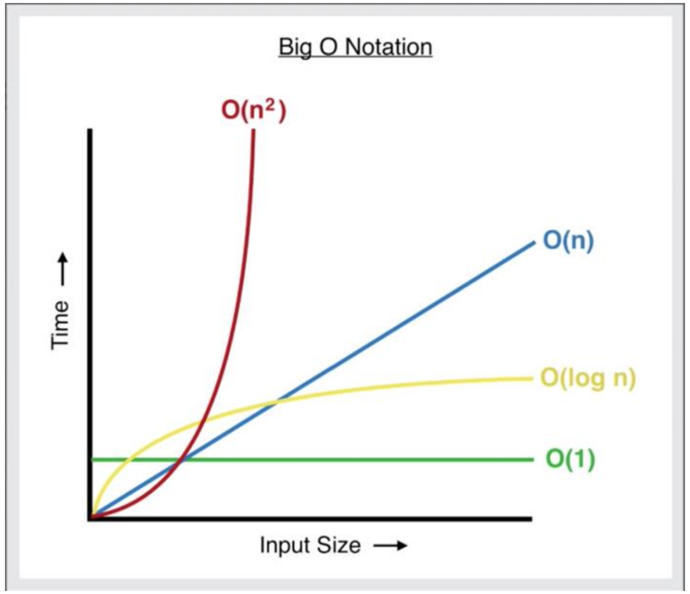


Kompleksitas Algoritma (Bagian 2)

Bahan Kuliah

IF2120 Matematika Diskrit

Oleh: Rinaldi Munir

Program Studi Teknik Informatika

STEI - ITB

Kompleksitas Waktu Asimptotik

- Seringkali kita kurang tertarik dengan kompleksitas waktu $T(n)$ yang presisi untuk suatu algoritma.

- Kita lebih tertarik pada bagaimana kebutuhan waktu sebuah algoritma tumbuh ketika ukuran masukannya (n) meningkat.

- Contoh, sebuah algoritma memiliki jumlah operasi perkalian sebesar

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

Kita mungkin tidak terlalu membutuhkan informasi seberapa presisi jumlah operasi perkalian di dalam algoritma tersebut.

Yang kita butuhkan adalah seberapa cepat fungsi $T(n)$ tumbuh ketika ukuran data masukan membesar.

- Kinerja algoritma baru akan tampak untuk *n* yang sangat besar, bukan pada *n* yang kecil.

- Kinerja algoritma-algoritma pengurutan seperti *selection sort* dan *bubble sort* misalnya, baru terlihat ketika mengurutkan larik berukuran besar, misalnya 10000 elemen.

- Oleh karena itu, kita memerlukan suatu notasi kompleksitas algoritma yang memperlihatkan kinerja algoritma untuk *n* yang besar.

- Notasi kompleksitas waktu algoritma untuk *n* yang besar dinamakan **kompleksitas waktu asimptotik**.

- Langkah pertama dalam mengukur kinerja algoritma adalah membuat makna “sebanding”. Gagasannya adalah dengan menghilangkan faktor koefisien di dalam ekspresi $T(n)$.

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

n	$T(n) = 2n^2 + 6n + 1$	n^2
10	261	100
100	20.601	10.000
1000	2.006.001	1.000.000
10.000	200.060.001	100.000.000

- Dari table di atas, untuk n yang besar pertumbuhan $T(n)$ sebanding dengan n^2.

- $T(n)$ tumbuh seperti n^2 tumbuh saat n bertambah. Kita katakan bahwa $T(n)$ <u>sebanding</u> dengan n^2 dan kita tuliskan

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

Notasi O-Besar (Big-O)

