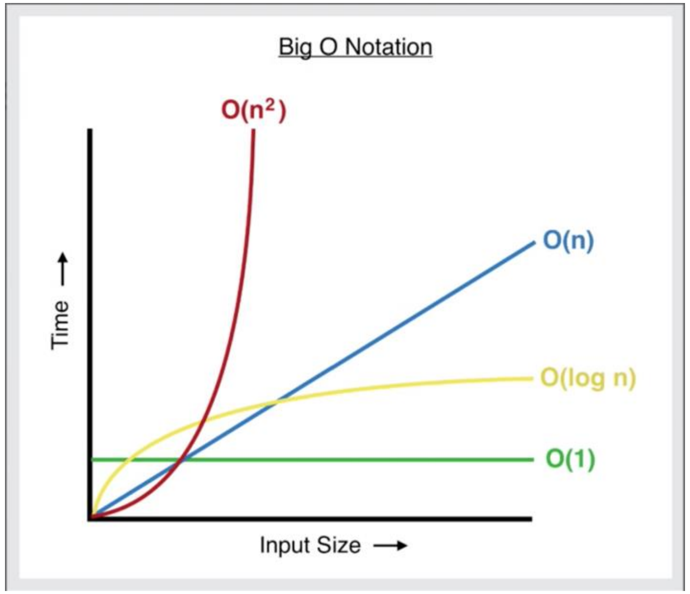


Kompleksitas Algoritma (Bagian 1)

Bahan Kuliah

IF2120 Matematika Diskrit

Oleh: Rinaldi Munir

Program Studi Teknik Informatika
STEI - ITB

```
for (i = 1; i <= n, i++)   {
    for (j =1; j <= n; j++) {
         for (k =1; k <= j; k++) {
            p = p * 20 * z;
         }
    }
}
```

Pendahuluan

- Sebuah algoritma tidak saja harus benar (sesuai spesifikasi persoalan), tetapi juga harus sangkil (*efisien*).

- Algoritma yang bagus adalah algoritma yang sangkil (*efficient*).

- Kesangkilan algoritma diukur dari waktu (*time*) yang diperlukan untuk menjalankan algoritma dan ruang (*space*) memori yang dibutuhkan oleh algoritma tersebut.

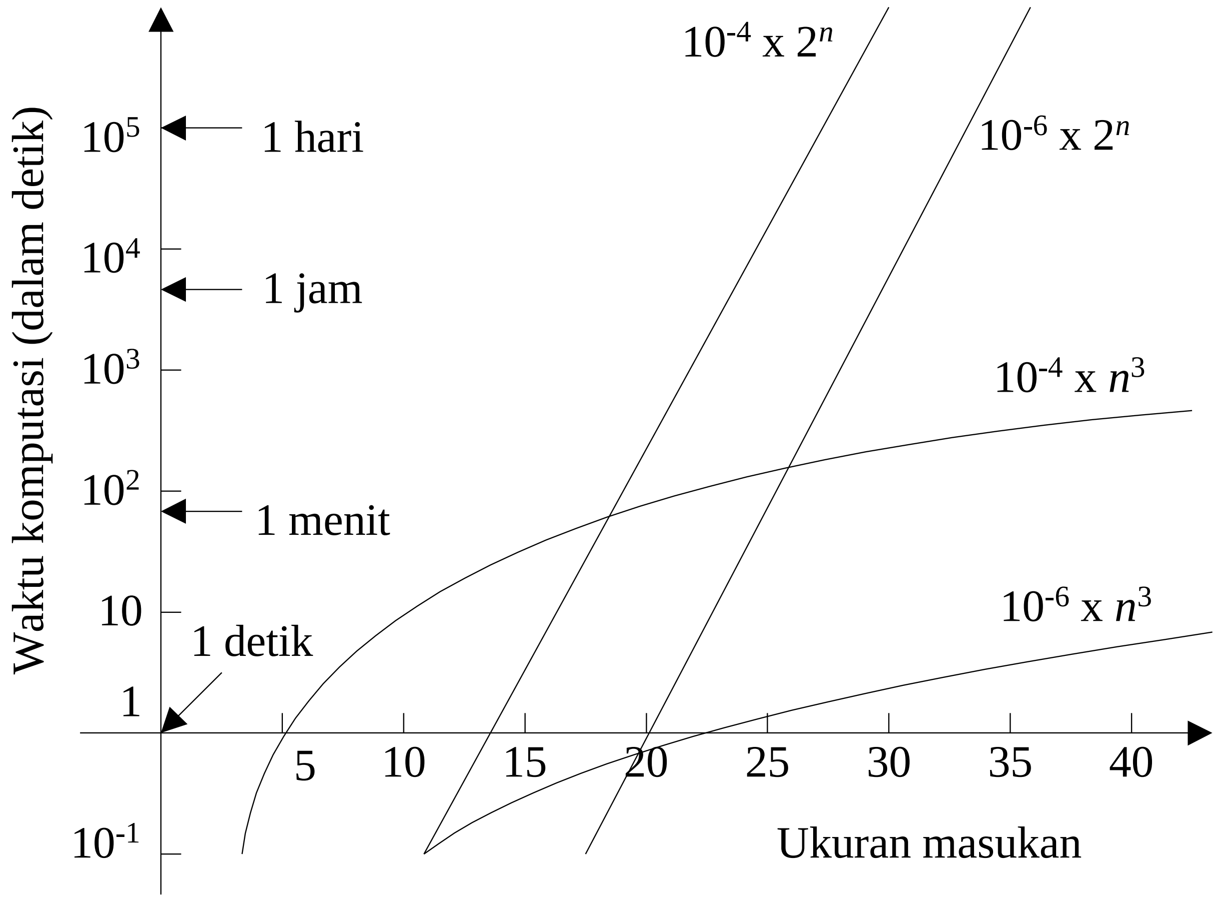
- Algoritma yang sangkil ialah algoritma yang **meminimumkan** kebutuhan waktu dan ruang memori.

