

## 2 - Analisis Kompleksitas Komputasional

[KOMS120403]

Desain dan Analisis Algoritma (2022/2023)

Dewi Sintiar

Prodi S1 Ilmu Komputer  
Universitas Pendidikan Ganesha

Week 2 (March 2023)

# Daftar isi

- Review algoritma fpb
- Model kompleksitas komputasi
- Notasi asimtotik dan *order of magnitude*
- Notasi Big-O: 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
- Latihan soal

# Tujuan pembelajaran

Anda diharapkan mampu untuk:

- 1 Menjelaskan konsep kompleksitas algoritma
- 2 Menjelaskan perbedaan *worst-case*, *best-case*, dan *average-case* algoritma
- 3 Menggunakan notasi Big-O, Big-Omega, dan Big-Theta dalam menuliskan kompleksitas
- 4 Menghitung kompleksitas algoritma
- 5 Mengklasifikasikan algoritma berdasarkan kelas kompleksitasnya

# Bagian 1: Contoh motivasi

# Perhatikan lagi algoritma penghitungan fpb dua integer

## 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 banyaknya operasi yang dilakukan untuk menjalankan algoritma tersebut!

# Jawaban

Coba bandingkan algoritma tersebut dengan algoritma berikut (yang disebut dengan *algoritma Euclid untuk fpb*)

# 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
```

---

### Questions:

- Mengapa algoritma tersebut berakhir (tidak mengalami *infinite looping*)? (lihat Teorema Lame <https://www.cut-the-knot.org/blue/LamesTheorem.shtml>)
- Tentukan banyaknya operasi yang terjadi, dan bandingkan dengan algoritma penghitungan fpb yang sebelumnya!

## 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 komputasional* dari suatu algoritma.

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

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

*Bagaimana menghitung kompleksitas suatu algoritma?*

# Bagaimana pengaruh kompleksitas 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**. Diketahui bahwa:

- 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 masing-masing komputer (superkomputer dan PC)!

## Bagaimana pengaruh kompleksitas algoritma?

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

# Bagaimana pengaruh kompleksitas algoritma?

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

**Solusi:** *running time masing-masing komputer*

$$\underline{n = 10^6}$$

- **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 mempengaruhi kompleksitas komputasi?

**Running time** bergantung pada banyak hal seperti *hardware*, *OS*, *processors*, *programming language* dan *compiler*, dll.

Tapi kita **tidak** memperhitungkan faktor-faktor ini saat menganalisis **kompleksitas** algoritma.

## Beberapa catatan dalam mempelajari kompleksitas algoritma:

- Fokus kita pada perkuliahan ini adalah pada **kompleksitas waktu**.
- Kita berasumsi bahwa mesin kita hanya menggunakan satu prosesor (yaitu *generic one-processor*). Jadi hanya satu instruksi yang dieksekusi dalam suatu waktu tertentu.
- Kompleksitas waktu dihitung berdasarkan **banyaknya operasi/instruksi**
- *Running time* dari suatu algoritma dihitung sebagai fungsi dari 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:**  $T(n) = (n + 2) + n + 1 = 2n + 3$  operations.

## Bagian 3: Tiga model kompleksitas algoritma

## Tiga macam pengukuran penggunaan sumber daya

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

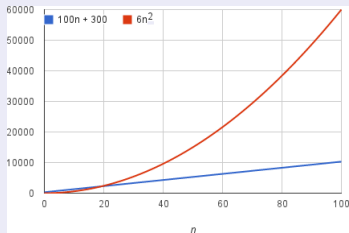
# Notasi asimtotik dan derajat besarannya (1)

- Ingatlah bahwa **kompleksitas waktu** suatu algoritma diukur sebagai **fungsi dari ukuran inputnya**.
- **Rate of growth** dari fungsi kompleksitas mengukur seberapa cepat suatu fungsi meningkat dengan peningkatan ukuran input. Secara **asimtotik** berarti fungsi itu penting *hanya untuk nilai  $n$  yang besar*.
- **Order of magnitude** dari fungsi menjelaskan bagian dari fungsi yang meningkat paling cepat saat nilai  $n$  meningkat.

## Notasi asimtotik dan derajat besarannya (2)

### Contoh

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

## 3.1. Big-O

# Review fungsi logaritma

Sebelum mempelajari notasi Big-O, silahkan tinjau kembali definisi dan sifat-sifat fungsi logaritma berikut.

## 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.



# Review fungsi logaritma

*Silahkan baca lagi buku catatan / rujukan tentang fungsi logaritma*

## Beberapa 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...

## 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* (batas atas) pada sumber daya yang dibutuhkan oleh algoritma.

### Mengapa mempelajari kompleksitas kasus terburuk?

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

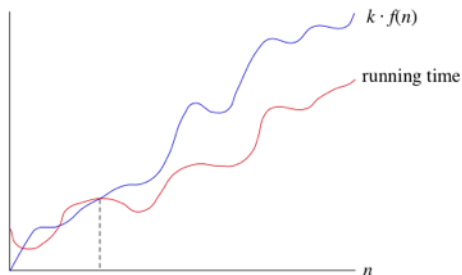
$\mathcal{O}$  adalah notasi matematika yang menjelaskan perilaku pembatas fungsi ketika argumen cenderung ke nilai tertentu atau tak terhingga.

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

## Definisi (Notasi Big-O $\mathcal{O}(\cdot)$ )

$g(n) \in \mathcal{O}(f(n))$  if  $\exists k > 0$  dan  $n_0$  sedemikian sehingga

$$g(n) \leq k \cdot f(n), \quad \forall n \geq n_0$$



## 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 dengan derajat  $m$ , 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:** kerjakan sebagai latihan!

# 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  $n^2 \in \mathcal{O}(n^2)$

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

Dalam Ilmu Komputer, kita biasanya menggunakan kompleksitas logaritma **basis-dua** secara standar (*default*). Mengapa?

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

Dalam Ilmu Komputer, kita biasanya menggunakan kompleksitas logaritma **basis-dua** secara standar (*default*). Mengapa?

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

## 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

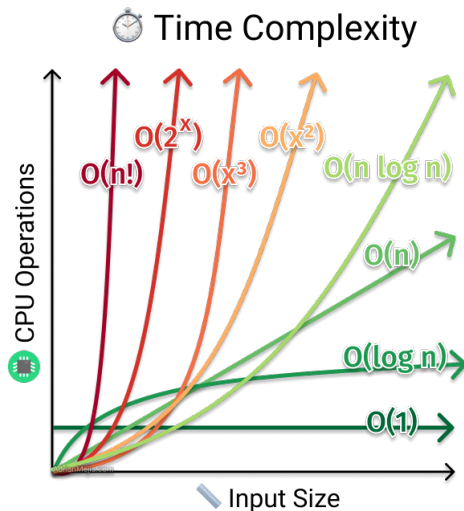
# Klasifikasi algoritma berdasarkan kompleksitas waktu terburuk

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

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



# Klasifikasi algoritma berdasarkan kompleksitas waktu terburuk



## Bagian 5: Penghitungan banyaknya operasi pada algoritma

# Menghitung jumlah operasi algoritma: **operasi dasar**

- 1 **Operasi assign (deklarasi)** (*perbandingan, operasi aritmatika, baca, tulis*) membutuhkan  $\mathcal{O}(1)$
- 2 **Mengakses** elemen array, atau mengambil nilai yang tersimpan memerlukan  $\mathcal{O}(1)$

## Contoh

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

## Menghitung 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 (Operasi dasar)