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

- **DEFINISI 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

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

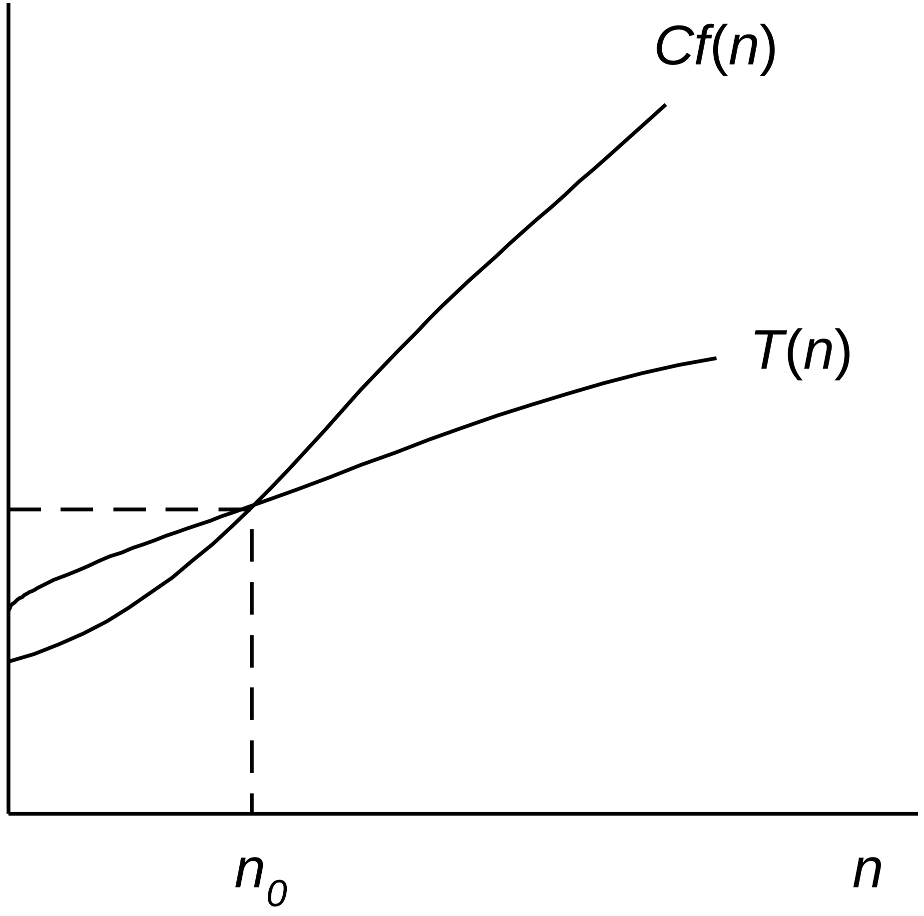
untuk $n \geq n_0$.

- $f(n)$ adalah <u>batas lebih atas</u> (*upper bound*) dari $T(n)$ untuk n yang besar.

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 $n \geq n_0$.



Fungsi $f(n)$ umumnya dipilih dari fungsi-fungsi standard seperti 1, n^2, n^3, ..., log n, n log n, 2^n, $n!$, dan sebagainya.

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 $n \geq n_0$.

- **Catatan**: Ada tak-berhingga nilai C dan n_0 yang memenuhi $T(n) \leq C f(n)$, kita cukup menunjukkan satu pasang (C, n_0) yang memenuhi definisi sehingga $T(n) = O(f(n))$

Contoh 7. Tunjukkan bahwa $2n^2 + 6n + 1 = O(n^2)$. (tanda '=' dibaca 'adalah')

Penyelesaian:

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

$2n^2 + 6n + 1 \leq 2n^2 + 6n^2 + n^2 = 9n^2$ untuk semua $n \geq 1$ $(C = 9,\ f(n) = n^2,\ n_0 = 1)$.

atau karena

$2n^2 + 6n + 1 \leq n^2 + n^2 + n^2 = 3n^2$ untuk semua $n \geq 7$ $(C = 3, f(n) = n^2, n_0 = 7)$.

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 $n \geq n_0$.

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

<u>Penyelesaian</u>:

$3n + 2 = O(n)$

karena

$3n + 2 \leq 3n + 2n = 5n$ untuk semua $n \geq 1$

($C = 5$, $f(n) = n$, dan $n_0 = 1$).

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 $n \geq n_0$.

