Pelatihan TOKI Tahap 1 2012

Dasar-dasar Analisis Algoritme

Motivasi

- Karena membuat algoritme yang benar saja tidak cukup
- Kita perlu membuat algoritme yang benar **dan** efisien!
- Permasalahan yang dapat diselesaikan secara teoritis, belum tentu dapat diselesaikan secara praktis
- Bagaimana mengukur efisiensi sebuah algoritme??

Contoh: *Sequential Search Algorithm*

Pernyataan Masalah:

Mencari lokasi/ indeks dari x dalam list. Algoritma tsb. membandingkan x dengan setiap entri satu persatu sampai ketemu atau tidak ada lagi yang dapat dibandingkan Jika x tidak ada dalam list, maka jawaban-nya = 0

3

Algoritma *Sequential Search*

Input:
L : list dengan n elemen, berindeks 1 … n
x : item yang akan dicari dalam L

Output:
Indeks, yaitu lokasi x dalam L
0, jika x tidak ditemukan

4

Algoritma *Sequential Search*

```
[1]  indeks := 1;
[2]  while indeks ≤ n and L[indeks] ≠ x do
[3]        indeks := indeks + 1
[4]  end {while} a
[5]  if indeks > n then indeks :=0
```

5

Apakah *problem statement* sudah benar?

- Bagaimana bila x muncul lebih dari sekali?
- Berapa nilai n yang diperbolehkan?

Revisi Pernyataan Masalah:

Diberikan suatu list L berisi n item ($n \geq 0$) dan nilai x, maka algoritma akan berhenti dengan nilai indeks = indeks dari kemunculan pertama x di L, jika x ada di L, atau = 0, bila x tidak ada di L

6

Amount of work done

Banyaknya pekerjaan yang dilakukan oleh algoritma

Perlu dicari ukuran yang dapat:

- Mengukur efisiensi dari metode
- Cukup *precise*
- Cukup umum

Bagaimana dengan *execution time*?

- Tergantung pada tipe komputer
- Tergantung pada bahasa pemrograman
- Tergantung pada gaya programmer

7

Diukur dengan banyaknya operasi dasar yang dilakukan oleh algoritma tersebut

Masalah	Operasi dasar
Mencari x dalam list	Pembandingan x dengan setiap entri dalam list
Penggandaan 2 matriks	Perkalian dua bilangan Penjumlahan dua bilangan
Mengurutkan elemen dalam list	Pembandingan antara 2 entri dalam list
Penelusuran *binary tree*	Penelusuran *link*

8

Banyaknya pekerjaan yang dilakukan:

- menggambarkan kompleksitas algoritma
- tergantung pada ukuran input
- tergantung pada tipe input

Masalah	Ukuran input
Mencari x dalam list	Banyaknya entri dalam list
Penggandaan 2 matriks	Ukuran matriks
Mengurutkan elemen dalam list	Banyaknya entri dalam list
Penelusuran *binary tree*	Banyaknya *node* dalam *tree*

9

Worst-case complexity

- Banyaknya operasi dasar yang maksimal
- Batas atas dari banyaknya pekerjaan yang dilakukan oleh algoritma

$$W(n) = maks \{ t(I), I \in Dn \},$$

dengan:

Dn = himpunan dari input yang berukuran n

t(I) = banyaknya operasi dasar yang dilakukan pada input I, I ∈ Dn

10

Average-case complexity

- Banyaknya operasi dasar yang dilakukan secara rata-rata

$$A(n) = \Sigma\, p\,(I)\, t\,(I), \quad I \in Dn,$$

dengan:

Dn = himpunan dari input yang berukuran n

t(I) = banyaknya operasi dasar yang dilakukan pada input I

p(I) = peluang kemunculan input I

11

W(n) dan A(n) *sequential search*

***Worst-case*: W(n) = n**

Kasus terburuk terjadi bila x berada di paling ujung atau x tidak ada dalam list

***Average-case*: (diasumsikan x ada dalam list)**

Andaikan Ii, $1 \le i \le n$, merepresentasikan kasus dimana x berada pada indeks i, p(Ii) adalah peluang x berada di posisi i, serta t(Ii) adalah banyaknya pembandingan yang dilakukan bila x berada di posisi i, maka:
p(Ii) = 1/n, t(Ii) = i.

$$A(n) = \Sigma\, p\,(I)\, t\,(I) = \Sigma\, i/\, n = 1/\, n\, \Sigma\, i = (n+1)/2$$

Secara rata-rata, separuh list akan diperiksa untuk menemukan x.

12

***Average-case*: (x tidak harus ada dalam list)**

Andaikan Ii, $1 \le i \le n$, merepresentasikan kasus dimana x berada pada indeks i, I_{n+1} adalah kasus dimana x tidak berada dalam list, q adalah peluang x ada dalam list, p(Ii) adalah peluang x berada di posisi i, serta t(Ii) adalah banyaknya pembandingan yang dilakukan bila x berada di posisi i, maka: $p(Ii) = q/n$, $t(Ii) = i$, $p(I_{n+1}) = 1-q$, $t(I_{n+1}) = n$.

$$A(n) = \sum p(I)\, t(I) = \sum i\, q/n + (1-q)\, n$$
$$= q/n \sum i + (1-q)\, n = q/n\, n(n+1)/2 + (1-q)\, n$$
$$= q(n+1)/2 + (1-q)\, n$$

Pada kasus dimana x dipastikan ada di list, berarti q = 1 sehingga A(n) = (n+1)/2, seperti sebelumnya.

13

Amount of space used

Banyaknya memori yang digunakan tergantung pada implementasi.

Perlu diperhitungkan hal-hal yang memakan memori:

- instruksi
- konstanta
- variabel
- manipulasi data
- penyimpanan informasi
- representasi data

14

Simplicity/ clarity

- Seringkali cara yang *simple* bukanlah yang efisien
- Namun faktor kesederhanaan tetap diperlukan, oleh karena dapat memudahkan verifikasi kebenaran algoritma, koreksi atau modifikasi
- Pada saat memilih algoritma, perlu diperkirakan waktu yang diperlukan untuk mengoreksi program/ men*debug* program
- Jika suatu program akan digunakan secara berulang-ulang, maka faktor efisiensi harus lebih diperhatikan

15

Optimality

- Suatu algoritma dikatakan optimal pada *worstcase*-nya jika tidak ada algoritma lain yang sekelas yang melakukan lebih sedikit operasi dasar

Bagaimana cara memeriksa optimalitas algoritma?

