

## 2 - Analisis Kompleksitas Komputasional

[KOMS120403]

Desain dan Analisis Algoritma (2022/2023)

Dewi Sintiar

Prodi S1 Ilmu Komputer  
Universitas Pendidikan Ganesha

Week 2 (February 2023)

# Daftar isi

- Review algoritma fpb
- Model kompleksitas komputasi
- Notasi asimtotik dan urutan besarnya
- Big-O notasi: batas atas asimtotik
  - ▶ Definisi
  - ▶ Linear & fungsi polinomial
  - ▶ Operasi aritmatika di  $\mathcal{O}$
  - ▶ Fungsi logaritmik
  - ▶ Klasifikasi algoritma
  - ▶ Menentukan kompleksitas asimtotik
- Notasi Big-Omega
- Notasi Big-Theta
- Quiz

# Bagian 1: Motivasi

# Algoritma untuk menghitung fpb

*Ingatlah materi minggu lalu...*

## Menghitung fpb:

- Input: bilangan bulat  $a$  dan  $b$
- Output: fpb dari  $m$  dan  $n$

---

### Algorithm 1 Algoritma sederhana fpb dari dua bilangan bulat

---

```
1: procedure FPB( $a, b$ )  
2:    $r = 1$   
3:    $x = \min(a, b)$   
4:   for  $i = 1$  to  $x$  do  
5:     if  $a \bmod i == 0$  and  $b \bmod i == 0$  then  $r = i$   
6:     end if  
7:   end for  
8: end procedure
```

---

Tentukan **kompleksitas**-nya!

Coba bandingkan algoritma tersebut dengan algoritma berikut (yang disebut dengan algoritma Euclid)

# Algoritma Euclid untuk menghitung fpb (1)

## Contoh

*Menggunakan algoritma Euclid, tentukan fpb dari 210 dan 45.*

**Solusi:**

# Algoritma Euclid untuk menghitung fpb (1)

## Contoh

*Menggunakan algoritma Euclid, tentukan fpb dari 210 dan 45.*

## Solusi:

$$210 = 4 \cdot 45 + 30$$

$$45 = 1 \cdot 30 + 15$$

$$30 = 2 \cdot 15 + 0$$

Jadi  $\text{fpb}(210, 45) = 15$

## Algoritma Euclidean untuk menghitung fpb (2)

---

### Algorithm 2 Euclidean algorithm

---

```
1: procedure EUCLIDFPB( $a, b$ )
2:   while  $b \neq 0$  do
3:      $r = a \bmod b$ 
4:      $a = b$ 
5:      $b = r$ 
6:   end while
7:   return  $a$ 
8: end procedure
```

---

Mengapa algoritma tersebut berakhir (tidak mengalami *infinite looping*)?  
(lihat Teorema Lame

<https://www.cut-the-knot.org/blue/LamesTheorem.shtml>)

**Tentukan kompleksitas algoritma!** kerjakan sebagai latihan!



## Bagian 2: Kompleksitas algoritma

# Model kompleksitas komputasi (1)

Dapatkah Anda menjelaskan kembali definisi dari **kompleksitas** algoritma, dan mengapa hal tersebut penting?

## Model kompleksitas komputasi (2)

Bagian dari *analisis algoritma* adalah menghitung *kompleksitas komputasi* dari suatu algoritma.

**Kompleksitas komputasional** atau cukup disebut **kompleksitas** dari sebuah algoritma adalah banyaknya sumber daya (*waktu* dan *memori*) yang diperlukan untuk menjalankannya.

- **Efisiensi waktu**: seberapa cepat suatu algoritma dijalankan
- **Efisiensi ruang**: berapa banyak memori yang dibutuhkan untuk mengeksekusi suatu algoritma

Bagaimana kompleksitas suatu algoritma dihitung?

# Apa pengaruh kompleksitas komputasi suatu algoritma?

## Contoh

Misalkan sebuah **superkomputer** mengeksekusi algoritma A, dan sebuah **PC** (personal computer) mengeksekusi algoritma B. Kedua komputer harus mengurutkan 1 juta elemen. Superkomputer dapat mengeksekusi 100 juta instruksi dalam satu detik, sedangkan PC hanya mampu mengeksekusi 1 juta instruksi dalam satu detik.

