

Algoritma *Divide and Conquer*

(Bagian 2)

Bahan Kuliah IF2211 Strategi Algoritma

Oleh: Rinaldi M



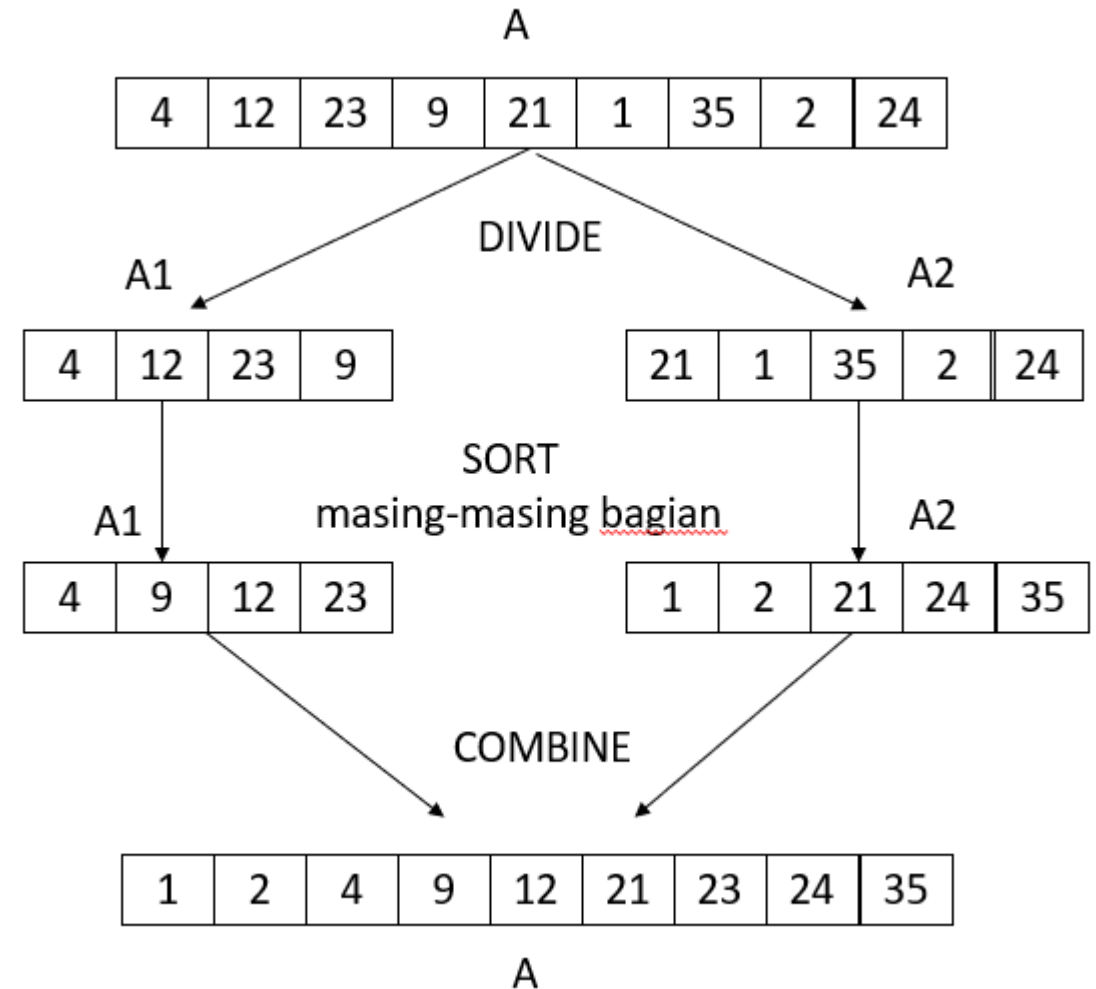
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika ITB
2026

4. Pengurutan Secara *Divide and Conquer*

- Algoritma pengurutan secara *brute force*: algoritma *selection sort*, *bubble sort*, *insertion sort*.
- Ketiganya memiliki kompleksitas algoritma $O(n^2)$.
- Dengan metode *divide and conquer*, dapatkah dihasilkan algoritma pengurutan dengan kompleksitas lebih rendah dari n^2 ?