- Tentukan batas bawah dari jumlah operasi yang diperlukan untuk menyelesaikan masalah
- Sembarang algoritma yang melakukan operasi sejumlah batas bawah tersebut berarti optimal

16

Untuk memeriksa optimalitas algoritma

- Buat algoritma yang efisien (A). Analisis A dan tentukan sebuah fungsi W, sedemikian sehingga A mengerjakan paling banyak W(n) langkah pada *worst-case*nya. Dhi, n adalah ukuran input.
- Untuk beberapa fungsi F, buktikan teori yang menyatakan bahwa sembarang algoritma dalam kelas yang dipelajari, terdapat ukuran input n yang menyebabkan algoritma tersebut melakukan paling sedikit F(n) langkah

Jika W = F, maka algortima A optimal pada *worst-case*nya.

Jika W ≠ F, mungkin terdapat algoritma yang lebih baik atau ada batas bawah yang lebih baik.

17

Contoh:

Masalah mencari elemen terbesar dalam list yang berisi n buah bilangan

Kelas algoritma:

Algoritma yang dapat membandingkan dan menyalin entri tapi tidak melakukan operasi terhadap entri-entri tersebut.

Operasi dasar:

Pembandingan 2 entri dalam list

18

Algoritma *FindMax*

Input : L, list dari bilangan
n ≥ 1 adalah banyaknya entri
Output : max, entri terbesar dalam L

```
[1]  max := L[1];
[2]  for indeks := 2 to n do
[3]       if max < L[indeks] then
[4]            max := L[indeks]
[5]       end
[6]  end
```

19

- Pembandingan 2 dilakukan pada baris-3, dieksekusi sebanyak (n-1) kali. Dengan demikian (n-1) adalah batas atas dari banyaknya pembandingan yang diperlukan untuk menemukan max pada worst-case. Jadi W(n) = n-1

Apakah terdapat algoritma lain yang lebih baik?

Asumsi: semua entri dalam list berbeda, sehingga terdapat (n-1) buah entri yang tidak maksimum. Suatu entri bukan maksimum hanya jika entri tersebut lebih kecil dari sedikitnya sebuah entri yang lain

20

- (n-1) buah entri ini 'kalah' dalam pembandingan yang dilakukan oleh algoritma tsb.
- Padahal dalam setiap pembandingan, hanya ada 1 yang kalah, sehingga sedikitnya (n-1) pembandingan yang harus dikerjakan. Jadi F(n)= n-1 adalah batas bawah jumlah pembandingan yang diperlukan.
- Karena W(n) = F(n), maka algoritma ini optimal

21

Contoh: Masalah Penggandaan Matriks
A=(a_{ij}) dan B=(b_{ij}) adalah matriks yang berukuran nxn dengan entri bilangan nyata. Hitung matriks C = A.B, dgn C=(c_{ij})

Kelas algoritma:

Algoritma yang dapat melakukan perkalian, pembagian, penjumlahan dan pengurangan pada entri-entri matriks dan hasil antaranya

Operasi dasar:

Perkalian

22

Berikut ini adalah algoritma untuk menggandakan matriks A dan B (masing-masing berukuran nxn) menghasilkan matriks C.

```
[1]  for  i := 1 to n do
[2]       for  j := 1 to n do
[3]            c_ij := 0;
[4]            for k := 1 to n do   c_ij := c_ij + a_ik b_kj   end
[5]       end
[6]  end
```

- Tentukan W(n) dan A(n) dari algoritma di atas.

23

Batas atas : $W(n) = n^3$, $A(n) = n^3$

Batas bawah : telah dibuktikan bahwa $F(n)=n^2$

Kesimpulan: tidak dapat disimpulkan algoritma perkalian matriks adalah optimal

Fakta:

Terdapat algoritma yang melakukan $n^{2.81}$ penggandaan, sehingga algoritma yang pertama terbukti tidak optimal

24

Bagaimana menghitung kompleksitas algoritma?

Contoh : Algoritma untuk menghitung $\sum i^3$

```
begin
[1]  temp := 0;                          1
[2]  for  i := 1 to n do                 3n
[3]       temp := temp + (i*i*i);        4n
[4]  sum := temp;                        1
end                                  total: 7n+2
```

25

- Pada contoh di atas, diperoleh T(n) = 7n+2 ≈ 7n
- Dhi, konstanta 2 dapat diabaikan untuk n yang cukup besar. Lebih lanjut konstanta 7 juga dapat diabaikan
- Metode yang dapat digunakan untuk membandingkan atau mengklasifikasikan fungsi-fungsi dengan mengabaikan faktor konstanta dan input yang kecil adalah laju pertumbuhan asimtot/ laju pertumbuhan fungsi

26

Faktor konstanta dpt diabaikan

n	$3n^3$ nanodtk	19500000n nanodtk
10	3 mikrodtk	200 milidtk
100	3 milidtk	2 detik
1000	3 detik	20 detik
2500	50 detik	50 detik
10000	49 menit	3,2 menit
1000000	95 tahun	5,4 jam

27

Aturan Umum

- Untuk *loop* :
  RT maks = RT dari *statement* dalam *loop* dikalikan dengan banyaknya iterasi

- Nested *loop* :
  RT dari *statement* dikalikan hasil perkalian banyaknya iterasi pada semua *loop* terkait

contoh:
```
for  i := 1 to m do
     for  j := 1 to n do
          k := k+1;
```
RT = $O(n^2)$

28

- Statement yang berurutan; diambil yang terbesar

contoh:
```
for i := 1 to n do
    a[i] := 0;
for i := 1 to n
    for j := 1 to n do
        a[i] := a[j] + i + j;
```

RT = $O(n^3)$

29

Insertion_Sort(A)

1. for j := 2 to length[A] do
2. key := A[j]
   {memasukkan A[j] ke dalam array A[1…j-1] yang sudah diurutkan}
3. i := j-1
4. while i > 0 dan A[i] > key do
5. A[i+1] := A[i]
6. i := i-1
7. A[i+1] := key

30

Berapa kompleksitasnya?