Contoh-contoh Lain

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

Jawaban:

$5 = O(1)$ karena $5 \leq 6 \cdot 1$ untuk $n \geq 1$ ($C = 6$, $f(n) = 1$, dan $n_0 = 1$)

Kita juga dapat memperlihatkan bahwa

$5 = O(1)$ karena $5 \leq 10 \cdot 1$ untuk $n \geq 1$ ($C = 10$, $f(n) = 1$, dan $n_0 = 1$)

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

<u>Jawaban</u>:

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

karena

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

untuk $n \geq 1$

$(C = 1, f(n) = n^2,$ dan $n_0 = 1)$.

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

Jawaban:

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

karena

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

untuk semua $n \geq 4$ ($C = 8$, $f(n) = 2^n$, dan $n_0 = 4$).

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

Jawaban:

Cara 1: $1 + 2 + \ldots + n \leq n + n + \ldots + n = n^2$ untuk $n \geq 1$

Cara 2: $1 + 2 + \ldots + n = \frac{1}{2}n(n + 1) \leq \frac{1}{2}n^2 + \frac{1}{2}n^2 = n^2$ untuk $n \geq 1$

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

Jawaban:

$n! = 1 \cdot 2 \cdot \ldots \cdot n \leq n \cdot n \cdot \ldots \cdot n = n^n$ untuk $n \geq 1$

6. Tunjukkan $\log n! = O(n \log n)$

<u>Jawaban</u>:

Dari soal 5 sudah diperoleh bahwa $n! \le n^n$ untuk $n \ge 1$ maka

$\log n! \le \log n^n = n \log n$ untuk $n \ge 1$maka

sehingga $\log n! = O(n \log n)$

7. Tunjukkan $8n^2 = O(n^3)$

<u>Jawaban</u>:

$8n^2 = O(n^3)$ karena $8n^2 \le n^3$ untuk $n \ge 8$

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

- Jadi, untuk menentukan notasi *Big-Oh*, cukup melihat suku (*term*) yang mempunyai pangkat terbesar di dalam $T(n)$.

- **Contoh 8**:

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

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

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

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

- Teorema 1 tersebut digeneralisasi untuk suku-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 9: $T(n) = 2^n + 2n^2 = O(2^n)$.

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

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

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

Perhatikan….(1)

Tunjukkan bahwa $T(n) = 5n^2 = O(n^3)$, tetapi $T(n) = n^3 \neq O(n^2)$.

<u>Jawaban</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)

- Definisi: $T(n) = O(f(n))$ 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

  $T(n) = 2n^2 = O(n^2)$ → benar

  $T(n) = 2n^2 = O(n^3)$ → juga benar, karena $2n^2 \leq 2n^3$ untuk $n \geq 1$

  $T(n) = 2n^2 = O(n^4)$ → juga benar, karena $2n^2 \leq 2n^4$ untuk $n \geq 1$

- Namun, untuk alasan praktikal 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 $O(n^4)$

Perhatikan …(3)

- Menuliskan

  $O(2n)$ tidak standard, seharusnya $O(n)$

  $O(n - 1)$ tidak standard, seharusnya $O(n)$

  $O(\frac{n^2}{2})$ tidak standard, seharusnya $O(n^2)$

  $O((n - 1)!)$ tidak standard, seharusnya $O(n!)$

- Ingat, di dalam notasi Big-Oh tidak ada koefisien atau suku-suku lainnya, hanya berisi fungsi-fungsi standard seperti 1, n^2, n^3, …, log n, n log n, 2^n, $n!$, dan sebagainya

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

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

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

Contoh 11: Tentukan notasi O-besar untuk $T(n) = (n + 1)\log(n^2 + 1) + 3n^2$.