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

- Kesangkilan algoritma dapat digunakan untuk menilai algoritma yang bagus dari sejumlah algoritma penyelesaian persoalan.

- Sebab, sebuah persoalan dapat memiliki banyak algoritma penyelesaian. Contoh: persoalan pengurutan (*sort*), ada puluhan algoritma pengurutan (*selection sort*, *insertion sort*, *bubble sort*, dll).

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



Model Perhitungan Kebutuhan Waktu

- Menghitung kebutuhan waktu algoritma dengan mengukur waktu eksekusi riil nya (dalam satuan detik) ketika program (yang merepresentasikan sebuah algoritma) dijalankan oleh komputer bukanlah cara yang tepat.

- Alasan:

  1. Setiap komputer dengan arsitektur berbeda memiliki bahasa mesin yang berbeda → waktu setiap operasi antara satu komputer dengan komputer lain tidak sama.

  2. *Compiler* bahasa pemrograman yang berbeda menghasilkan kode Bahasa mesin yang berbeda → waktu setiap operasi antara *compiler* dengan *compiler* lain tidak sama.

- Model abstrak pengukuran waktu/ruang memori algoritma harus independen dari pertimbangan mesin (*computer*) 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** (*time complexity*) dan **kompleksitas ruang** (*space complexity*).

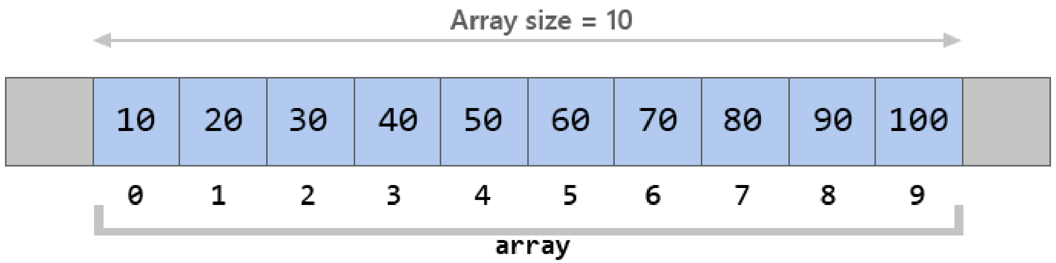
- Kompleksitas waktu, $T(n)$, diukur dari jumlah tahapan komputasi yang dilakukan di dalam algoritma sebagai fungsi dari ukuran masukan n.

- 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.

- Di dalam kuliah ini kita hanya membatasi bahasan kompleksitas waktu saja, karena dua alasan:
  1. Materi struktur data diluar lingkup mata kuliah matematika diskrit
  2. Saat ini memori komputer bukan persoalan yang kritis dibandingkan waktu

- Ukuran masukan (*n*) menyatakan banyaknya data yang diproses oleh sebuah algoritma.



Contoh:

1. algoritma pengurutan 10 elemen larik (*array*), maka n = 10.
2. algoritma pencarian pada 500 elemen larik, maka n = 500
3. algoritma *TSP* pada sebuah graf lengkap dengan 100 simpul, maka n = 100.
4. algoritma perkalian 2 buah matriks berukuran 50 x 50, maka n = 50.
5. algoritma menghitung polinom dengan derajat $\leq$ 100, maka n = 100

- Dalam perhitungan kompleksitas waktu, ukuran masukan dinyatakan sebagai variabel *n* saja (bukan instans suatu nilai).

Kompleksitas Waktu

- Pekerjaan utama di dalam kompleksitas waktu adalah menghitung (*counting*) jumlah tahapan komputasi di dalam algoritma .