	cost	time
for j := 2 to length[A] do	c1	n
key := A[j]	c2	n -1
{memasukkan A[j] ke dalam array A[1…j-1] yang sudah diurutkan}		
i := j-1	c3	n -1
while i > 0 dan A[i] > key do	c4	∑ tj
A[i+1] := A[i]	c5	∑ (tj-1)
i := i-1	c6	∑ (tj-1)
A[i+1] := key	c7	n - 1

tj adalah banyaknya test pada *while loop* yang dikerjakan

31

$$T(n) = c1(n) + c2(n-1) + c3(n-1) + c4\ (\sum tj) + c5\ (\sum(tj-1)) + c6\ (\sum(tj-1)) + c7(n-1)$$

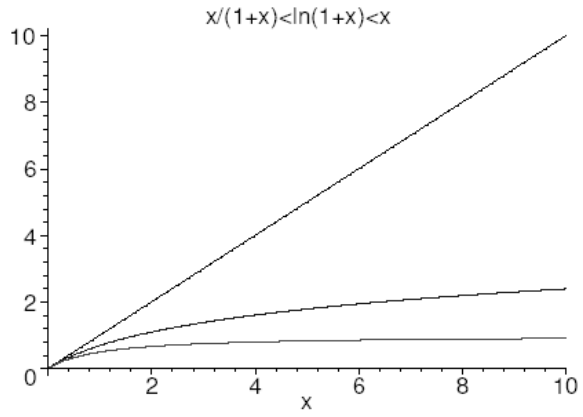
$$T(n) = c1(n) + c2(n-1) + c3(n-1) + c4\ (n(n+1)/2 - 1) + c5\ (n(n-1)/2) + c6\ (n(n-1)/2) + c7(n-1)$$

$T(n) = A\ n^2 + B\ n + C$

32

Laju pertumbuhan fungsi :

- Menggambarkan perilaku fungsi pada nilai variabel bebas yang sangat besar



33

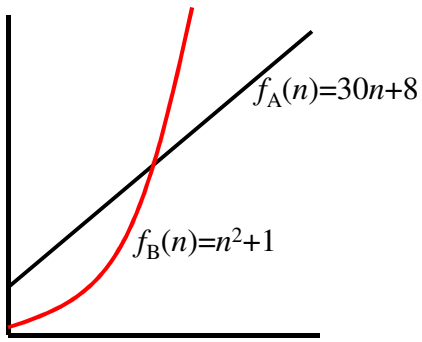
Motivasi

- Andaikan perlu membuat sebuah algoritme yang menyelesaikan suatu permasalahan dengan data tertentu
- Bila Program A memerlukan $f_A(n)=30n+8$ mikrodetik utk memroses, sementara program B perlu $f_B(n)=n^2+1$ mikrodetik utk memroses n buah data.
- Program mana yang akan kita pilih, dengan pertimbangan bahwa nilai input n dapat mencapai jutaan?

34

Motivasi (lanjutan)

Pada grafik terlihat, semakin ke kanan, fungsi yang tumbuh lebih cepat pada akhirnya selalu menjadi yang terbesar



35

Manfaat

- Dapat digunakan untuk membandingkan efisiensi dari beberapa algoritma, dengan cara mengamati pertumbuhan fungsi yang menggambarkan perilaku masing-masing algoritma tersebut untuk ukuran input yang sangat besar -> *asymptotic analysis*

36

Notasi Asimtotik

Terdapat 5 simbol masing-masing untuk kelas yang berbeda :

- O (*Big Oh*)
- Ω (*Big OMEGA*)
- Θ (*Big THETA*)
- o (*Little Oh*)
- ω (*Little OMEGA*)

37

O (Big Oh)

- Contoh berikut memberikan gambaran lebih jelas bagaimana pertumbuhan suatu fungsi lebih cepat dibandingkan pertumbuhan fungsi lainnya.

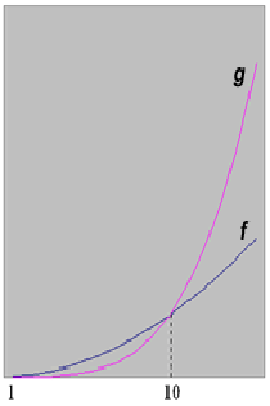
  Contoh ini selanjutnya digunakan untuk mengenalkan konsep big-Oh dan konsep-konsep notasi asimtotik lainnya.

38

Dari tabel dan gambar terlihat bahwa *g(n)* tumbuh lebih cepat dibanding *f(n) ketika n* > 10. Dhi, dikatakan *f* adalah big-Oh dari *g* dan dituliskan *f* = *O(g)*

$f(n) = 100\ n^2$,
$g(n) = n^4$,

n	*f(n)*	*g(n)*
10	10,000	10,000
50	250,000	6,250,000
100	1,000,000	100,000,000
150	2,250,000	506,250,000



39

Definisi (big-oh):

Andaikan ***f*** and ***g*** adalah fungsi-fungsi yang memetakan himpunan bilangan bulat ke himpunan bilangan nyata, maka:

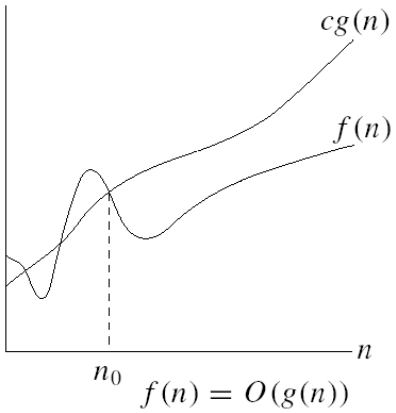
***f* (*x*)** adalah ***O* (*g* (*x*))** , atau
***f* (*x*)** adalah **big-oh** dari ***g* (*x*)** ,
jika hanya jika terdapat konstanta ***C*** dan n_0 sedemikian sehingga:

$| f(x) | \leq C | g(x) |$ ketika $x > n_0$

40

Cormen:

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



41

Contoh:

- Tunjukkan bahwa 30*n*+8 adalah O(*n*).
  - Tunjukkan $\exists\, c, n_0 : \forall n > n_0 : 30n+8 \le cn$.
    - Ambil $c = 31$, $n_0 = 8$.
      Asumsikan $n > n_0 = 8$, maka $cn = 31n = 30n + n >$ $30n+8$, sehingga $30n+8 < cn$.
- Tunjukkan bahwa n^2+1 adalah O(n^2).
  - Tunjukkan $\exists c, n_0 : \forall n > n_0 : n^2+1 \le cn^2$.
    - Ambil $c=2$, $n_0 = 1$.
      Asumsikan $n > 1$, maka
      $cn^2 = 2n^2 = n^2+n^2 > n^2+1$,
      atau $n^2+1 < cn^2$.

