NOTASI ASIMTOTIK (1)

ANALISIS ALGORITMA DAN KOMPLEKSITAS

ORDER OF GROWTH

- **Merupakan abstraksi untuk mempermudah analisis dan berfokus pada fitur-fitur yang penting saja.**
- **Memberikan gambaran sederhana tentang efisiensi dari algoritma.**
- **Hanya mempertimbangkan term dengan order terbesar dalam persamaan untuk menghitung running time.**
  - Term dengan order lebih rendah dihilangkan.
  - Abaikan konstanta koefisien yang ada pada term yang dipakai.
- **Contoh:**
  - Worst-case running time dari insertion adalah $an^2 + bn + c$.
  - Hilangkan term dengan order lebih rendah ⇒ an^2.
  - Abaikan konstanta koefisien ⇒ n^2.
  - Maka, running time-nya adalah $\Theta(n^{2,})$ yang berarti order of growth-nya n^2.

- **Pada saat ukuran input *n* menjadi cukup besar, yang perlu diperhatikan adalah efisiensi algoritma secara asimtotik.**
  - Memungkinkan kita membandingkan kinerja secara relatif dari beberapa algoritma.
  - Contoh: pada saat ukuran input menjadi besar, merge sort (dengan kompleksitas $\Theta(n \lg n)$ pada worst case) akan lebih cepat daripada insertion sort (dengan worst-case running time $\Theta(n^2)$).
  - Titik berat: bagaimana running time dari algoritma meningkat seiring dengan peningkatan ukuran input tanpa ada batasnya.
- **Algoritma yang secara asimtotik lebih efisien biasanya merupakan pilihan terbaik, kecuali untuk ukuran input yang kecil.**

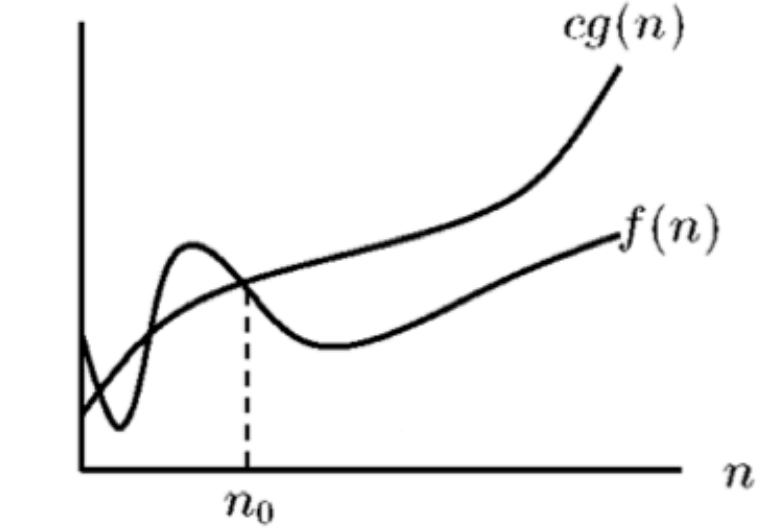
NOTASI ASIMTOTIK (ASYMPTOTIC NOTATION)

- **Notasi asimtotik: digunakan untuk mendeskripsikan running time dari algoritma.**
- **Notasi asimtotik diterapkan pada fungsi.**
- **Dapat diaplikasikan juga untuk fungsi-fungsi yang mendeskripsikan aspek lain algoritma, misal: space yang dipakai.**
- **Macam-macam:**
  - Big oh
  - Big omega
  - Theta
  - Little oh
  - Little omega

NOTASI BIG OH (O)

- **Disebut juga notasi 'the order of'**
- **Pengucapan: 'big-oh' atau 'oh'**
  - O(g(n)) : big oh dari g(n)
- **Definisi:**
  - Untuk suatu fungsi g(n), O(g(n)) adalah himpunan fungsi

  O(g(n))={f(n): terdapat konstanta positif c dan n_0 sedemikian hingga $0 \leq f(n) \leq cg(n)$ untuk semua $n \geq n_0$ }
- **Penulisan *f*(*n*) = *O*(*g*(*n*)) mempunyai arti bahwa *f*(*n*) adalah anggota himpunan *O*(*g*(*n*)).**



- ***f*(*n*) = *O*(*g*(*n*)) menunjukkan bahwa g(n) adalah batas atas asimtotik pada f(n), tetapi tidak menunjukkan seberapa ketat batas tersebut.**
- **Secara praktis, notasi O digunakan untuk mendeskripsikan worst-case running time dari algoritma.**
- **"Algoritma memiliki *O*(*g*(*n*))" berarti bahwa:**
  - Running time tertinggi adalah *g*(*n*), untuk n cukup besar, tanpa memandang bagaimana input tertentu dengan ukuran n dipilih.

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

Penyelesaian:

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

karena

$2n^2 + 6n + 1 \leq 2n^2 + 6n^2 + n^2 = 9n^2$ untuk semua $n \geq 1$ (c =9 dan $n_0 = 1$).

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

Penyelesaian:

$3n + 2 = O(n)$

karena

$3n + 2 \leq 3n + 2n = 5n$ untuk semua $n \geq 1$ ($c = 5$ dan $n_0 = 1$).