Jawaban:

Cara 1: $\bullet\ n + 1 = O(n)$

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

$= \log(2) + 2\log(n)$

$\le \log(n) + 2\log(n) = 3\log(n)$ untuk $n \ge 2$

$= O(\log n)$

$\bullet\ (n + 1)\log(n^2 + 1) = O(n)\,O(\log n) = O(n \log n)$

$\bullet\ 3n^2 = O(n^2)$

$\bullet\ (n + 1)\log(n^2 + 1) + 3n^2 = O(n \log n) + O(n^2) = O(\max(n \log n,\ n^2) = O(n^2)$

Cara 2: suku yang dominan di dalam $(n + 1)\log(n^2 + 1) + 3n^2$ untuk n yang besar adalah $3n^2$, sehingga $(n + 1)\log(n^2 + 1) + 3n^2 = O(n^2)$

Pengelompokan Algoritma Berdasarkan Notasi *O*-Besar

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

Urutan spektrum kompleksitas waktu algoritma adalah :

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

algoritma polinomial

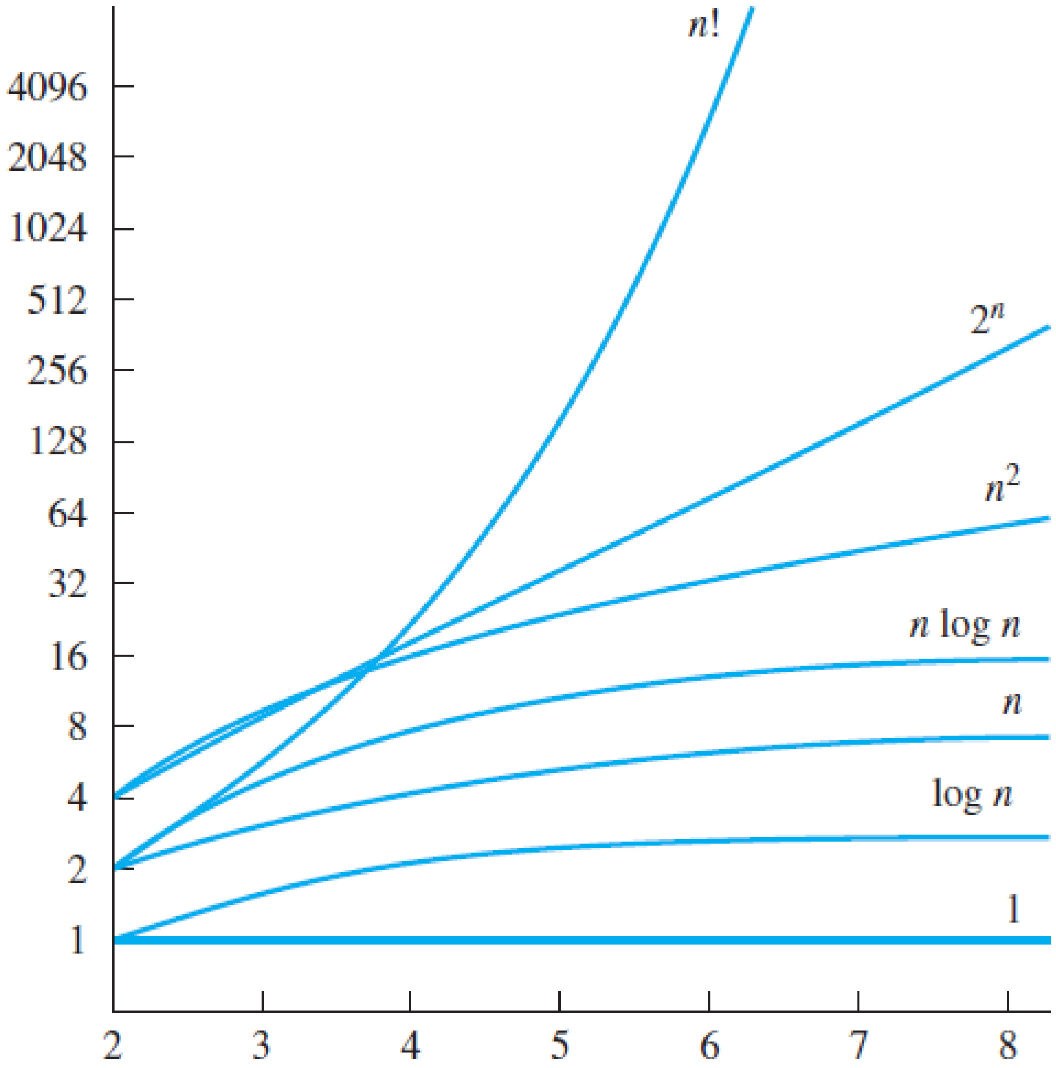
(bagus)

algoritma eksponensial

(buruk)

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 untuk ditulis)



Sumber: Kenneth H. Rosen

$O(1)$

- Kompleksitas $O(1)$ berarti waktu pelaksanaan algoritma adalah tetap, tidak bergantung pada ukuran masukan.

