Aturan Untuk Menentukan Kompleksitas Waktu Asimptotik

1. Jika kompleksitas waktu $T(n)$ dari algoritma diketahui,
   Contoh: (i) pada algoritma `cari_maksimum`
   $T(n) = n - 1 = O(n)$

   (ii) pada algoritma `pencarian_beruntun`
   $T_{\min}(n) = 1 = O(1)$
   $T_{\max}(n) = n = O(n)$
   $T_{\text{avg}}(n) = (n + 1)/2 = O(n)$,

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

   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

   $\log(n^2 + 1) \le \log(2n^2) = \log 2 + \log n^2$
   $= \log 2 + 2 \log n \le 3 \log n$ 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

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

Contoh:

```
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.

Contoh

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

Kompleksitas waktu asimptotik $= n \,.\, O(1)$
$= O(n \,.1)$
$= O(n)$

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.

Contoh: 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
$= O(1) + (n\text{-}1) \{ O(1) + O(1) + O(1) \}$
$= O(1) + (n\text{-}1) \; O(1)$
$= O(1) + O(n\text{-}1)$
$= O(1) + O(n)$
$= O(n)$

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.

(f) Prosedur dan fungsi. Waktu yang dibutuhkan untuk memindahkan kendali ke rutin yang dipanggil adalah $O(1)$.

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 :

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

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.

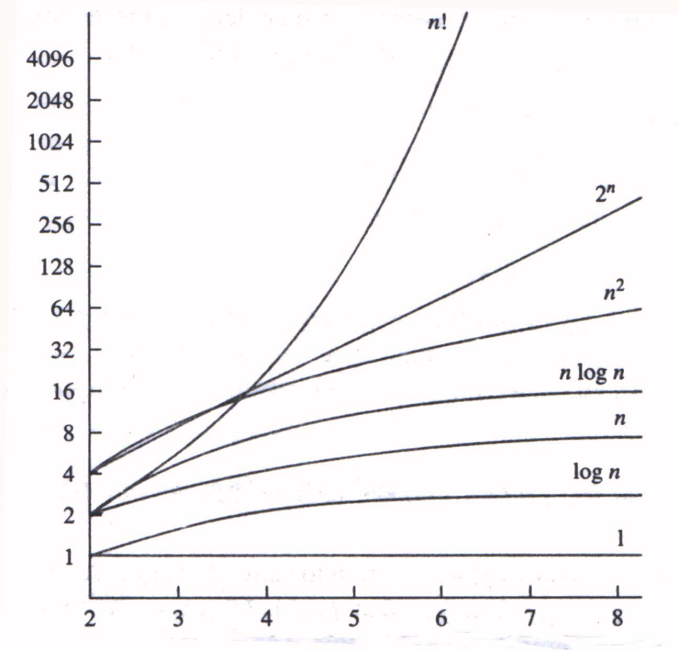
$O(n^3)$ Seperti halnya algoritma kuadratik, algoritma kubik memproses setiap masukan dalam tiga buah kalang bersarang, misalnya algoritma perkalian matriks. Bila n = 100, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, waktu pelaksanan algoritma meningkat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

$O(2^n)$ Algoritma yang tergolong kelompok ini mencari solusi persoalan secara "*brute force*", misalnya pada algoritma mencari sirkuit Hamilton (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	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 [ROS99]. 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 [JOH90]?

- 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 terbuktikan, bahwa tidak ada masalah di dalam kelas NP-lengkap yang dapat dipecahkan dalam waktu polinomial [ROS99].

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 $2n^2 + 6n + 1 = O(n^2)$ 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 $T(n) = 1 + 2 + \ldots + n$.
Jawab:

$1 + 2 + \ldots + n = O(n^2)$ karena
$1 + 2 + \ldots + n \leq n + n + \ldots + n = n^2$ untuk $n \geq 1$.

$1 + 2 + \ldots + n = \Omega(n)$ karena
$1 + 2 + \ldots + n \leq 1 + 1 + \ldots + 1 = n$ untuk $n \geq 1$.

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

Kita menyimpulkan bahwa

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

Oleh karena itu,

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

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.