42

Sifat-sifat Big-oh:

- Big-oh, sebagai relasi bersifat transitif:
  $f \in O(g) \wedge g \in O(h) \rightarrow f \in O(h)$
- Jika $g \in O(f)$ dan $h \in O(f)$, maka $g+h \in O(f)$
- $\forall c > 0$, $O(cf)=O(f+c)=O(f-c)=O(f)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \rightarrow$
  - $f_1 f_2 \in O(g_1 g_2)$
  - $f_1+f_2 \in O(g_1+g_2)$
    $= O(\max(g_1,g_2))$
    $= O(g_1)$ jika $g_2 \in O(g_1)$

43

- $\forall f,g$ & konstanta $a,b \in \mathbf{R}$, dengan $b \geq 0$,
  - $af = O(f)$; (*e.g.* $3x^2 = O(x^2)$)
  - $f + O(f) = O(f)$; (*e.g.* $x^2+x = O(x^2)$)
- Jika $f=\Omega(1)$ (*sedikitnya orde* 1), maka:
  - $|f|^{1-b} = O(f)$; (*e.g.* $x^{-1} = O(x)$)
  - $(\log_b |f|)^a = O(f)$. (*e.g.* $\log x = O(x)$)
  - $g=O(fg)$ (*e.g.* $x = O(x \log x)$)
  - $fg \neq O(g)$ (*e.g.* $x \log x \neq O(x)$)
  - $a=O(f)$ (*e.g.* $3 = O(x)$)

44

Definisi (big-omega):

Andaikan ***f*** and ***g*** adalah fungsi-fungsi yang memetakan himpunan bilangan bulat ke himpunan bilangan nyata, maka:

***f* (*x*)** adalah ***Ω* (*g* (*x*))** , atau

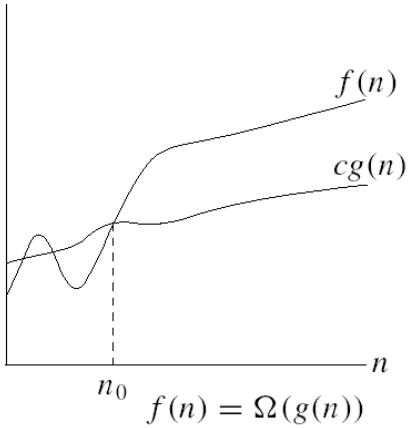
***f* (*x*)** adalah **big-omega** dari ***g* (*x*)**,

jika hanya jika terdapat konstanta ***C*** dan n_0 sedemikian sehingga:

| *f* (*x*) |≥ C | *g* (*x*) | ketika $x > n_0$

45

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



$f(n) = \Omega(g(n))$

46

Definisi (big-theta):

Andaikan ***f*** and ***g*** adalah fungsi-fungsi yang memetakan himpunan bilangan bulat ke himpunan bilangan nyata, maka:

***f* (*x*)** adalah ***ϑ* (*g* (*x*))**, atau

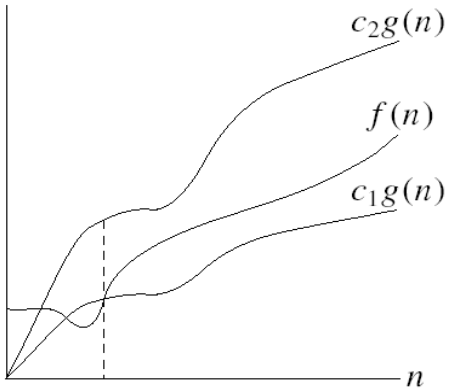
***f* (*x*)** adalah **big-theta** dari ***g* (*x*)**,

jika hanya jika

***f* (*x*)** adalah ***Ω* (*g* (*x*))** dan
***f* (*x*)** adalah ***O* (*g* (*x*))**

47

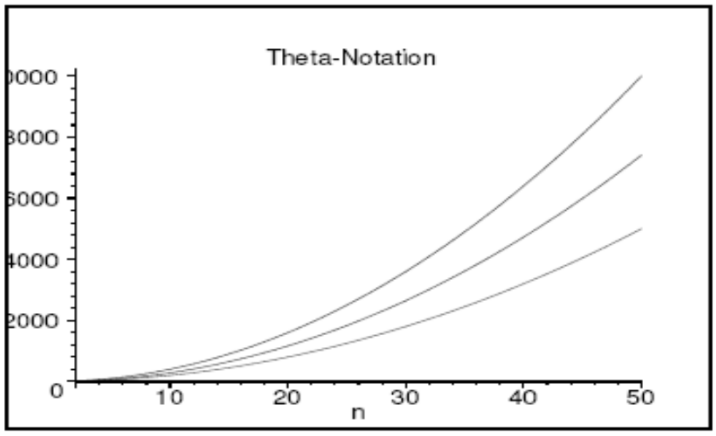
$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.[1]



$f(n) = \Theta(g(n))$

48

$$2n^2 < 3n^2 - \sqrt{\pi}n < 4n^2 \quad \Rightarrow \quad 3n^2 - \sqrt{\pi}n = \Theta(n^2).$$



49

o (*Little Oh*)

$$o(g) = \{f\,;\, \forall c{>}0\ \exists n_0\ \forall x{>}\ n_0 : |f(x)| < |cg(x)|\}$$

$o(g) \subset O(g) - \Theta(g)$

adalah fungsi fungsi yang mempunyai order yang lebih kecil dari g

$o(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Contoh: $2n = o(n^2)$ **tetapi** $2n^2 \neq o(n^2)$