```
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:*

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

## Menghitung 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:   sum := sum + a[1]  
3: end for
```

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

## Menghitung 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:  $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$*

# Menghitung 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  $\mathcal{O}(1)$ .*

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

## Menghitung 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:*

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



## Menghitung 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.

## Operasi pada *procedure* dan *function*

- 6 Untuk sebuah prosedur atau fungsi yang dipanggil pada suatu algoritma, kompleksitas waktunya adalah  $\mathcal{O}(1)$ .
- 7 Namun, pada penghitungan kompleksitas waktu secara keseluruhan, kita tetap memperhitungkan kompleksitas waktu prosedur atau fungsi tersebut terhadap besarnya input  $n$ .

## 3.2. Big-Omega

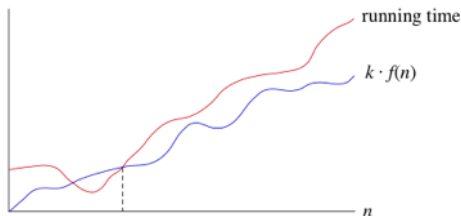
## Notasi $\Omega$ : Batas-bawah (*lower-bound*) asimtotik

Kita juga dapat mengatakan bahwa suatu algoritma membutuhkan *minimal sejumlah waktu tertentu*. Hal ini biasanya dilakukan dengan memberikan *batas bawah* (*lower bound*).

### Definisi (Notasi Big-Omega $\Omega(\cdot)$ )

$g(n) \in \Omega(f(n))$  jika  $\exists k > 0$  dan  $n_0$  sedemikian sehingga

$$g(n) \geq k \cdot f(n), \quad \forall n \geq n_0$$



## 3.3. Big-Theta

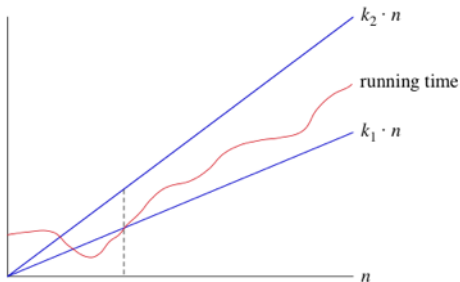
## Notasi $\Theta$ : Batas-ketat (*tight-bound*) asimtotik

Batas ketat dari suatu fungsi berarti suatu fungsi lain yang membatasi fungsi tersebut dari atas dan bawah. Secara formal, didefinisikan sebagai berikut:

### Definisi (Notasi Big-Theta $\Theta(\cdot)$ )

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

$$k_1 \cdot f_n \leq g(n) \leq k_2 \cdot f(n), \quad \forall n \geq n_0$$



# Contoh penghitungan kompleksitas waktu terbaik, terburuk, dan rata-rata

## Contoh:

---

### Algorithm 4 Sequential search

---