- Algoritma A membutuhkan  $2n^2$  instruksi untuk mengurutkan  $n$  elemen;
- Algoritma B membutuhkan  $50n \log n$  instruksi

Hitunglah banyaknya waktu yang dibutuhkan untuk mengurutkan 1 juta elemen di setiap komputer (superkomputer dan PC)!

# Apa pengaruh kompleksitas komputasi suatu algoritma?

*Dapatkah Anda menebak, secara intuitif, komputer manakah yang memiliki waktu eksekusi lebih singkat?*

# Apa pengaruh kompleksitas komputasi suatu algoritma?

*Dapatkan Anda menebak, secara intuitif, komputer manakah yang memiliki waktu eksekusi lebih singkat?*

## Solusi:

- Superkomputer:  $\frac{2 \cdot (10^6)^2 \text{ instructions}}{10^8 \text{ instructions / sec}} = 20000 \text{ sec} \approx 5.56 \text{ hours}$
- PC:  $\frac{50 \cdot 10^6 \log 10^6 \text{ instructions}}{10^6 \text{ instructions / sec}} \approx 1000 \text{ sec} \approx 16.67 \text{ minutes}$

*Apa yang dapat Anda simpulkan?*

# Apa yang memengaruhi kompleksitas komputasi?

Kompleksitas waktu (dan ruang) bergantung pada banyak hal seperti *hardware*, *OS*, *processors*, *programming language* dan *compiler*, dll. Tapi kita tidak pertimbangkan faktor-faktor ini saat menganalisis kompleksitas algoritma.

## Beberapa catatan dalam mempelajari kompleksitas algoritma:

- Fokus kita pada subjek ini adalah pada **kompleksitas waktu**.
- Kita berasumsi bahwa mesin kita hanya menggunakan satu prosesor (yaitu *generic one-processor*).
- Kompleksitas waktu dihitung berdasarkan **banyaknya operasi/instruksi**
- *Running time* dari suatu algoritma dihitung sebagai ukuran input ( $n$ ), dan merupakan fungsi yang *tak-turun* (*non-decreasing*).

# Contoh penghitungan kompleksitas komputasi

---

**Algorithm 3** Rata-rata array bilangan bulat

---

```
1: procedure AVERAGE( $A[1..n]$ )
2:    $sum \leftarrow 0$ 
3:   for  $i = 1$  to  $n$  do
4:      $sum \leftarrow sum + A[i]$ 
5:   end for
6:    $avg \leftarrow sum/n$ 
7: end procedure
```

---

Jumlah operasi:

- Penugasan: baris 2, 4, 6; dengan operasi  $1 + n + 1 = n + 2$
- Penjumlahan: baris 4, dengan operasi  $n$
- Divisi: baris 6, dengan 1 operasi

**Kompleksitas waktu:**  $T(n) = (n + 2) + n = 2n + 2$  operations.



## Bagian 3: Tiga macam kompleksitas algoritma

## Tiga macam pengukuran penggunaan sumber daya

- **Kasus terburuk** ( $T_{\max}(n)$ ): ini mengukur sumber daya (mis. waktu berjalan, memori) yang diperlukan algoritma dalam **kasus terburuk** yaitu **paling sulit case**, diberi input ukuran acak  $n$  (biasanya dilambangkan dengan notasi asimtotik).
- **Kasus terbaik** ( $T_{\min}(n)$ ): jelaskan perilaku algoritma dalam **kondisi optimal**.
- **Kasus rata-rata** ( $T_{\text{avg}}(n)$ ): jumlah waktu komputasi yang digunakan oleh algoritma, **rata-rata dari semua input yang mungkin**.

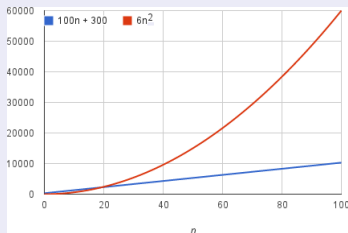
# Notasi asimtotik dan urutan besarnya (1)

- *Running time* suatu algoritma diukur sebagai *fungsi ukuran inputnya*.
- **Rate of growth** dari waktu berjalan mengukur seberapa cepat suatu fungsi tumbuh dengan ukuran input. Secara **asimptotis** berarti fungsi itu penting *hanya untuk nilai  $n$  yang besar*.
- Fungsi **order of magnitude** menjelaskan bagian dari fungsi yang meningkat paling cepat saat nilai  $n$  meningkat.