- Jumlah tahapan komputasi dihitung dari berapa kali suatu operasi dilakukan sebagai fungsi ukuran masukan (*n*).

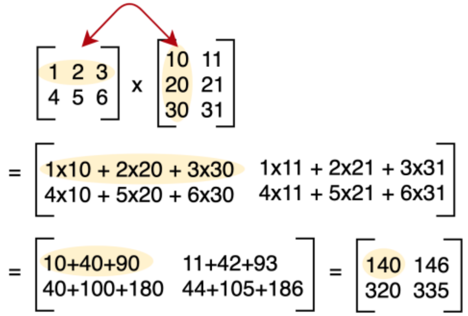
- Di dalam sebuah algoritma terdapat banyak jenis operasi:
  - Operasi baca/tulis (input a, print a)
  - Operasi aritmetika (+, -, *, /) (a + b, M * N)
  - Operasi pengisian nilai (*assignment*) (a $\leftarrow$ 10)
  - Operasi perbandingan (a < b, k >= 10)
  - Operasi pengaksesan elemen larik, pemanggilan prosedur/fungsi, dll

- Untuk menyederhanakan perhitungan, kita tidak menghitung semua jenis operasi, tetapi kita hanya menghitung jumlah operasi khas (tipikal) yang *mendasari* suatu algoritma.

Contoh operasi khas di dalam algoritma

- Algoritma pencarian (*searching*)
  Operasi khas: operasi perbandingan elemen larik

- Algoritma pengurutan (*sorting*)
  Operasi khas: operasi perbandingan elemen dan operasi pertukaran elemen



- Algoritma perkalian dua buah matriks $AB = C$
  Operasi khas: operasi perkalian dan penjumlahan

- Algoritma menghitung nilai sebuah polinom $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$
  Operasi khas: operasi perkalian dan penjumlahan

Contoh 1. Tinjau algoritma menghitung rerata elemen di dalam sebuah larik (*array*).

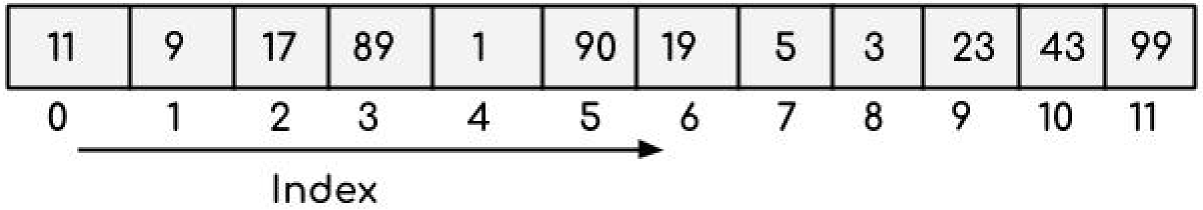
$sum \leftarrow 0$

for $i \leftarrow 1$ **to** n **do**

    $sum \leftarrow sum + a[i]$

endfor

$rata\_rata \leftarrow sum/n$



- Operasi yang mendasar pada algoritma tersebut adalah operasi penjumlahan elemen-elemen larik (yaitu $sum \leftarrow sum+a[i]$) yang dilakukan sebanyak *n* kali.

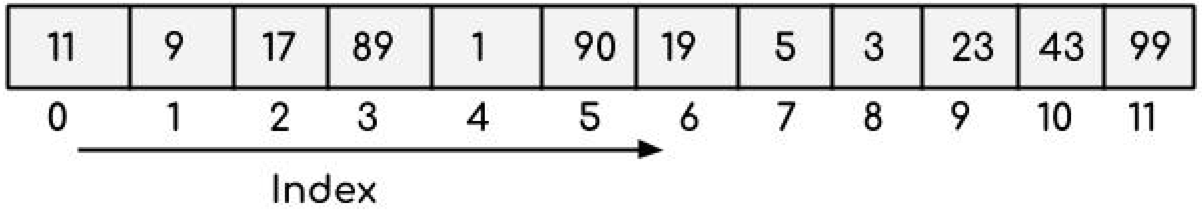
- Kompleksitas waktu: $T(n) = n$.

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

procedure *CariElemenTerbesar*(**input** $a_1, a_2, ..., a_n$: **integer**, **output** *maks* : **integer**)
{ Mencari elemen terbesar dari sekumpulan elemen larik integer $a_1, a_2, ..., a_n$.
Elemen terbesar akan disimpan di dalam maks. }

Deklarasi
  k : **integer**

Algoritma
  $maks \leftarrow a_1$
  $k \leftarrow 2$
  while $k \leq n$ **do**
    if $a_k > maks$ **then**
      $maks \leftarrow a_k$
    endif
    $k \leftarrow k + 1$
  endwhile



Kompleksitas waktu algoritma dihitung dari jumlah operasi perbandingan elemen larik ($a_k > maks$).
Kompleksitas waktu *CariElemenTerbesar* : $T(n) = n - 1$.