Ide pengurutan larik secara *divide and conquer*:

1. Jika ukuran larik = 1 elemen, larik sudah terurut dengan sendirinya.
2. Jika ukuran larik > 1 , bagi larik menjadi dua bagian, lalu urut masing-masing bagian
3. Gabungkan hasil pengurutan masing-masing bagian menjadi sebuah larik yang terurut.



procedure *Sort*(**input/output** A : *LarikInteger*, **input** n : **integer**)

{ Mengurutkan larik A dengan metode Divide and Conquer

Masukan: Larik A dengan n elemen

Luaran: Larik A yang terurut

}

Algoritma:

if *ukuran*(A) > 1 **then**

Bagi A menjadi dua bagian, $A1$ dan $A2$, masing-masing berukuran $n1$ dan $n2$ ($n = n1 + n2$)

Sort($A1$, $n1$) *{ urut larik bagian kiri yang berukuran n1 elemen }*

Sort($A2$, $n2$) *{ urut larik bagian kanan yang berukuran n2 elemen }*

Combine($A1$, $A2$, A) *{ gabung hasil pengurutan bagian kiri dan bagian kanan }*

end

Terdapat dua pendekatan melakukan pengurutan dengan *divide and conquer*:

1. Mudah membagi, tetapi sulit menggabung (*easy split/hard join*)
 - Pembagian larik menjadi dua bagian mudah secara komputasi (hanya membagi berdasarkan posisi atau indeks larik)
 - Penggabungan dua buah larik terurut menjadi sebuah larik terurut sulit secara komputasi (diukur dari kompleksitas algoritmanya)
2. Sulit membagi, tetapi mudah menggabung (*hard split/easy join*)
 - Pembagian larik menjadi dua bagian sukar secara komputasi (pembagiannya berdasarkan nilai elemen, bukan posisi elemen larik)
 - Penggabungan dua buah larik terurut menjadi sebuah larik terurut mudah dilakukan secara komputasi

Contoh: Misalkan larik A adalah sebagai berikut:

A	8	1	4	6	9	3	5	7
-----	---	---	---	---	---	---	---	---

Dua pendekatan (*approach*) pengurutan:

1. Mudah membagi, sulit menggabung (*easy split/hard join*)

Tabel A dibagi dua berdasarkan posisi elemen:

<i>Divide:</i>	$A1$	8	1	4	6	$A2$	9	3	5	7
----------------	------	---	---	---	---	------	---	---	---	---

<i>Sort:</i>	$A1$	1	4	6	8	$A2$	3	5	7	9
--------------	------	---	---	---	---	------	---	---	---	---

<i>Combine:</i>	$A1$	1	3	4	5	6	7	8	9
-----------------	------	---	---	---	---	---	---	---	---

Algoritma pengurutan yang termasuk jenis ini:urut-gabung (*Merge Sort*)

2. Sulit membagi, mudah menggabung (*hard split/easy join*)

Tabel A dibagidua berdasarkan nilai elemennya. Misalkan elemen-elemen $A1 \leq$ elemen-elemen $A2$.