## Notasi asimtotik dan urutan besarnya (2)

### Contoh

Misalkan sebuah algoritma berjalan pada input berukuran  $n$ , membutuhkan  $6n^2 + 100n + 300$  eksekusi.



Kita hanya menyimpan **suku yang paling “penting”**. Dalam hal ini, fungsi  $6n^2$  memiliki nilai yang lebih dari  $100n + 300$  untuk setiap nilai  $n$  dalam batas bawah tertentu.

## Notasi $\mathcal{O}$ (O-besar/*big-O*): Batas-atas asimtotik

Kompleksitas kasus terburuk mengukur sumber daya yang dibutuhkan algoritma dalam *kasus terburuk*. Ini memberikan **upper bound** pada sumber daya yang dibutuhkan oleh algoritma.

### Mengapa mempelajari kompleksitas kasus terburuk?

- memberikan informasi tentang kebutuhan sumber daya maksimum
- secara alami, hal ini sering terjadi pada suatu sistem

**Notasi Big-O ( $\mathcal{O}(\cdot)$ ):** notasi matematika yang menjelaskan perilaku pembatas fungsi ketika argumen cenderung ke nilai tertentu atau tak terhingga.

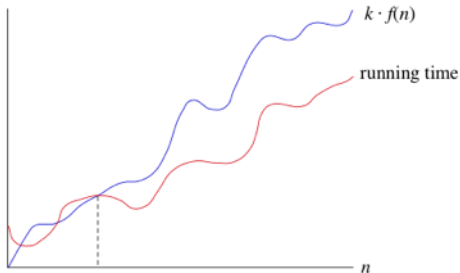
#### Definisi

$g(n) \in \mathcal{O}(f(n))$  if  $\exists k > 0$  dan  $n_0$  s.t.  $g(n) \leq k \cdot f(n)$ ,  $\forall n \geq n_0$ .

# Notasi $\mathcal{O}$ (O-besar/*big-O*): Batas atas asimtotik

## Definisi

$g(n) \in \mathcal{O}(f(n))$  if  $\exists k > 0$  dan  $n_0$  s.t.  $g(n) \leq k \cdot f(n)$ ,  $\forall n \geq n_0$ .



## Contoh notasi $\mathcal{O}$ (1): Fungsi linier

### Contoh

*Tunjukkan bahwa  $g(n) = 5n + 3$  ada di  $\mathcal{O}(n)$ .*

## Contoh notasi $\mathcal{O}$ (1): Fungsi linier

### Contoh

*Tunjukkan bahwa  $g(n) = 5n + 3$  ada di  $\mathcal{O}(n)$ .*

### Solusi:

Perhatikan bahwa  $5n + 3 \leq 5n + 3n = 8n$  untuk semua  $n \geq 1$ . Dalam hal ini,  $k = 8$  dan  $n_0 = 1$ . Jadi,  $g(n) \in \mathcal{O}(n)$ .



## Contoh notasi $\mathcal{O}$ (2): Fungsi polinomial

### Contoh

*Tunjukkan bahwa  $g(n) = 3n^2 - 5n + 6$  ada di  $\mathcal{O}(n^2)$ .*

## Contoh notasi $\mathcal{O}$ (2): Fungsi polinomial

### Contoh

Tunjukkan bahwa  $g(n) = 3n^2 - 5n + 6$  ada di  $\mathcal{O}(n^2)$ .

### Solusi:

Perhatikan bahwa  $3n^2 - 5n + 6 \leq 3n^2 + 0 + 6n^2 = 9n^2$  untuk semua  $n \geq 1$ . Dalam hal ini,  $k = 9$  dan  $n_0 = 1$ . Jadi,  $g(n) \in \mathcal{O}(n^2)$ .

## Bagian 3: Operasi aritmatika di $\mathcal{O}$

# Operasi aritmatika di $\mathcal{O}$

Fungsi kompleksitas waktu dilambangkan dengan  $T(n)$ .