Kompleksitas waktu dibedakan atas tiga macam :

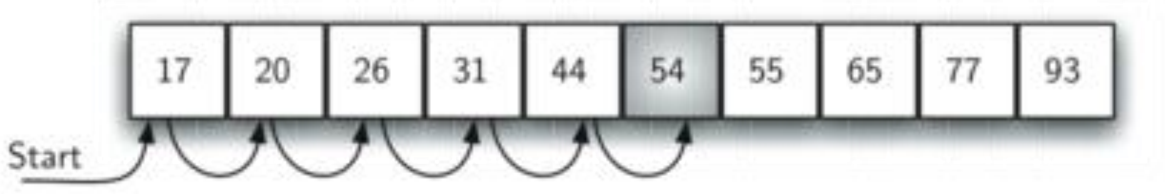
1. $T_{max}(n)$: kompleksitas waktu untuk kasus terburuk *(worst case),*
   → kebutuhan waktu maksimum.

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* (linear search)

```
procedure PencarianBeruntun(input a1, a2, ..., an : integer, x : integer,  output idx : integer)
{ Mencari elemen x di dalam larik A yang berisi n elemen. Jika x ditemukan, maka indeks elemen larik disimpan
   di dalam idx, idx bernilai  –1 jika x tidak ditemukan
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
  if  ketemu then   { x ditemukan }
      idx←k
  else
      idx← –1       { 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 (`ak = x`) akan dieksekusi sebanyak j kali.

$$T_{\mathrm{avg}}(n) = \frac{(1+2+3+\ldots+n)}{n} = \frac{\frac{1}{2}n(1+n)}{n} = \frac{(n+1)}{2}$$

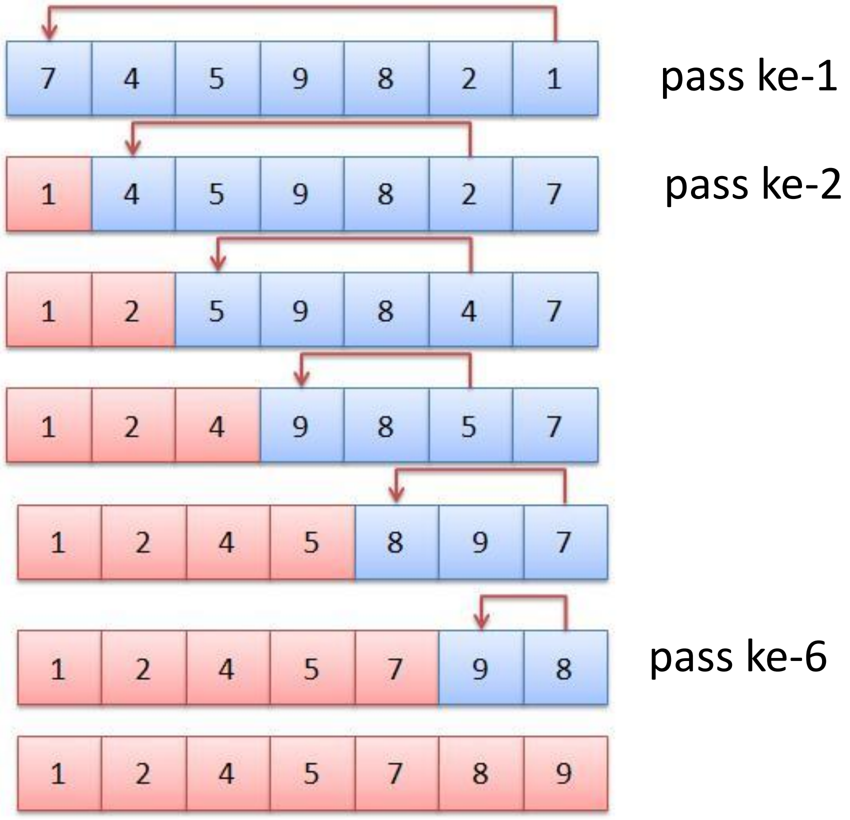
Cara lain: asumsikan bahwa $P(a_j = x) = 1/n$. Jika $a_j = x$ maka T_j yang dibutuhkan adalah $T_j = j$. Jumlah perbandingan elemen larik rata-rata:

$$T_{\text{avg}}(n) = \sum_{j=1}^{n} T_j \, P(a[j] = x) = \sum_{j=1}^{n} T_j \frac{1}{n} = \frac{1}{n} \sum_{j=1}^{n} T_j$$

$$= \frac{1}{n} \sum_{j=1}^{n} j = \frac{1}{n} \left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

Contoh 4: Algoritma pengurutan seleksi (*selection sort*)

procedure *SelectionSort*(**input/output** $a_1, a_2, ..., a_n$: **integer**)
{ *Mengurutkan elemen-elemen larik A yang berisi n elemen integer sehingga terurut menaik* }
Deklarasi
  i, j, imin, temp : **integer**
Algoritma
  for $i \leftarrow 1$ **to** $n - 1$ **do** { *pass sebanyak n – 1 kali* }
    $imin \leftarrow i$
    for $j \leftarrow i + 1$ **to** n **do**
      if $a_j < a_{imin}$ **then**
        $imin \leftarrow j$
      endif
    endfor
    { *pertukarkan* a_{imin} *dengan* a_i }
    $temp \leftarrow a_i$
    $a_i \leftarrow a_{imin}$
    $a_{imin} \leftarrow temp$
  endfor



(i) Jumlah operasi perbandingan elemen-elemen larik ($a_j < a_{imin}$)

Untuk setiap pass ke-*i*,

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

i = 2 $\rightarrow$ jumlah perbandingan = n – 2

i = 3 $\rightarrow$ jumlah perbandingan = n – 3

$\vdots$

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