50

ω (*Little OMEGA*)

$$\omega(g) = \{f\,;\, \forall c{>}0\ \exists n_0\ \forall x{>}\ n_0 : |cg(x)| < |f(x)|\}$$

$$\omega(g) \subset \Omega(g) - \Theta(g)$$

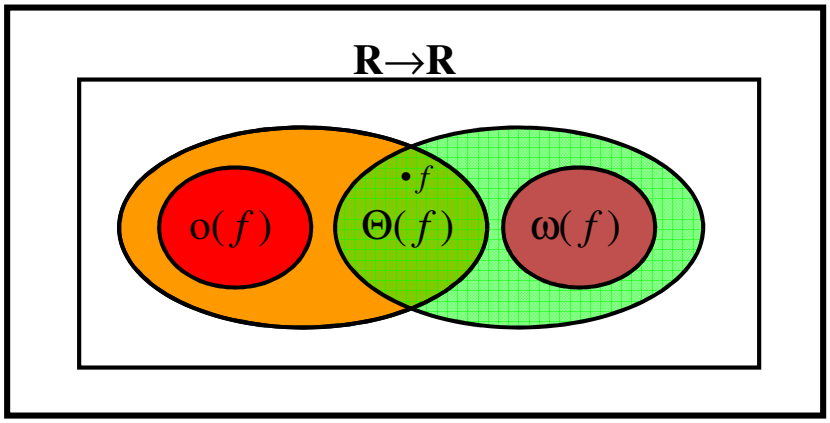
adalah fungsi fungsi yang mempunyai order yang lebih besar dari g

$\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

Contoh: $n^2/2 = \omega(n)$ **tetapi** $n^2/2 \neq \omega(n^2)$

51

Hubungan antar notasi asimtotik



52

Definisi-definisi orde pertumbuhan, $\forall g$:**R**→**R**

- $O(g) :\equiv \{f ; \exists c>0, \exists n_0 \forall x>n_0 |f(x)| < |cg(x)|\}$
- $o(g) :\equiv \{f ; \forall c>0 \exists n_0 \forall x>n_0 |f(x)| < |cg(x)|\}$
- $\Omega(g) :\equiv \{f ; g \in O(f)\}$
- $\omega(g) :\equiv \{f ; g \in o(f)\}$
- $\Theta(g) :\equiv O(g) \cap \Omega(g)$

53

Analogi pada relasi asimtotik

Asymptotic Relation between Functions	Relation between Real Numbers
$f(n) = \Theta(g(n))$	$a = b$
$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Omega(g(n))$	$a \geq b$
$f(n) = o(g(n))$	$a < b$
$f(n) = \omega(g(n))$	$a > b$

54

Perbandingan kompleksitas algoritma

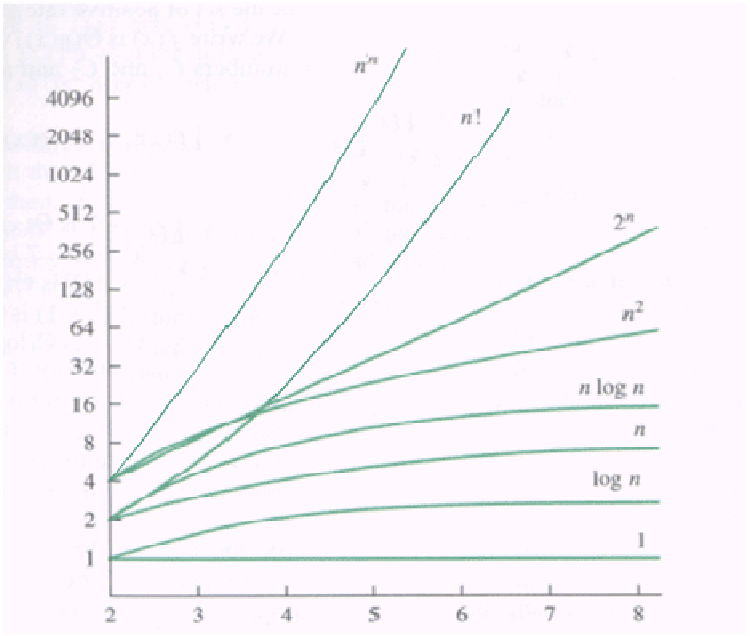
Run time (nanoseconds)		$1.3\ N^3$	$10\ N^2$	$47\ N \log_2 N$	$48\ N$
Time to solve a problem of size	1000	1.3 seconds	10 msec	0.4 msec	0.048 msec
	10,000	22 minutes	1 second	6 msec	0.48 msec
	100,000	15 days	1.7 minutes	78 msec	4.8 msec
	million	41 years	2.8 hours	0.94 seconds	48 msec
	10 million	41 millennia	1.7 weeks	11 seconds	0.48 seconds
Max size problem solved in one	second	920	10,000	1 million	21 million
	minute	3,600	77,000	49 million	1.3 billion
	hour	14,000	600,000	2.4 trillion	76 trillion
	day	41,000	2.9 million	50 trillion	1,800 trillion
N multiplied by 10, time multiplied by		1,000	100	10+	10

55

Perbandingan kompleksitas algoritma

$logn$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

56



57

Problem

1. Lakukan Analisis untuk Algoritme Selection Sort di bawah ini

```
Procedure SelectionSort(integer A[n])
begin
1.    integer i,j,min,tmp;
2.    for(i=0;i<n;i++)
3.    begin
4.      min := i;
5.      for(j=i+1; j<n; j++)
6.      begin
7.        if(A[j] < A[min]) then min := j;
8.      end
9.      tmp := A[i];
10.     A[i] := A[j];
11.     A[j] := tmp;
12.   end
end
```

Problem

2. Lakukan Analisis pada algoritme Binary Search di bawah ini

```
Function BinarySearch(integer A[n], integer key,
                      integer min, integer max)
Begin
1.   If (min<max) then return NOT_FOUND;
2.   Else
3.   Begin
4.    integer mid := midpoint(min, max);
5.    if(A[mid] > key) then return BinarySearch(A, key, min, mid-1)
6.    else if (A[mid] < key) then return BinarySearch(A, key, mid+1,
     max)
7.    else return mid;
End
```

Rekursi

- Ada kalanya kita mengalami kesulitan untuk mendefinisikan suatu objek secara **eksplisit**.
- Dalam kasus ini, mungkin lebih mudah untuk mendefinisikan objek tersebut menggunakan dirinya sendiri. Ini dinamakan sebagai **proses rekursif**
- Kita dapat mendefinisikan barisan, himpunan atau fungsi secara rekursif.