## Teorema (Big-O dari kompleksitas polinomial)

Jika  $T(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$  polinomial dari urutan  $m$ , maka  $T(n) \in \mathcal{O}(n^m)$ .

## Teorema (Operasi aritmatika dengan Big-O)

Let  $T_1(n) \in \mathcal{O}(f(n))$  dan  $T_2(n) \in \mathcal{O}(g(n))$ , then:

- 1  $T_1(n) + T_2(n) \in \mathcal{O}(f(n)) + \mathcal{O}(g(n)) \in \mathcal{O}(\max(f(n), g(n)))$
- 2  $T_1(n) T_2(n) \in \mathcal{O}(f(n)) \mathcal{O}(g(n)) \in \mathcal{O}(f(n)g(n))$
- 3  $\mathcal{O}(cf(n)) \in \mathcal{O}(f(n))$ , dimana  $c$  adalah konstanta
- 4  $f(n) \in \mathcal{O}(f(n))$

**Proof:** tugas!

# Operasi aritmatika dengan $\mathcal{O}$

## Contoh (Operasi aritmatika dengan Big-O)

- ① Misalkan  $T_1(n) \in \mathcal{O}(n)$  dan  $T_2(n) \in \mathcal{O}(n^2)$ , maka:

$$T_1(n) + T_2(n) \in \mathcal{O}(\max(n, n^2)) \in \mathcal{O}(n^2)$$

- ② Misalkan  $T_1(n) \in \mathcal{O}(n)$  dan  $T_2(n) \in \mathcal{O}(n^2)$ , maka:

$$T_1(n)T_2(n) \in \mathcal{O}(n \cdot n^2) = \mathcal{O}(n^3)$$

- ③  $\mathcal{O}(5n^2) \in \mathcal{O}(n^2)$

- ④  $n^2 \in \mathcal{O}(n^2)$

# Review fungsi logaritma

## Pratinjau fungsi logaritma dan eksponensial

$$\log_b a = c \Leftrightarrow b^c = a$$

- $a > 0$  adalah “pangkat logaritma”
- $b > 0$  adalah “basis logaritma”
- $c$  adalah “hasil logaritma”

**Catatan.** Jika basis  $b = 2$ , maka disebut **logaritma biner** (*binary logarithm*). Dalam hal ini, basisnya seringkali tidak dituliskan.

# Notasi $\mathcal{O}$ pada fungsi logaritma

Dalam Ilmu Komputer, kita biasanya menggunakan kompleksitas logaritma **base-two** secara default. Mengapa?

# Notasi $\mathcal{O}$ pada fungsi logaritma

Dalam Ilmu Komputer, kita biasanya menggunakan kompleksitas logaritma **base-two** secara default. Mengapa?

- Umum untuk bekerja dengan bilangan biner atau membagi data input menjadi dua
- Dalam notasi Big-O (pertumbuhan batas atas), semua logaritma adalah *setara dengan asimtotik* (satu-satunya perbedaan adalah faktor konstanta perkalian)
- Jadi, kita tidak menentukan basisnya, dan hanya menuliskannya sebagai  $\mathcal{O}(\log n)$



# Notasi $\mathcal{O}$ pada fungsi logaritma

## Sifat-sifat fungsi logaritma

- $\log_b 1 = 0$  untuk setiap  $b \geq 0$
- **Penggantian basis:**  $\log_b a = \frac{\log_p a}{\log_p b}$
- **Penjumlahan:**  $\log_p m + \log_p n = \log_p mn$
- **Pengurangan:**  $\log_p m - \log_p n = \log_p \frac{m}{n}$
- **Pangkat:**  $\log_p a^x = x \cdot \log_p a$
- **Invers:**  $\log_p \frac{1}{a} = -\log_p a$
- dsb...

## Contoh notasi $\mathcal{O}$ (3): Fungsi logaritma

### Contoh

*Tunjukkan bahwa  $g(n) = (n + 3) \log(n^2 + 1) + 2n^2$  ada di  $\mathcal{O}(n^2)$*

## Contoh notasi $\mathcal{O}$ (3): Fungsi logaritma

### Contoh

Tunjukkan bahwa  $g(n) = (n + 3) \log(n^2 + 1) + 2n^2$  ada di  $\mathcal{O}(n^2)$

### Solusi:

Perhatikan bahwa:

$$\log(n^2 + 1) \leq \log(2n^2) = \log 2 + \log n^2 \leq 2 \log n^2 = 4 \log n.$$

Jadi,  $\log(n^2 + 1) \in \mathcal{O}(\log n)$ .