```
for i←1 to n – 1 do   { pass sebanyak n – 1 kali }
   imin←i
   for j←i + 1 to n do
      if a_j < a_imin then
         imin←j
      endif
   endfor
   { pertukarkan a_imin dengan a_i }
   temp←a_i
   a_i←a_imin
   a_imin←temp
endfor
```

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah

$$T(n) = (n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

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

```
for i←1 to n – 1 do   { pass sebanyak n – 1 kali }
   imin←i
   for j←i + 1 to n do
      if a_j < a_imin then
         imin←j
      endif
   endfor
   { pertukarkan a_imin dengan a_i }
   temp←a_i
   a_i←a_imin
   a_imin←temp
endfor
```

(ii) Jumlah operasi pertukaran

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

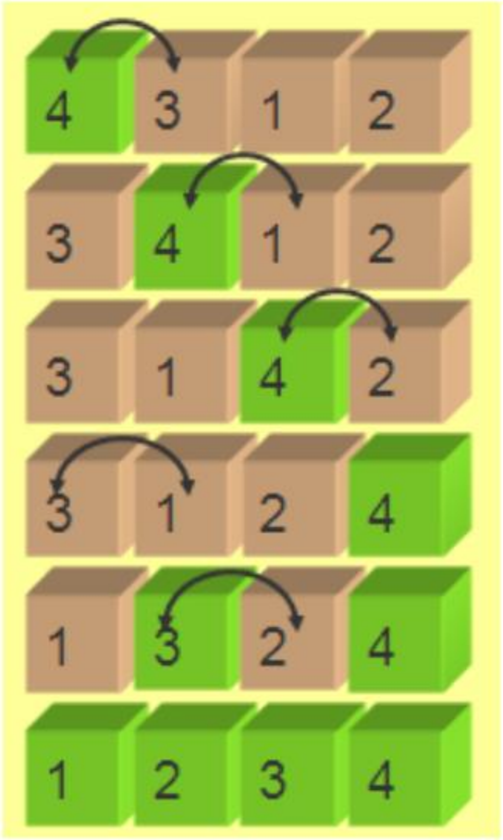
$$T(n) = n – 1.$$

Ini adalah jumlah pertukaran untuk semua kasus.

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

Contoh 5: Diberikan algoritma pengurutan *bubble-sort* seperti berikut ini. Hitung kompleksitas waktu algoritma didasarkan pada jumlah operasi perbandingan elemen-elemen larik dan jumlah operasi pertukaran.

procedure *BubbleSort*(**input/output** $a_1, a_2, ..., a_n$: **integer**)
{ *Mengurut larik A yang berisi n elemen integer sehingga terrut menaik* }
Deklarasi
  $i, j, temp$: **integer**
Algoritma
  for $i \leftarrow n - 1$ **downto** 1 do
    for $j \leftarrow 1$ **to** i do
      if $a_{j+1} < a_j$ **then**
        { *pertukarkan* a_j *dengan* a_{j+1} }
        $temp \leftarrow a_j$
        $a_j \leftarrow a_{j+1}$
        $a_{j+1} \leftarrow temp$
      endif
    endfor
  endfor



Sumber gambar: Prof. Amr Goneid, Department of Computer Science, AUC

(i) Jumlah operasi perbandingan elemen-elemen larik ($a_{j+1} < a_j$)

Untuk setiap *pass* ke-*i*,

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

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

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

$\vdots$

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