A	8	1	4	6	9	3	5	7
-----	---	---	---	---	---	---	---	---

<i>Divide:</i>	$A1$	5	1	4	3
----------------	------	---	---	---	---

$A2$	9	6	8	7
------	---	---	---	---

<i>Sort:</i>	$A1$	1	3	4	5
--------------	------	---	---	---	---

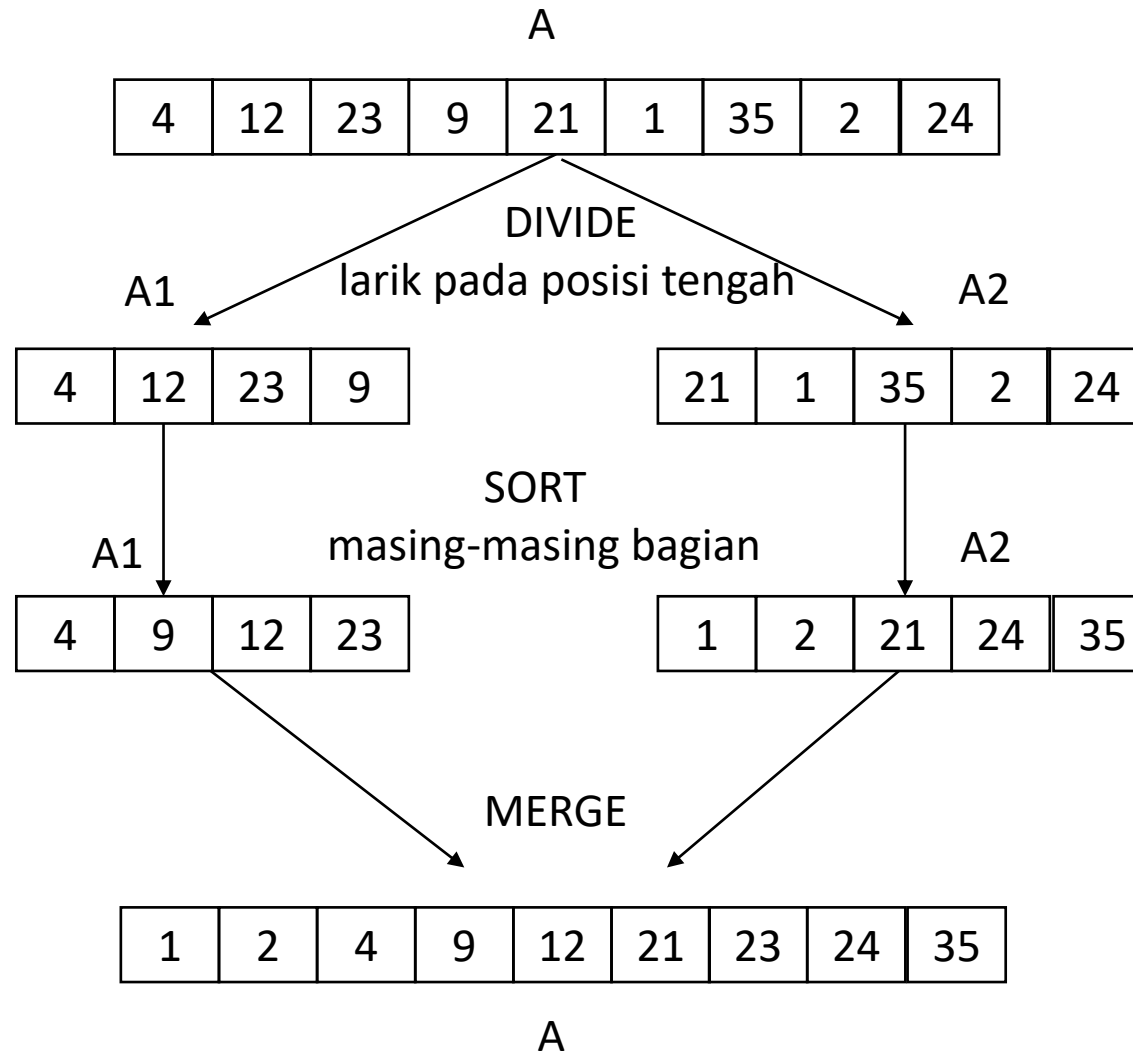
$A2$	6	7	8	9
------	---	---	---	---

<i>Combine:</i>	A	1	3	4	5	6	7	8	9
-----------------	-----	---	---	---	---	---	---	---	---

Algoritma pengurutan yang termasuk jenis ini: urut-cepat (*Quick Sort*)

4.1 Merge Sort

- Ide *merge sort*:



Pertanyaan:

1. Larik dibagi sampai ukurannya (n) tinggal berapa elemen?
2. Bagaimana menggabungkan dua larik terurut menjadi satu larik terurut?