60

Langkah-langkah mendefinisikan himpunan atau fungsi secara rekursif:

1. Langkah basis: Spesifikasi koleksi awal dari anggota
2. Langkah rekursif: Mendefinisikan aturan konstruksi anggota baru dari anggota yang telah diketahui

61

Contoh fungsi yang didefinisikan secara rekursif

f(0) = 3

f(n + 1) = 2f(n) + 3

maka

- **f(0) = 3**
- **f(1) = 2f(0) + 3 = 2·3 + 3 = 9**
- **f(2) = 2f(1) + 3 = 2·9 + 3 = 21**
- **f(3) = 2f(2) + 3 = 2·21 + 3 = 45**
- **f(4) = 2f(3) + 3 = 2·45 + 3 = 93**

62

Bilangan Fibonacci

- **$f_0 = 0, f_1 = 1$**
- **$f_n = f_{n-1} + f_{n-2}$, n=2,3,4,...**

$f_0 = 0$
$f_1 = 1$
$f_2 = f_1 + f_0 = 1 + 0 = 1$
$f_3 = f_2 + f_1 = 1 + 1 = 2$
$f_4 = f_3 + f_2 = 2 + 1 = 3$
$f_5 = f_4 + f_3 = 3 + 2 = 5$
$f_6 = f_5 + f_4 = 5 + 3 = 8$

63

Pertumbuhan populasi kelinci

Sepasang kelinci ditaruh di suatu pulau. Pasangan kelinci ini tidak akan beranak sampai berumur 2 bulan. Setelah berumur 2 bulan, setiap sepasang menghasilkan sepasang yg lain setiap bulannya.
Tentukan relasi *recurrence* dari jumlah pasangan setelah n bulan, bila tidak ada kelinci yg mati.

64

Populasi kelinci:

- bulan 0 : 1 pasang
- bulan 1 : 1 pasang
- bulan 2 : 1 + 1 pasang
- bulan 3 : 2 + 1 pasang
- bulan 4 : 3 + 2 pasang
- ...
- bulan n : bulan(n-1) + bulan(n-2) pasang

65

Solusi:

- Misalkan f_n: jumlah pasangan kelinci setelah n bulan, maka, $f_1 = 1$, $f_2 = 1$.
- Untuk mencari f_n, tambahkan jumlah pasangan pada bulan sebelumnya, f_{n-1}, dengan jumlah pasangan yang baru lahir, f_{n-2}.
- Jadi, $f_n = f_{n-1} + f_{n-2}$.

66

Solusi:

$$F_n = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$$

dengan

$$\phi = \frac{1+\sqrt{5}}{2}$$

disebut *golden ratio*

67

Komputasi rekursif:

```
recFib(n) {
  if (n ≤ 1)
    return n
  else
    return recFib(n-1) + recFib(n-2)
}
```

68

$T(n) = T(n-1) + T(n-2)$

$2T(n-2) \leq T(n-1) + T(n-2) \leq 2T(n-1)$

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

$T(n) = \Omega(2^{n/2})$

69

Komputasi iteratif:

```
IterFib (n) {
  f[0] = 0;
  f[1] = 1;
  for ( i=2 ; i ≤ n ; i++)
    f[i] = f[i-1] + f[i-2];
}
```

$T(n) = O(n)$

70

Relasi rekursif (Recurrence)

Definisi:

•**Relasi *Recurrence*** untuk barisan $\{a_n\}$ adalah *persamaan* yang menyatakan a_n dalam salah satu atau lebih bentuk a_0, a_1, …, a_{n-1} untuk semua n dengan $n \geq n_0$ dimana n_0 bilangan bulat non-negatif.

•Barisan $\{a_n\}$ tersebut dikatakan sebagai ***solusi*** dari relasi *recurrence* ini bila a_n memenuhi relasi *recurrence*.

71

Penyelesaian relasi rekursif

- Metode substitusi:
  dengan cara membuat tebakan terhadap solusinya, kemudian tebakan tsb dibuktikan dengan induksi matematika
- Metode (pohon) rekursif:
  dengan cara mengubah bentuk rekursif menjadi bentuk penjumlahan, kemudian diselesaikan.
- Metode master:
  digunakan utk menyelesaikan pers rekursif dalam bentuk : $T(n) = aT(n/b) + f(n)$

72

Selesaikan T(n) = 2T ([n/2]) + n

Misalnya digunakan tebakan : O(n log n)

T(n) = 2 T([n/2]) + n , T(1) = 1

T(n) ≤ 2(c[n/2]log([n/2])) + n

≤ c.n.lg([n/2]) + n

≤ c.n.lg n - c.n.lg 2 + n

≤ c.n.lg n - c.n + n

≤ c.n.lg n

yang berlaku untuk c ≥ 1

73

Metode iterasi/ rekursif:

Metode iterasi tidak memerlukan tebakan solusinya, tetapi memerlukan manipulasi aljabar yang lebih intensif dari pada metode substitusi. Ide dasarnya adalah dengan mengembangkan bentuk rekursif tersebut serta merepresentasikannya dalam bentuk jumlah. Teknik untuk mengevaluasi bentuk jumlah dapat digunakan untuk mendapatkan nilai batas dari solusinya.

74

Contoh: T(n) = 3T([n/4])+ n
Bila bentuk rekursif diuraikan:

T(n) = n + 3T([n/4])
= n + 3([n/4]) + 3T(n/4[/4]))
= n + 3([n/4]+ 3([n/16]+ 3T(n/64)))
= n + 3n/4+ 9n/16+ 27T(n/64)

Suku ke-i berbentuk $3^i n/4^i$
Oleh karena n berkurang sebesar 4 pada setiap iterasi, maka proses berlangsung sampai $\log_4 N$ langkah

75

T(n) ≤ n + 3n/4 + 9n/16 + 27n/64 + . . .+ $3^{\log_4 N}$ N / N

$$\leq n. \sum_{i=0}^{log_4 N - 1} (3/4)^i + \mathcal{O}(3^{log_4 N})$$

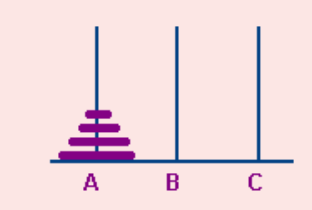
$$\leq n. \sum_{i=0}^{\infty} (3/4)^i + \mathcal{O}(3^{log_4 N})$$

$$\leq 4n + \mathcal{O}(n) \qquad \leq 5n$$
$$= \mathcal{O}(n)$$

76

Menara Hanoi

Merupakan sebuah puzzle populer yang ditemukan oleh seorang matematikawan Perancis Edouard Lucas pada abad 19



Terdapat menara dengan 3 tiang untuk meletakkan sejumlah disk berukuran berbeda. Awalnya semua disk terletak secara terurut pada tiang pertama dengan disk terbesar paling bawah

Aturan: Satu disk dapat dipindahkan setiap waktu dari satu tiang ke tiang lain selama disk tsb tidak berada di atas disk yang lebih kecil.