```
for i ← n – 1 downto 1 do
   for j ← 1 to i do
      if a_{j+1} < a_j then
         { pertukarkan a_j dengan a_{j-1} }
         temp ← a_j
         a_j ← a_{j+1}
         a_{j+1} ←temp
      endif
   endfor
endfor
```

Jumlah seluruh operasi perbandingan elemen-elemen larik adalah

$$T(n) = (n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Ini adalah kompleksitas waktu untuk kasus terbaik dan terburuk, karena algoritma *BubbleSort* tidak bergantung pada apakah data masukannya sudah terurut atau acak. Jumlah operasi perbandingan sama dengan *selection sort.*

(ii) Jumlah operasi pertukaran ($temp \leftarrow a_i$; $a_i \leftarrow a_{imin}$; $a_{imin} \leftarrow temp$ **)**

Jumlah operasi pertukaran di dalam *bubble sort* hanya dapat dihitung pada kasus terbaik dan kasus terburuk. Kasus terbaik adalah tidak ada pertukaran (yaitu jika **if** $a_{j+1} < a_j$ false), yaitu semua elemen larik pada awalnya sudah terurut menaik, sehingga

$$T_{min}(n) = 0.$$

Pada kasus terburuk, (yaitu jika **if** $a_{j+1} < a_j$ bernilai true), pertukaran elemen selalu dilakukan. Jadi, jumlah operasi pertukaran elemen pada kasus terburuk sama dengan jumlah operasi perbandingan elemen-elemen larik, yaitu

$$T_{max}(n) = (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

Jadi, algoritma pengurutan *bubble sort* membutuhkan $n(n - 1)/2$ buah operasi pertukaran, lebih banyak daripada algoritma *selection sort*. Ini berarti secara keseluruhan *bubble sort* lebih buruk daripada *selection sort*.

Latihan 1

Hitung kompleksitas waktu algoritma berikut berdasarkan jumlah operasi perkalian.

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. 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  ← j  div 2
endwhile
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

$= n + n/2 + n/4 + \ldots + 2 + 1$ → deret geometri

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

```
j ← n
while j ≥ 1 do
   for i ← 1 to j do
      x ← x * i
   endfor
   j ← j div 2
endwhile
jumlah ← x
```

Latihan 2

Di bawah ini adalah algoritma untuk menguji apakah dua buah matriks, *A* dan *B*, yang masing-masing berukuran $n \times n$, sama.

function *samaMatriks*(*A*, *B* : *matriks*; *n* : integer) $\rightarrow$ *boolean*
{ true jika A dan B sama; sebaliknya false jika A $\neq$ *B }*
Deklarasi
  i, *j* : **integer**
Algoritma:
  for $i \leftarrow 1$ **to** n **do**
    for j $\leftarrow 1$ **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 terburuknya.

Jawaban:

(a) Kasus terbaik terjadi jika ketidaksamaan matriks ditemukan pada elemen pertama ($A_{1,1} \neq B_{1,1}$)

Kasus terburuk terjadi jika ketidaksamaan matriks ditemukan pada elemen ujung kanan bawah ($A_{n,n} \neq B_{n,n}$) atau pada kasus matriks A dan B sama, sehingga seluruh elemen matriks dibandingkan.

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

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

Latihan Mandiri

1. Diberikan matriks persegi berukuran n x n. Hitung kompleksitas waktu untuk memeriksa apakah matriks tersebut merupakan matriks simetri terhadap diagonal utama.

2. Berapa kompleksitas waktu untuk menjumlahkan matriks A dan B yang keduanya berukuran n x n?

3. Ulangi soal 2 untuk perkalian matriks A dan B.

4. Tulislah algoritma pengurutan *insertion sort* pada larik yang berukuran n elemen, hitung masing-masing kompleksitas waktu algoritma diukur dari jumlah operasi perbandingan dan jumlah operasi pertukaran elemen-elemen larik.

5. Berapa kali operasi penjumlahan pada potongan algoritma ini dilakukan?

for $i \leftarrow 1$ **to** n **do**

  for $j \leftarrow 1$ **to** n **do**

    for $k \leftarrow 1$ to j do

      $x \leftarrow x + 1$

    endfor

  endfor

endfor

6. Algoritma di bawah ini menghitung nilai polinom $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$