Karena  $n + 3 \in \mathcal{O}(n)$ , maka

$$(n + 3) \log(n^2 + 1) \in \mathcal{O}(n) \cdot \mathcal{O}(\log n) \in \mathcal{O}(n \log n).$$

Karena  $2n^2 \in \mathcal{O}(n^2)$ , dan  $\max(n \log n, n^2) = n^2$ , maka  $g(n) \in \mathcal{O}(n^2)$ .

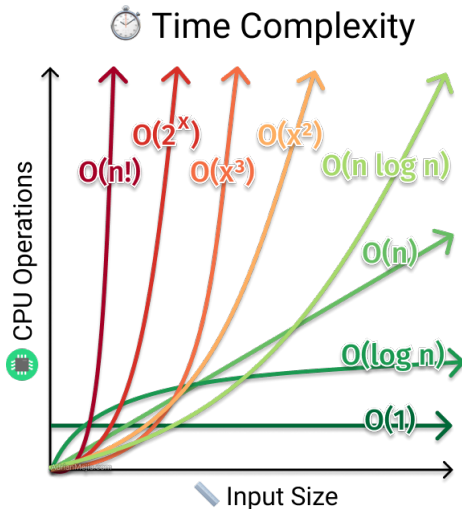
## Bagian 4: Klasifikasi algoritma berdasarkan kompleksitas waktu terburuk

# Klasifikasi algoritma berdasarkan kompleksitas waktu terburuk

Complexity	Class
$\mathcal{O}(1)$	constant
$\mathcal{O}(\log n)$	logarithmic
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	quasilinear /linearithmic
$\mathcal{O}(n^2)$	square
$\mathcal{O}(n^3)$	cubic
$\mathcal{O}(n^k), k \geq 2$	polynomial
$\mathcal{O}(2^n)$	exponential
$\mathcal{O}(n!)$	factorial

$$\underbrace{\mathcal{O}(1) < \mathcal{O}(\log n) < \mathcal{O}(n) < \mathcal{O}(n \log n) < \mathcal{O}(n^2) < \mathcal{O}(n^3) < \dots < \mathcal{O}(2^n) < \mathcal{O}(n!)}_{\text{polynomial algorithms}} \quad \underbrace{\hspace{10em}}_{\text{exponential algorithms}}$$

# Klasifikasi algoritma berdasarkan kompleksitas waktu terburuk



## Bagian 5: Penghitungan banyaknya operasi pada algoritma

# Kalkulasi jumlah operasi algoritma: Operasi dasar

- 1 **Penugasan nilai** (*perbandingan, operasi aritmatika, baca, tulis*) membutuhkan  $\mathcal{O}(1)$
- 2 **Mengakses** elemen array, atau memilih bidang dari catatan memerlukan  $\mathcal{O}(1)$

## Contoh

- ▶  $read(x) \rightarrow \mathcal{O}(1)$
- ▶  $x : x + a[k] \rightarrow \mathcal{O}(1)$
- ▶  $print(x) \rightarrow \mathcal{O}(1)$



## Kalkulasi jumlah operasi algoritma: If-Else

- ③ **If-Else condition:** IF C THEN A1 ELSE A2 membutuhkan waktu:  
 $T_C + \max(T_{O1}, T_{O2})$

### Contoh

```
1: read(x)
2: if x mod 2 = 0 then
3:   x := x + 1
4:   print("Even")
5: else
6:   print("Odd")
7: end if
```

*Kompleksitas waktu asimtotik:*

$$\mathcal{O}(1) + \mathcal{O}(1) + \max(\mathcal{O}(1) + \mathcal{O}(1), \mathcal{O}(1)) \in \mathcal{O}(1)$$

# Kalkulasi jumlah operasi algoritma: For loop

- 4 **For loop:** kompleksitas waktu adalah jumlah iterasi dikalikan dengan kompleksitas waktu *body loop* (yaitu *pernyataan loop*)