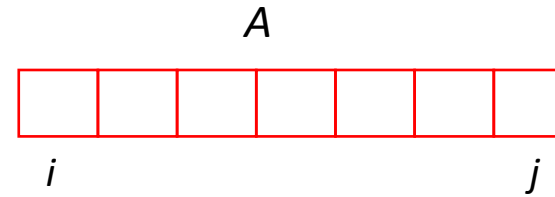
Jawaban:

1. Sampai $n = 1$
2. Gunakan algoritma *merge*

Algoritma *Merge Sort* (A, n):

1. Jika $n = 1$, maka larik A sudah terurut dengan sendirinya (langkah SOLVE).
2. Jika $n > 1$, maka
 - (a) DIVIDE: bagi larik A menjadi dua bagian pada posisi pertengahan, masing-masing bagian berukuran $n/2$ elemen.
 - (b) CONQUER: secara rekursif, terapkan *Merge Sort* pada masing-masing bagian.
 - (c) MERGE: gabung hasil pengurutan kedua bagian sehingga diperoleh larik A yang terurut.

- Dalam notasi *pseudo-code*:



procedure MergeSort(input/output *A* : LarikInteger, input *i, j* : integer)

{ Mengurutkan larik *A[i..j]* dengan algoritma Merge Sort.

Masukan: Larik *A[i..j]* yang sudah terdefinisi elemen-elemennya

Luaran: Larik *A[i..j]* yang terurut

}

Kamus

k : integer

Algoritma:

if *i* < *j* **then**

k ← (*i* + *j*) div 2

MergeSort(*A*, *i*, *k*)

MergeSort(*A*, *k* + 1, *j*)

Merge(*A*, *i*, *k*, *j*)