```
function p(input x:real)→real
{ Mengembalikan nilai p(x)}
Deklarasi
    j, k : integer
    jumlah, suku : real
Algoritma
   jumlah ← a0
   for  j ← 1 to n do
     { hitung aj x^j }
      suku ← aj
      for  k ← 1 to j do
           suku ← suku * x
       endfor
       jumlah ← jumlah + suku
    endfor
    return jumlah
```

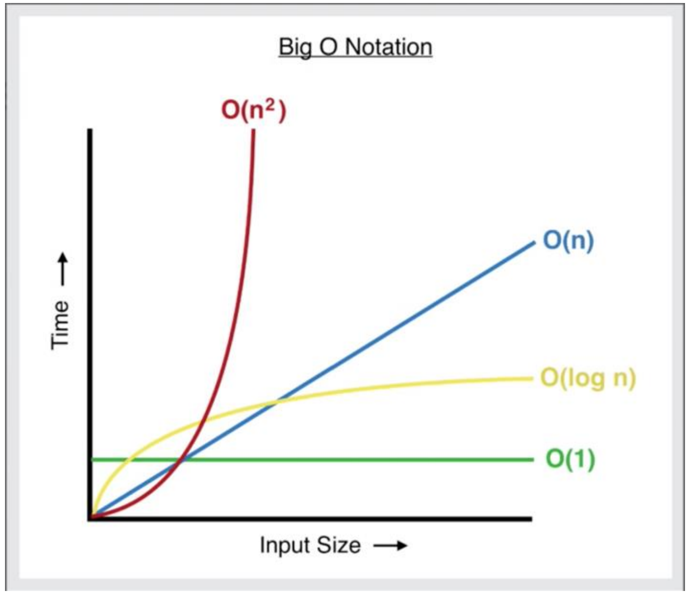
Hitunglah berapa operasi perkalian dan berapa operasi penjumlahan yang dilakukan oleh algoritma tsb

Algoritma menghitung polinom yang lebih baik dapat dibuat dengan metode Horner berikut: $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + a_n x)))\ldots))$