- Algoritma yang memiliki kompleksitas $O(1)$ terdapat pada algoritma yang instruksinya dijalankan satu kali (tidak ada pengulangan)

  Contoh: **if** $a > b$ **then** $maks \leftarrow a$ **else** $maks \leftarrow b$ $T(n) = O(1)$

- Contoh lainnya, operasi pertukaran a dan b sebagai berikut:

  temp $\leftarrow$ a
  a $\leftarrow$ b
  b $\leftarrow$ temp

  Di sini jumlah operasi pengisian nilai 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.
- Contoh algoritma: algoritma *binary search*

- Di sini basis logaritma tidak terlalu penting sebab bila *n* dinaikkan dua kali semula, misalnya, log *n* meningkat sebesar sejumlah tetapan.

***O*(*n*)**

- Algoritma yang waktu pelaksanaannya lanjar (linier) umumnya terdapat pada kasus yang setiap elemen masukannya dikenai proses yang sama.

- Contoh algoritma: algoritma *sequential search*, algoritma mencari nilai maksimum, menghitung rata-rata, dan sebagainya.

$O(n \log n)$

- 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 (*divide and conquer*).

- Algoritma yang diselesaikan dengan *divide and conquer* mempunyai kompleksitas asimptotik jenis ini.

- Bila n = 1000, maka $n \log n$ sekitar 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.

- Contoh algoritma: algoritma pengurutan *selection sort*, *insertion sort*, *bubble sort*, penjumlahan dua buah matriks, dsb.

***O*(n^3)**

- Seperti halnya algoritma kuadratik, algoritma kubik memproses setiap masukan dalam tiga buah kalang bersarang.
- Contoh: algoritma perkalian matriks.
- Bila n = 100, maka waktu komputasi algoritma adalah 1.000.000 operasi. 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*“.
- Contoh: algoritma mencari sirkuit Hamilton, algoritma *knapsack*, algoritma *sum of subset*, dsb.
- Laju peningkatan fungsi bersifat ekponensial, artinya jika n bertambah sedikit, maka nilai fungsi bertambah sangat signifikan.
- Contoh: n = 15, nilai 2^n = 65.536,
  n = 18, nilai 2^n = 262.144

$O(n!)$

- Algoritma jenis ini memproses setiap masukan dan menghubungkannya dengan $n - 1$ masukan lainnya.

- Contoh: Algoritma Persoalan Pedagang Keliling (*Travelling Salesperson Problem*).

- Seperti halnya pada algoritma eksponensial, laju pertumbuhan fungsi kebutuhan waktu algoritma jenis ini meningkat signifikan dengan bertambahnya nilai n.

- Bila $n = 5$, maka waktu komputasi algoritma adalah 120. Bila $n = 20$, maka waktu komputasinya 2,432,902,008,176,640,000.

Kegunaan Notasi *Big-Oh*

- Notasi *Big-Oh* berguna untuk membandingkan beberapa algoritma untuk persoalan yang sama

  → menentukan yang terbaik.

- Contoh: persoalan pengurutan memiliki banyak algoritma penyelesaian,

  Selection sort, *bubble 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 Big-Omega dan Big-Tetha

- Definisi Ω-Besar adalah:

Definisi 2. $T(n) = \Omega(g(n))$ (dibaca "$T(n)$ adalah Omega ($g(n)$)" yang artinya *T(n) berorde paling kecil g(n)*) bila terdapat konstanta C dan n_0 sedemikian sehingga $T(n) \geq C(g(n))$ untuk $n \geq n_0$.