CONTOH-CONTOH LAIN

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

Penyelesaian:

- **$5 = O(1)$ karena $5 \leq 6.1$ untuk $n \geq 1$.**

  ($c = 6$ dan $n_0 = 1$)
- **Kita juga dapat memilih c lain, misalnya 10:**

  $5 = O(1)$ 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 $n_0 = 1$).

- **Teorema: Bila $T(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ 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 $T(n) = n^3 \neq O(n^2)$.**

- **<u>Penyelesaian</u>:**
  - $5n^2 = O(n^3)$ 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) ≤ c.f(n) untuk n ≥ n_0**

  → tidak menyiratkan seberapa atas fungsi f itu.

- **Jadi, menyatakan bahwa**

  $T(n) = 2n^2 = O(n^2)$ → benar

  $T(n) = 2n^2 = O(n^3)$ → juga benar

  $T(n) = 2n^2 = O(n^4)$ → 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 $T_1(\text{n}) = O(f(n))$ 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. 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. $O(5n^2) = O(n^2)$

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

Pengelompokan Algoritma Berdasarkan Notasi *O*-Besar

Kelompok Algoritma	Nama
$O(1)$	konstan
$O(\log n)$	logaritmik
$O(n)$	linier
$O(n \log n)$	linieritmik
$O(n^2)$	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:

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`). Bila n dinaikkan dua kali semula, misalnya, running time algoritma meningkat menjadi log $2n$.

$O(n)$ Algoritma yang waktu pelaksanaannya linier umumnya terdapat pada kasus yang setiap elemen masukannya dikenai proses yang sama, misalnya algoritma pencarian secara sekuensial. 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 devide and conquer mempunyai kompleksitas asimtotik jenis ini. Bila $n = 1000$, maka $n \log n$ mungkin 20.000. Bila n dijadikan dua kali semula, maka $n \log n$ menjadi sekitar dua kali semula.

$O(n^2)$ 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 penjumlahan matriks dengan ukuran nxn. 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.

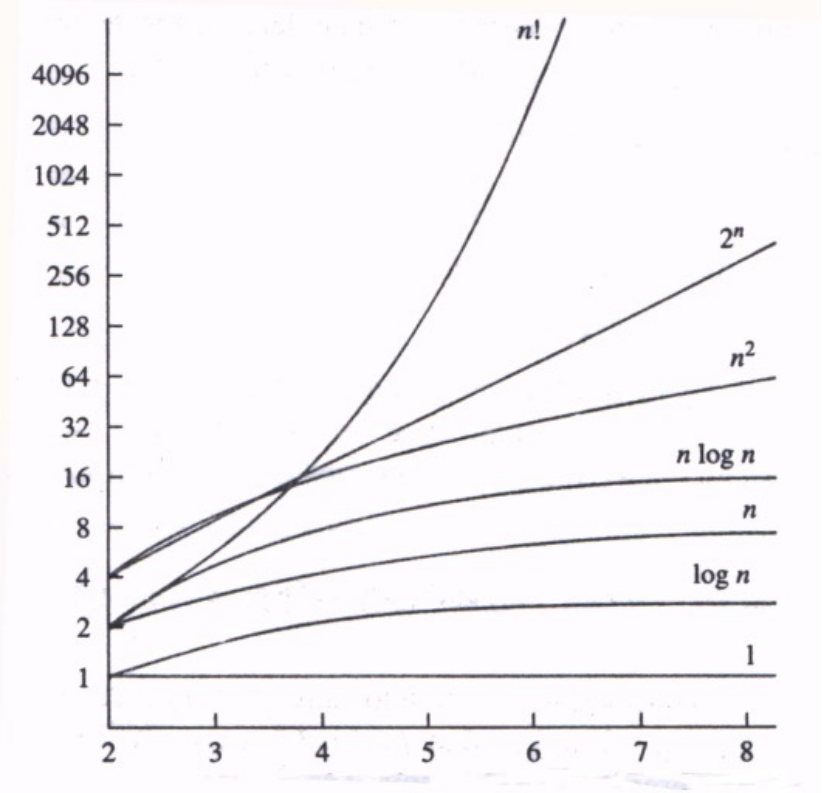
$O(n^3)$ Algoritma kubik memproses setiap masukan dalam tiga buah kalang bersarang, misalnya algoritma perkalian matriks. Bila n = 100, maka waktu pelaksanaan algoritma adalah 1.000.000. Bila n dinaikkan menjadi dua kali semula, waktu pelaksanan algoritma meningkat menjadi delapan kali semula.

$O(2^n)$ 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)



PERBANDINGAN PERTUMBUHAN KOMPLEKSITAS WAKTU

- **Algoritma linier (O(n)), yang awalnya dapat memproses 10 item dalam 1 menit, dapat memproses:**
  - 1.4 x 10^4 item dalam 1 hari
  - 5.3 x 10^6 item dalam 1 tahun
  - 5.3 x 10^8 item dalam 1 abad
- **Algoritma eksponensial ($O(2^n)$), yang awalnya dapat memproses 10 item dalam 1 menit, dapat memproses :**
  - 20 item dalam 1 hari
  - 29 items dalam 1 tahun
  - 35 items dalam 1 abad

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 efisien) daripada algoritma *selection sort* dan *insertion sort*.