```
function p2(input x:real)→real
{ Mengembalikan nilai p(x) dengan metode Horner}
 Deklarasi
    k : integer
    b1, b2, ..., bn : real
Algoritma
   bn←an
   for k←n – 1 downto 0 do
         bk←ak + bk+1 * x
   endfor
   return b0
```

Hitunglah berapa operasi perkalian dan berapa operasi penjumlahan yang dilakukan oleh algoritma di atas? Manakah yang terbaik, algoritma *p* atau *p*2?

BERSAMBUNG



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

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

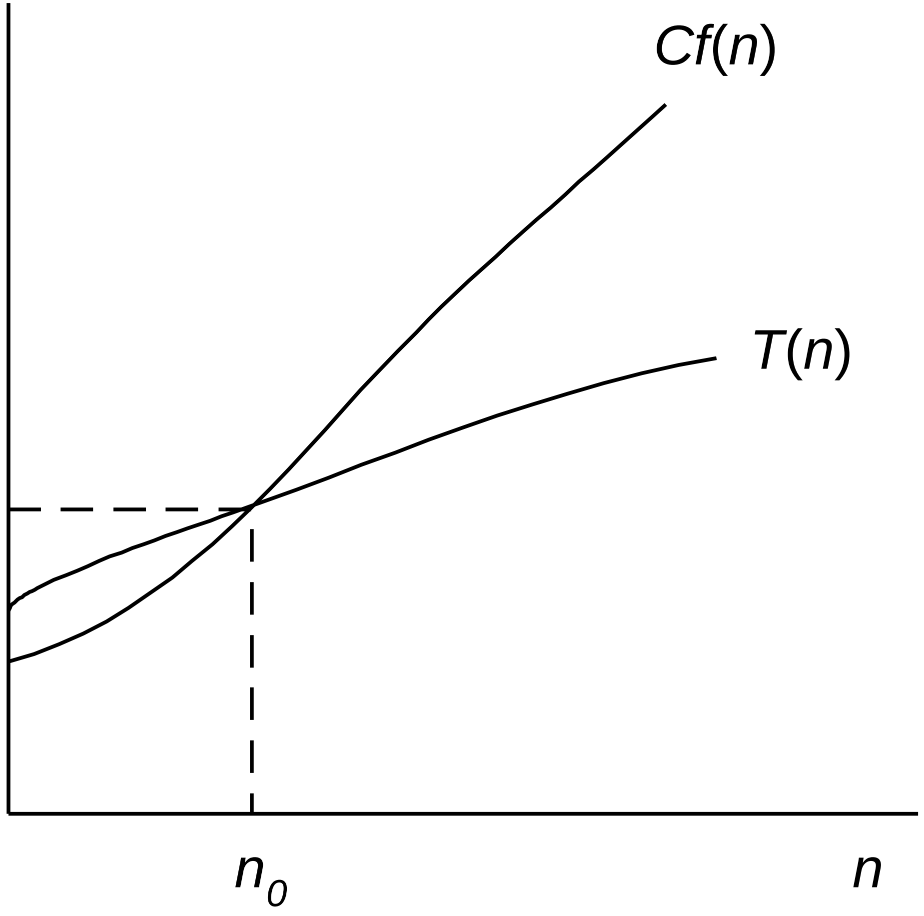
untuk $n \geq n_0$.

- $f(n)$ adalah batas lebih atas (*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

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

untuk $n \ge n_0$.

Contoh-contoh Lain

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

Jawaban:

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

Kita juga dapat memperlihatkan bahwa

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

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

Jawaban:

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

karena

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

untuk $n \ge 1$

$(C = 1, f(n) = n^2,$ 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 $\log n! = O(n \log n)$

Jawaban:

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 $8n^2 = O(n^3)$

Jawaban:

$8n^2 = O(n^3)$ 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 $T(n) = n^3 \neq O(n^2)$.

Jawaban:

- $5n^2 = O(n^3)$ 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 $T_1(\text{n}) = O(f(n))$ 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 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: • $n + 1 = O(n)$

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

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

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

$= O(\log n)$

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

• $3n^2 = O(n^2)$

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

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

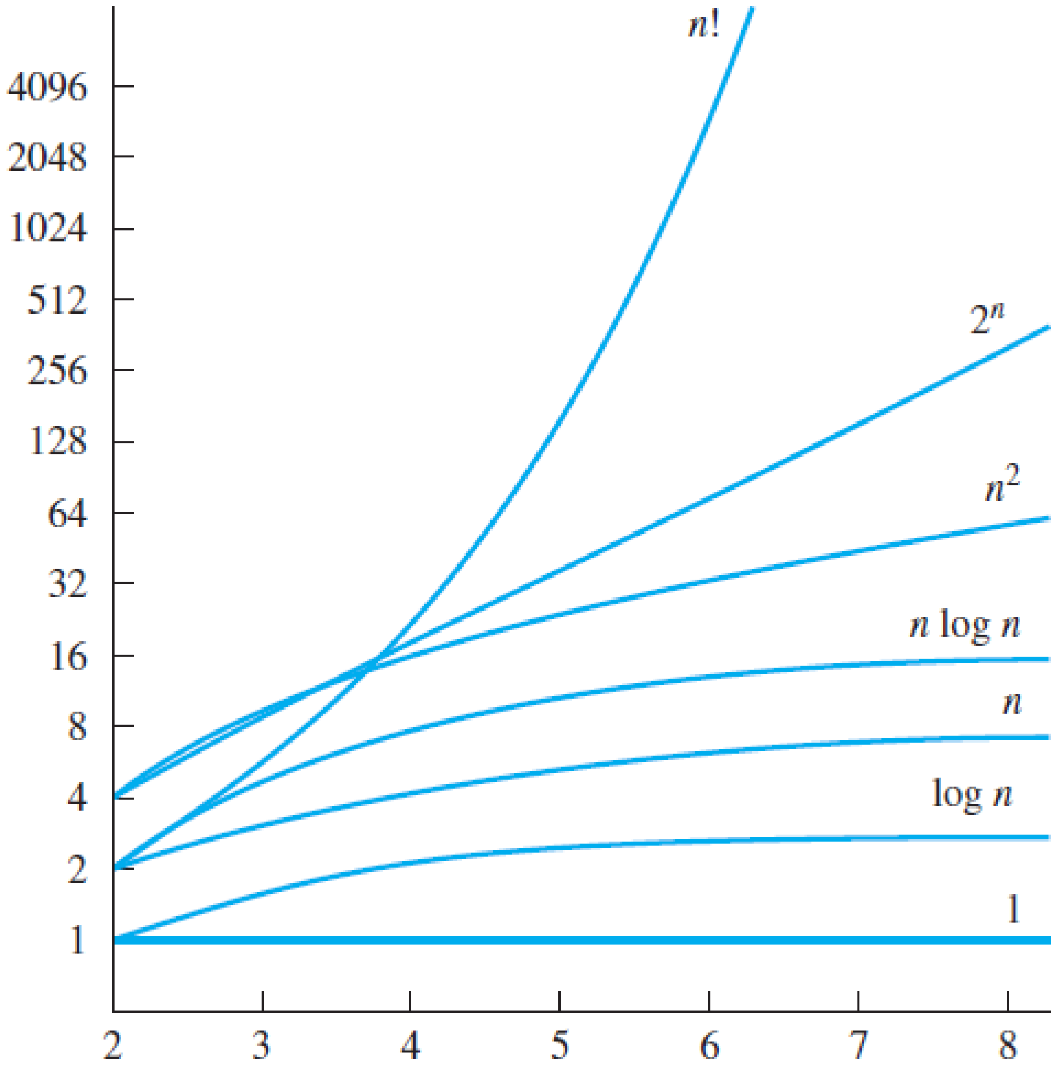
(bagus)

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 $1 + 2 + \ldots + n \le n + n + \ldots + n = n^2$ untuk $n \ge 1$.

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

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

Atau, dengan cara kedua:

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

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 $T(n) = a_m n^m + a_{m-1} n^{m-1} + ... + a_1 n + a_0$ 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 ← 1 to n do
   for j ← 1 to i do
      for k ← j to n do
         a ← 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 $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow$ **1 to** i **do**
    for $k \leftarrow j$ **to** n **do**
      $a \leftarrow a + 1$
    endfor
  endfor
endfor

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

- 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 $T(n) \le C f(n)$ untuk semua $n \ge n_0$:
   (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), ..., (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