Tujuan: Memindahkan semua disk ke tiang kedua dengan disk terbesar di urutan paling bawah.

77

- Misalkan H_n: banyaknya langkah yg diperlukan untuk memindahkan n disk dalam masalah menara Hanoi.
- Kita mulai dengan n disk pada tiang 1. Kita dapat memindahkan n-1 disk paling atas dengan mengikuti aturan ke tiang 3 dalam H_{n-1} langkah.
- Kemudian, dengan menggunakan 1 langkah kita bisa memindahkan disk terbesar ke tiang 2.
- Selanjutnya, pindahkan n-1 disk dari tiang 3 ke tiang 2, dengan mengikuti aturan dalam H_{n-1} langkah. Dengan demikian, kita telah memecahkan puzzle dengan banyak langkah:

$$H_n = 2H_{n-1} + 1 \text{ dan } H_1 = 1.$$

78

Untuk mencari solusinya, dilakukan proses iteratif:

$H_n = 2H_{n-1} + 1$

$= 2(2H_{n-2} + 1)+1 = 2^2H_{n-2} + 2 +1$

$= 2^2(2H_{n-3} +1) + 2 +1 = 2^3H_{n-3} + 2^2 + 2 +1$

…

$= 2^{n-1}H_1 + 2^{n-2} + 2^{n-3} + … + 2 +1$

$= 2^{n-1} + 2^{n-2} + 2^{n-3} + … + 2 +1$ **(deret geometri)**

$= 2^n - 1$

Jadi, untuk memindahkan 64 disk diperlukan langkah sebanyak:

$2^{64} - 1 = 18,446,744,073,709,551,615.$

79

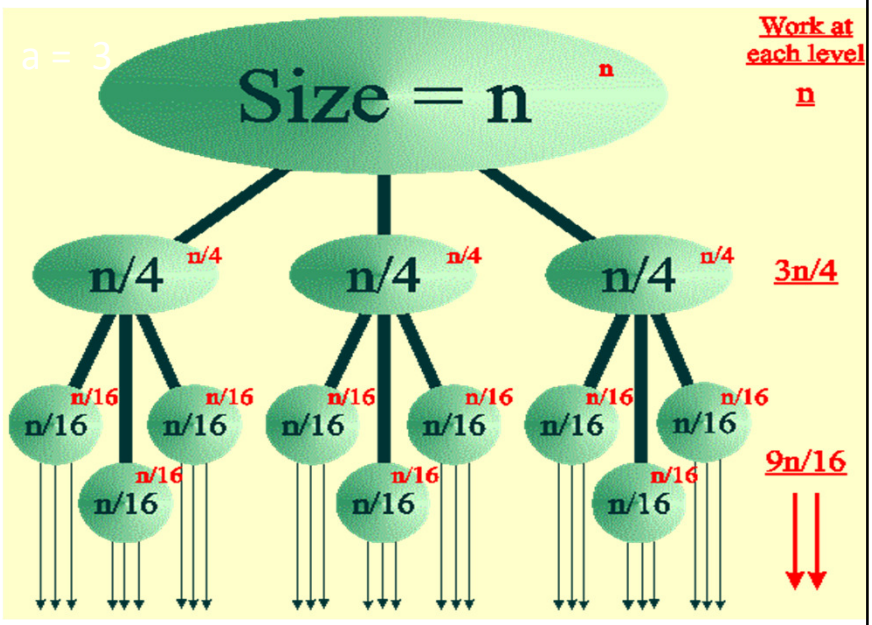
Contoh: T(n) = aT([n/4])+ n

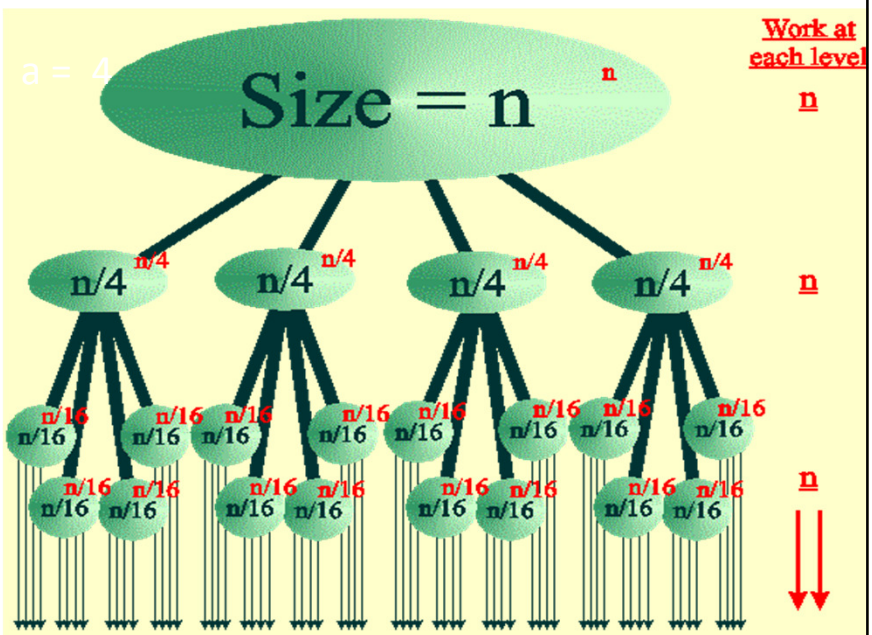
a=3: n, $(3/4)$n, $(3/4)^2$n, ...