## Contoh (Single for loop)

```
1: for  $i = 1$  to  $n$  do  
2:    $\text{sum} := \text{sum} + a[1]$   
3: end for
```

*Kompleksitas waktu asimtotik:  $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$*

# Kalkulasi jumlah operasi algoritma: Loop bersarang

## Contoh (Two nested for loops with one instruction)

```
1: for  $i = 1$  to  $n$  do  
2:   for  $j = 1$  to  $n$  do  
3:      $a[i,j] := i + j$   
4:   end for  
5: end for
```

*Kompleksitas waktu asimtotik:  $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$*

# Kalkulasi jumlah operasi algoritma: Loop bersarang

## Contoh (Two nested for loops with two instructions)

```
1: for  $i = 1$  to  $n$  do  
2:   for  $j = 1$  to  $i$  do  
3:      $a := a + 1$   
4:      $b := b - 1$   
5:   end for  
6: end for
```

Loop luar dieksekusi  $n$  kali, dan loop dalam dieksekusi  $i$  kali untuk setiap  $j$ . Jumlah iterasi:  $1 + 2 + \dots + n = \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$ .

Perulangan pada body membutuhkan waktu  $\mathcal{O}(1)$ .

Asymptotic time complexity:  $\mathcal{O}(n^2)$

# Kalkulasi jumlah operasi algoritma: While loop

⑤ **While loop:** WHILE C DO A; and REPEAT A UNTIL C.

Time complexity = # iterations  $\times T_{\text{body}}$

Contoh (Single loop with  $n - 1$  iterations)

```
1:  $i := 2$   
2: while  $i \leq n$  do  
3:   sum := sum +  $a[i]$   
4:    $i := i + 1$   
5: end while
```

*Kompleksitas waktu asimtotik:*

$$\mathcal{O}(1) + (n - 1)(\mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(1)) = \mathcal{O}(1) + \mathcal{O}(n - 1) \in \mathcal{O}(n)$$

# Kalkulasi jumlah operasi algoritma: Infinite loop

## Contoh (Infinite loop)

```
1:  $x := 0$   
2: while  $x < 5$  do  
3:    $x := 1$   
4:    $x := x + 1$   
5: end while
```

*Dalam situasi ini,  $x$  tidak akan pernah lebih besar dari 5, karena pada awal perulangan while,  $x$  diberi nilai 1, sehingga perulangan akan selalu berakhir dengan 2 dan perulangan tidak akan pernah terputus.*

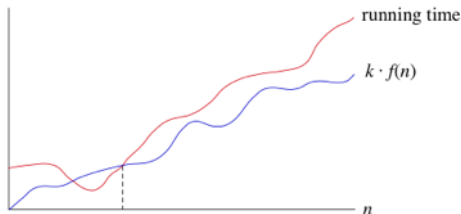
## Notasi $\Omega$ : Batas-bawah asimptotik

Kita juga dapat mengatakan bahwa suatu algoritma membutuhkan *setidaknya sejumlah waktu tertentu*, tanpa memberikan batas atas.

Big-Omega ( $\Omega(\cdot)$ ) notation

### Definisi

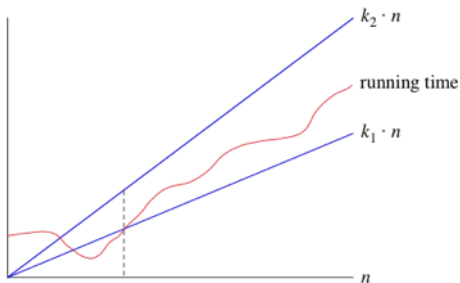
$g(n) \in \Omega(f(n))$  jika  $\exists k > 0$  dan  $n_0$  sedemikian sehingga  $g(n) \geq k \cdot f(n)$ ,  $\forall n \geq n_0$ .



# Notasi $\Theta$ : Batas-ketat asimptotik

## Definisi

$g(n) \in \Theta(f(n))$  jika  $\exists k_1, k_2 > 0$  dan  $n_0$  sedemikian sehingga  $k_1 \cdot f(n) \leq g(n) \leq k_2 \cdot f(n)$ ,  $\forall n \geq n_0$ .





*end of slide...*