- Definisi Θ-Besar adalah:

Definisi 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 $T(n) = \Omega(h(n))$.

- Jika $T(n) = \Theta(h(n))$ maka kita katakan $T(n)$ berorde $h(n)$

Contoh 12: Tentukan notasi Ω dan Θ untuk $T(n) = 2n^2 + 6n + 1$.

Jawaban:

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

$2n^2 + 6n + 1 \geq 2n^2$ untuk $n \geq 1$ $\quad$ $(C = 2, n_0 = 1)$

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

Jawaban:

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 14: Tentukan 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 $n \geq 1$.

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

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

Atau, dengan cara kedua:

$$
\begin{aligned}
1 + 2 + \ldots + n = \Omega(n^2) \text{ karena } 1 + 2 + \ldots + n &= \frac{1}{2}n(n + 1) \\
&= \frac{1}{2}n^2 + \frac{1}{2}n \geq \frac{1}{2}n^2 \text{ untuk } n \geq 1.
\end{aligned}
$$

Oleh karena $1 + 2 + \ldots + n = O(n^2)$ dan $1 + 2 + \ldots + n = \Omega(n^2)$, maka $1 + 2 + \ldots + n = \Theta(n^2)$

TEOREMA 3. Bila $T(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ adalah polinom derajat $\leq m$ maka $T(n)$ adalah berorde n^m.

- Teorema 3 menyatakan bahwa jika $T(n)$ berbentuk polinom derajat $\leq$ m, maka $T(n) = \Theta(n^m)$, yang berarti juga bahwa $T(n) = O(n^m)$ dan $T(n) = \Omega(n^m)$.

Contoh: $3n^3 + 2n^2 + n + 1 = \Theta(n^3)$

- Penulis dapat menggunakan salah satu notasi Big-O, Big-Ω, Big-Θ dalam menyatakan kompleksitas asimptotik algoritma. Jika menggunakan Big-Θ berarti penulis menyatakan bahwa *lower bound* dan *upper bound* fungsi kebutuhan waktu algoritma adalah sama.

Latihan

Tentukan kompleksitas waktu dari algoritma dibawah ini dihitung dari banyaknya operasi penjumlahan a←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 diberi penjelasan)

Jawaban

Untuk $i = 1$,

  Untuk $j = 1$, jumlah perhitungan = n kali

Untuk $i = 2$,

  Untuk $j = 1$, jumlah perhitungan = n kali

  Untuk $j = 2$, jumlah perhitungan = $n - 1$ kali

...

Untuk $i = n$,

  Untuk $j = 1$, jumlah perhitungan = n kali

  Untuk $j = 2$, jumlah perhitungan = $n - 1$ kali

  ...

  Untuk $j = n$, jumlah perhitungan = 1 kali.

Jadi jumlah perhitungan = $T(n) = n^2 + (n - 1)^2 + (n - 2)^2 + ... + 1$

```
for i ← 1 to n do
  for j ← 1 to i do
    for k ← j to n do
      a ← a + 1
    endfor
  endfor
endfor
```

- $T(n) = O(n^3) = \Omega(n^3) = \Theta(n^3)$.

- Salah satu cara penjelasannya adalah:

$$T(n) = n^2 + (n-1)^2 + (n-2)^2 + ... + 1$$
$$= n(n+1)(2n+1)/6$$
$$= 2n^3 + 3n^2 + 1.$$

- Diperoleh

$2n^3 + 3n^2 + 1 = O(n^3)$ karena $2n^3 + 3n^2 + 1 \leq 6n^3$ untuk $n \geq 1$

dan

$2n^3 + 3n^2 + 1 = \Omega(n^3)$ karena $2n^3 + 3n^2 + 1 \geq 2n^3$ untuk $n \geq 1$.

Menentukan Notasi Big-O suatu Algoritma

Cara 1:

- Tentukan $T(n)$ dari algoritma.
- Notasi Big-O dapat langsung ditentukan dengan mengambil suku yang mendominasi fungsi T dan menghilangkan koefisiennya.