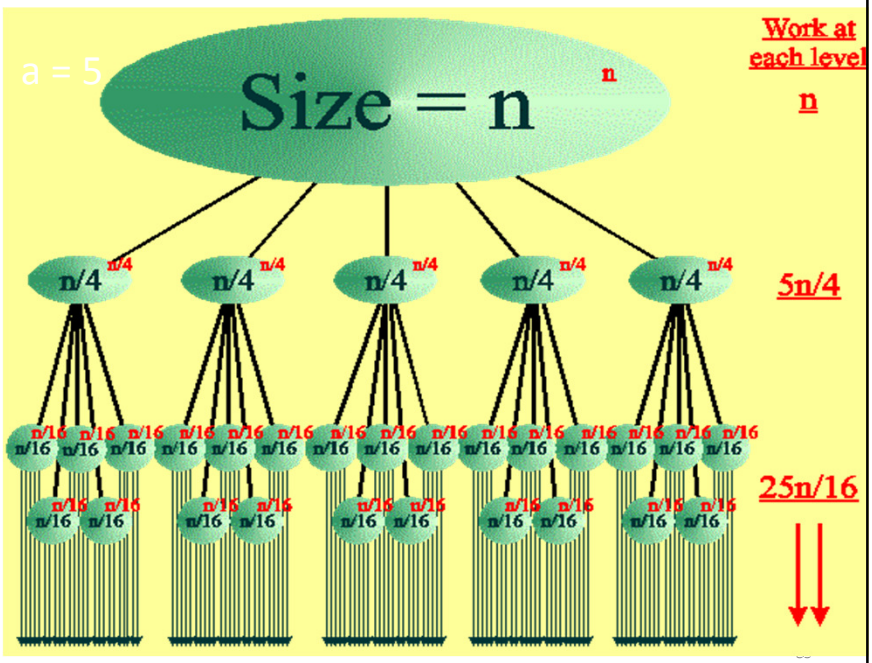
a=4: n, $(4/4)$n, $(4/4)^2$n, ...

a=5: n, $(5/4)$n, $(5/4)^2$n, ...

80


a = 3
Size = n
n
Work at each level
n
n/4
n/4
n/4
n/4
n/4
n/4
3n/4
n/16
9n/16
81


a = 4
Size = n
n
Work at each level
n
n/4
n
n/16
n
82



$T_n = aT_{n/4} + n$

a=3: n, $(3/4)n$, $(3/4)^2n$, ...
Kasus 2 dari MT

a=4: n, $(4/4)n$, $(4/4)^2n$, ...
Kasus 3 dari MT

a=5: n, $(5/4)n$, $(5/4)^2n$, ...
Kasus 1 dari MT

84

Metode Master

- Metode master menyediakan semacam "cookbook" untuk menyelesaikan persamaan rekursif dalam bentuk:

  $$T(n) = aT(n/b) + f(n)$$

  dengan $a \geq 1$ dan $b > 1$ adalah konstanta dan f(n) adalah fungsi yang secara asimtotik positif.

- Dalam metode master perlu diingat 3 kasus, namun sesudahnya bentuk-bentuk rekursif dapat diselesaikan dengan lebih mudah.

85

$T(n) = aT(n/b) + f(n)$ $a \geq 1$, $b > 1$

- jika **$f(n) = O(n^{\log_b (a-\varepsilon)})$**, $\varepsilon > 0$
  maka **$T(n) = \Theta(n^{\log_b a})$**
- jika **$f(n) = \Theta(n^{\log_b a})$**
  maka **$T(n) = \Theta(n^{\log_b a} \log n)$**
- jika **$f(n) = \Omega(n^{\log_b (a+\varepsilon)})$**, $\varepsilon > 0$ dan
  $a\, f(n/b) \leq c\, f(n)$ untuk $c < 1$
  maka **$T(n) = \Theta(f(n))$**

86

Gunakan metode Master untuk menentukan
$T(n) = \Theta(?)$ bila : $T(n) = 7T(n/2) + n^2$

Solusi:

$a = 7, b = 2, f(n) = n^2, \quad n^{\log_2 7} = n^{2.803..}$

$n^2 = O(n^{\log_2 7 - \varepsilon})$, oleh karena

$n^2 \leq cn^{2.803..- \varepsilon} \quad \varepsilon, c > 0$

$$T(n) = \Theta(n^{\log_2 7})$$

87

Gunakan metode Master untuk menentukan
$T(n) = \Theta(?)$ bila : $T(n) = 4T(n/2) + n^2$

Solusi:

$a = 4, b = 2, f(n) = n^2$

$n^2 = \Theta(n \log_2 4) = \Theta(n^2)$

$$T(n) = \Theta(n^{\log_2 4} \log n) = \Theta(n^2 \log n)$$

88

Gunakan metode Master untuk menentukan
$T(n) = \Theta(?)$ bila : $T(n) = 2T(n/3) + n^3$

Solusi:

$a = 2, b = 3, f(n) = n^3$

$n^{\log_3 2} = n0.63..$

$n^3 = \Omega(n^{\log_3 2} + \varepsilon)$ ok $n^3 \geq cn0.63..+ \varepsilon \quad \varepsilon, c > 0$

dan $2f(n/3) = 2n^3/27 \leq cn^3$ utk $2/27 < c < 1$

$$T(n) = \Theta(n^3)$$

89

Gunakan metode Master untuk menentukan
$T(n) = \Theta(?)$ bila : $T(n) = 3T(n/2) + n\log n$

Solusi:

$a = 3, b = 2, f(n) = n\log n$

$n\log n = O(n^{\log_2 3 - \varepsilon})$

$n\log n \leq cn^{1.5}$

$\log n < cn^{0.5}$

$c = 1, n_0 = 16$

$$T(n) = \Theta(n^{\log_2 3})$$

90

Gunakan metode Master untuk menentukan T(n) = Θ(?) bila : $T(n) = 2T\left(\frac{n}{2}\right) + 10n$

Solusi: $a = 2$, $b = 2$, $f(n) = 10n$, $\log_b a = \log_2 2 = 1$

dapat dicek bahwa

$$f(n) \in \Theta\left(n^{\log_b a}\right) \quad 10n \in \Theta\left(n^1\right) = \Theta(n)$$

jadi

$$T(n) \in \Theta\left(n^{\log_b a} \log(n)\right). \; T(n) \in \Theta(n \log(n)).$$

91