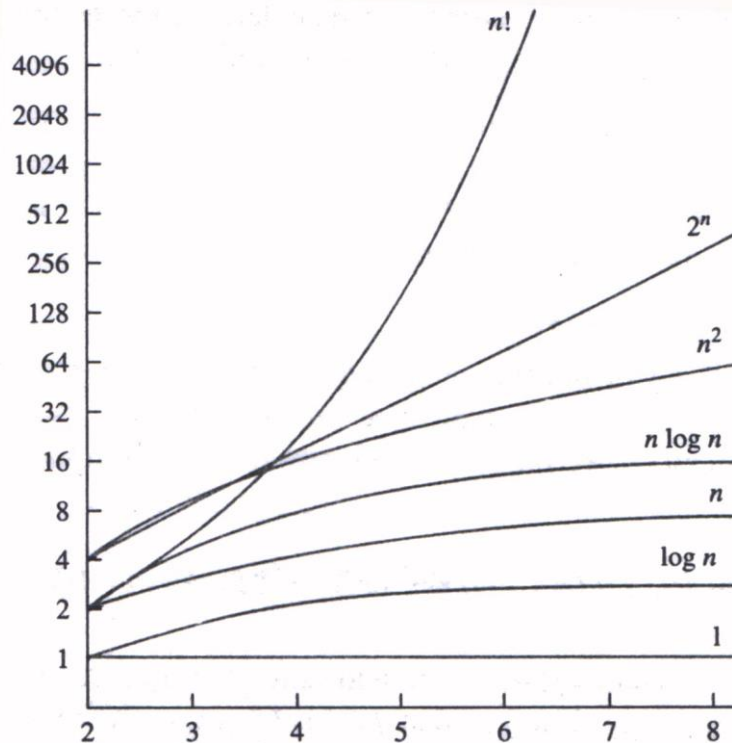
$O(n^3)$ Seperti halnya algoritma kuadratik, algoritma kubik memproses setiap masukan dalam tiga buah kalang bersarang, misalnya algoritma perkalian matriks. Bila $n = 100$, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, waktu pelaksanaan algoritma meningkat 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 9). 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 $2n$.

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	8	24	64	512	256	362880
4	16	64	256	4096	65536	20922789888000
5	32	160	1024	32768	4294967296	(terlalu besar)



Kegunaan Notasi *Big-Oh*

- Notasi *Big-Oh* berguna untuk membandingkan beberapa algoritma dari untuk masalah yang sama
→ menentukan yang terbaik.
- Contoh: masalah pengurutan memiliki banyak algoritma penyelesaian,
Selection sort, insertion sort → $T(n) = O(n^2)$
Quicksort → $T(n) = O(n \log n)$

Karena $n \log n < n^2$ untuk n yang besar, maka algoritma *quicksort* lebih cepat (lebih baik, lebih mangkus) daripada algoritma *selection sort* dan *insertion sort*.

Notasi Omega-Besar dan Tetha-Besar

Definisi Ω -Besar adalah:

$T(n) = \Omega(g(n))$ (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 $0 \leq 6n^2 \log n \leq 6n^3$, 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 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: Tentukan notasi notasi O , Ω dan Θ untuk $T(n) = 1 + 2 + \dots + n$.

Jawab:

$1 + 2 + \dots + n = O(n^2)$ karena

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

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

$$1 + 2 + \dots + n \leq 1 + 1 + \dots + 1 = n \text{ untuk } n \geq 1.$$

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

Kita menyimpulkan bahwa

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

Oleh karena itu,

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

Latihan

Tentukan kompleksitas waktu dari algoritma dibawah ini jika melihat banyaknya operasi $a \leftarrow a + 1$

```
for i  $\leftarrow$  1 to n do  
  for j  $\leftarrow$  1 to i do  
    for k  $\leftarrow$  j to n do  
      a  $\leftarrow$  a + 1  
    endfor  
  endfor  
endfor
```

Tentukan pula nilai O -besar, Ω -besar, dan Θ -besar dari algoritma diatas (harus penjelasan)

TEOREMA. 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 .

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  $A \neq B$  }
```

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 .

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


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 $T(n) \leq C \cdot f(n)$ untuk semua $n \geq n_0$:

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

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