Contoh:

1. Algoritma mencari nilai maksimum: $T(n) = n – 1 = O(n)$
2. Algoritma sequential search:

$$T_{min}(n) = 1 = O(1), \quad T_{max}(n) = n = O(n), \quad T_{avg}(n) = (n + 1)/2 = O(n)$$

3. Algoritma *selection sort*: $T(n) = \dfrac{n(n-1)}{2} = O(n^2)$

Cara 2:

- Setiap operasi yang terdapat di dalam algoritma (baca/tulis, *assignment*, operasi aritmetika, operasi perbandingan, dll) memiliki kompleksitas O(1). Jumlahkan semuanya.
- Jika ada pengulangan, hitung jumlah pengulangan, lalu kalikan dengan total Big-O semua instruksi di dalam pengulangan
- Contoh 1:

```
read(x)                          O(1)
if  x mod 2 = 0 then             O(1)
    x ← x + 1                    O(1)
   write(x)                      O(1)
else
  write(x)                       O(1)
endif
```

Kompleksitas waktu asimptotik algoritma:

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

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

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

$= O(1)$

- **Contoh 2:**

jumlah $\leftarrow$ 0	$O(1)$
$i \leftarrow 2$	$O(1)$
while $i \le n$ **do**	$O(1)$
$jumlah \leftarrow jumlah + a[i]$	$O(1)$
$i \leftarrow i + 1$	$O(1)$
endwhile	
$rata \leftarrow jumlah/n$	$O(1)$

Kalang **while** dieksekusi sebanyak n – 1 kali, sehingga kompleksitas asimptotiknya

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

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

Contoh 3:

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

Kompleksitas untuk $a \leftarrow a + 1$ $= O(1)$

Kompleksitas untuk $b \leftarrow b - 2$ $= O(1)$

Kompleksitas total keduanya $= O(1) + O(1) = O(1)$

Jumlah pengulangan seluruhnya $= 1 + 2 + \ldots + n = n(n + 1)/2$

Kompleksitas seluruhnya $= n(n + 1)/2\ O(1) = O(n(n + 1)/2)$

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

$= O(n^2)$

Latihan Mandiri

1. Untuk soal (a) sampai (e) berikut, tentukan C, $f(n)$, n_0, dan notasi O-besar sedemikian sehingga $T(n) = O(f(n))$ jika $T(n) \leq C f(n)$ untuk semua $n \geq n_0$:
   - (a) $T(n) = 2 + 4 + 6 + \ldots + 2n$
   - (b) $T(n) = (n + 1)(n + 3)/(n + 2)$
   - (c) $T(n) = n \log(n^2 + 1) + n^2 \log n$
   - (d) $T(n) = (n \log n + 1)^2 + (\log n + 1)(n^2 + 1)$
   - (e) $T(n) = n^{2^n} + n^{n^2}$

2. Perhatikan potongan kode C berikut:

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        a = a + j;
    }
}
for (k = 0; k < N; k++) {
    b = b + k;
}
```

(a) Hitung kompleksitas waktu algoritma berdasarkan banyaknya operasi penjumlahan

(b) Nyatakan kompleksitas waktu algoritma dalam notasi Big-O, Big-Ω, dan Big-Θ

3. Diberikan *n* buah titik pada bidang kartesian, yaitu $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. Algoritma berikut mencari sepasang titik yang jaraknya terdekat dengan algoritma *brute force*. Tentukan kompleksitas waktu asimptotik algoritma dalam notasi Big-O.

```
function closestPair((x₁, y₁), (x₂, y₂), ..., (xₙ, yₙ): titik) → pasangan titik

Deklarasi
      min, dist: real
      closest: pasangan titik

Algoritma
      min ← ∞
      for i ← 2 to n do
            for j ← 1 to (i - 1) do
                  dist ← (xj - xi)² + (yj - yi)²
                  if  dist < min then
                        min ← dist
                        closest ← {(xi, yi), (xj, yj)}
                  endif
            endfor
      endfor
      → closest
```

TAMAT