```
1: procedure SEQSEARCH( $A[1..n]$ ,  $x$ )
2:   found  $\leftarrow$  False
3:    $i \leftarrow 1$ 
4:   while (not found) and ( $i \leq N$ ) do
5:     if ( $A[i] = x$ ) then found  $\leftarrow$  True
6:     else  $i \leftarrow i + 1$ 
7:     end if
8:   end while
9:   if (found) then index  $\leftarrow i$ 
10:  else index  $\leftarrow 0$ 
11:  end if
12: end procedure
```

---

# Contoh penghitungan kompleksitas waktu terbaik, terburuk, dan rata-rata

---

**Algorithm 2** Sequential search

---

```
1: procedure SEQSEARCH( $T[1..n]$ )
2:   found  $\leftarrow$  False
3:    $i \leftarrow 1$ 
4:   while (not found) and ( $i \leq N$ ) do
5:     if ( $T[i] = x$ ) then found  $\leftarrow$  True
6:     else  $i \leftarrow i + 1$ 
7:     end if
8:   end while
9:   if (found) then index  $\leftarrow i$ 
10:  else index  $\leftarrow 0$ 
11:  end if
12: end procedure
```

---

- Kasus terbaik adalah ketika  $x = A[1]$ , yaitu

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

Ini dapat juga diartikan bahwa algoritma memiliki kompleksitas waktu  $\Omega(1)$ .

- Kasus terburuk adalah ketika  $x = A[n]$  atau  $x$  tidak ditemukan, yaitu

$$T_{\max}(n) = n$$

Ini dapat juga diartikan bahwa algoritma memiliki kompleksitas waktu  $\mathcal{O}(n)$ .



# Contoh penghitungan kompleksitas waktu terbaik, terburuk, dan rata-rata

---

**Algorithm 2** Sequential search

---

```
1: procedure SEQSEARCH( $T[1..n]$ )
2:   found  $\leftarrow$  False
3:    $i \leftarrow 1$ 
4:   while (not found) and ( $i \leq N$ ) do
5:     if ( $T[i] = x$ ) then found  $\leftarrow$  True
6:     else  $i \leftarrow i + 1$ 
7:   end if
8: end while
9: if (found) then index  $\leftarrow i$ 
10: else index  $\leftarrow 0$ 
11: end if
12: end procedure
```

---

- Kasus rata-rata dapat dihitung sebagai berikut:  
Jika  $x = A[j]$ , kompleksitas waktunya adalah  $T(j) = j$ . Jadi:

$$T_{\text{avg}}(n) = \frac{1}{n} \sum_{j=1}^n T(j) = \frac{1+2+\dots+n}{n} = \frac{1/2 \cdot n(n+1)}{n} = \frac{n+1}{2}$$

Apa yang dapat Anda simpulkan? (1)

## Apa yang dapat Anda simpulkan? (1)

Tiga macam penghitungan kompleksitas waktu.

- Kompleksitas waktu **terbaik** (*best case*)

Kasus terbaik adalah fungsi yang melakukan jumlah langkah minimum pada input data  $n$  elemen.

- Kompleksitas waktu **terburuk** (*worst case*)

Kasus terburuk adalah fungsi yang melakukan jumlah langkah maksimum pada data masukan berukuran  $n$ .

- Kompleksitas waktu **rata-rata** (*average case*)

Kasus rata-rata adalah fungsi yang melakukan jumlah langkah rata-rata pada data input  $n$  elemen.

Apa yang dapat Anda simpulkan? (2)

## Apa yang dapat Anda simpulkan? (2)

- Jika  $g(n)$  adalah  $\mathcal{O}(f(n))$ , ini berarti bahwa  $g(n)$  tumbuh secara asimtotik tidak lebih cepat dari  $f(n)$ .
- Jika  $g(n)$  adalah  $\Omega(f(n))$ , ini berarti bahwa  $g(n)$  tumbuh secara asimtotik tidak lebih lambat dari  $f(n)$ .
- Jika  $g(n)$  adalah  $\Theta(f(n))$ , ini berarti bahwa  $g(n)$  tumbuh secara asimtotik pada kecepatan yang sama dengan  $f(n)$ .

### Catatan

*Kompleksitas waktu terbaik, terburuk, dan rata-rata **tidak sama** dengan penggunaan notasi  $\mathcal{O}$ ,  $\Omega$ , dan  $\Theta$ .*

# Kesimpulan lain?

1 ...

2 ...

3 ...

**Latihan.** Kerjakan Exercise 1 “Kompleksitas”

*end of slide...*