end

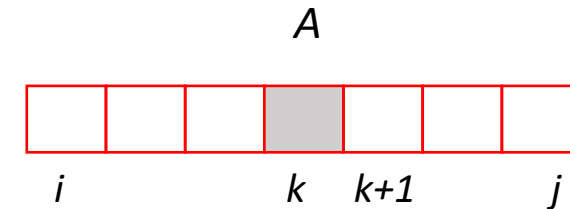
{ ukuran(*A*) > 1 }

{ bagi *A* pada posisi pertengahan }

{ urut upalarik *A[i..k]* }

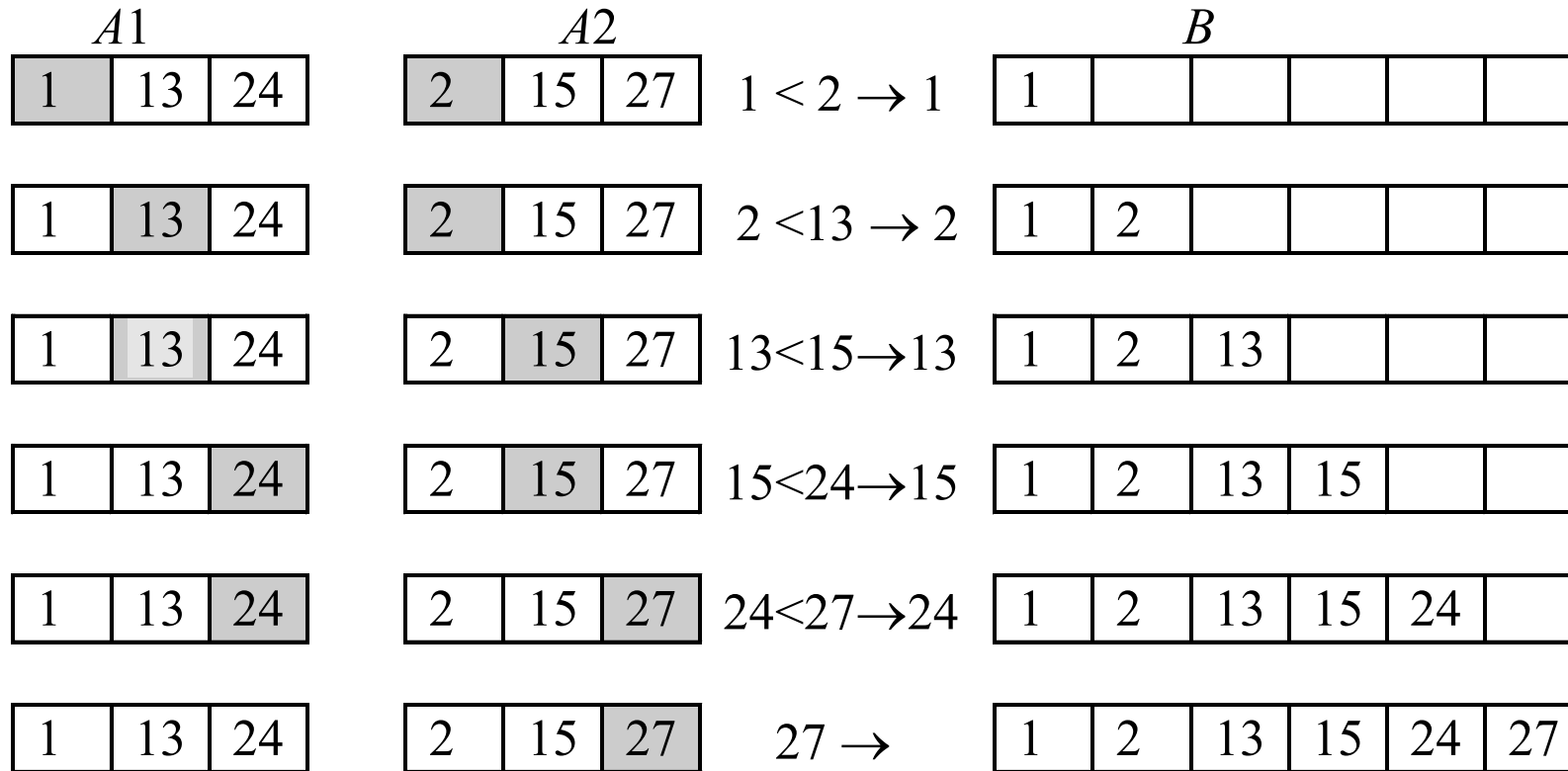
{ urut upalarik *A[k+1..j]* }

{ gabung hasil pengurutan *A[i..k]* dan *A[k+1..j]* menjadi *A[i..j]* }



Pemanggilan pertama kali: MergeSort(*A*, 1, *n*)

Contoh *Merge* dua larik terurut menjadi satu larik terurut:

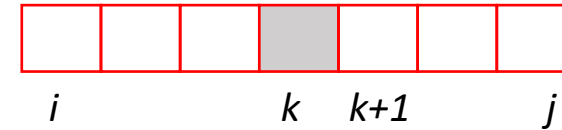


procedure Merge(input/output $A : \text{LarikInteger}$, input $i, k, j : \text{integer}$)

{ Menggabung larik $A[i..k]$ dan larik $A[k+1..j]$ menjadi larik $A[i..j]$ yang terurut menaik. A

Masukan: $A[i..k]$ dan $A[k+1..j]$ sudah terurut menaik.

Luaran: $A[k+1..j]$ yang terurut menaik. }



Kamus

$B : \text{LarikInteger}$ { larik temporer untuk menyimpan hasil penggabungan }

$p, q, r : \text{integer}$

Algoritma:

$p \leftarrow i$ { $A[i .. k]$ }

$q \leftarrow k + 1$ { $A[k+1 .. j]$ }

$r \leftarrow i$

while ($p \leq k$) **and** ($q \leq j$) **do**

if $A[p] \leq A[q]$ **then**

$B[r] \leftarrow A[p]$ { salin elemen $A[p]$ dari larik bagian kiri ke dalam larik B }

$p \leftarrow p + 1$

else

$B[r] \leftarrow A[q]$ { salin elemen $A[q]$ dari larik bagian kanan ke dalam larik B }

$q \leftarrow q + 1$

endif

$r \leftarrow r + 1$

endwhile

{ $p > k$ or $q > j$ }

..... continued

{ salin sisa larik A bagian kiri ke larik B, jika masih ada }

while ($p \leq k$) **do**

$B[r] \leftarrow A[p]$

$p \leftarrow p + 1$

$r \leftarrow r + 1$

endwhile

{ $p > k$ }

{ salin sisa larik A bagian kanan ke larik B, jika masih ada }

while ($q \leq j$) **do**

$B[r] \leftarrow A[q]$

$q \leftarrow q + 1$

$r \leftarrow r + 1$

endwhile

{ $q > j$ }

{ salin kembali elemen-elemen larik B ke dalam A }

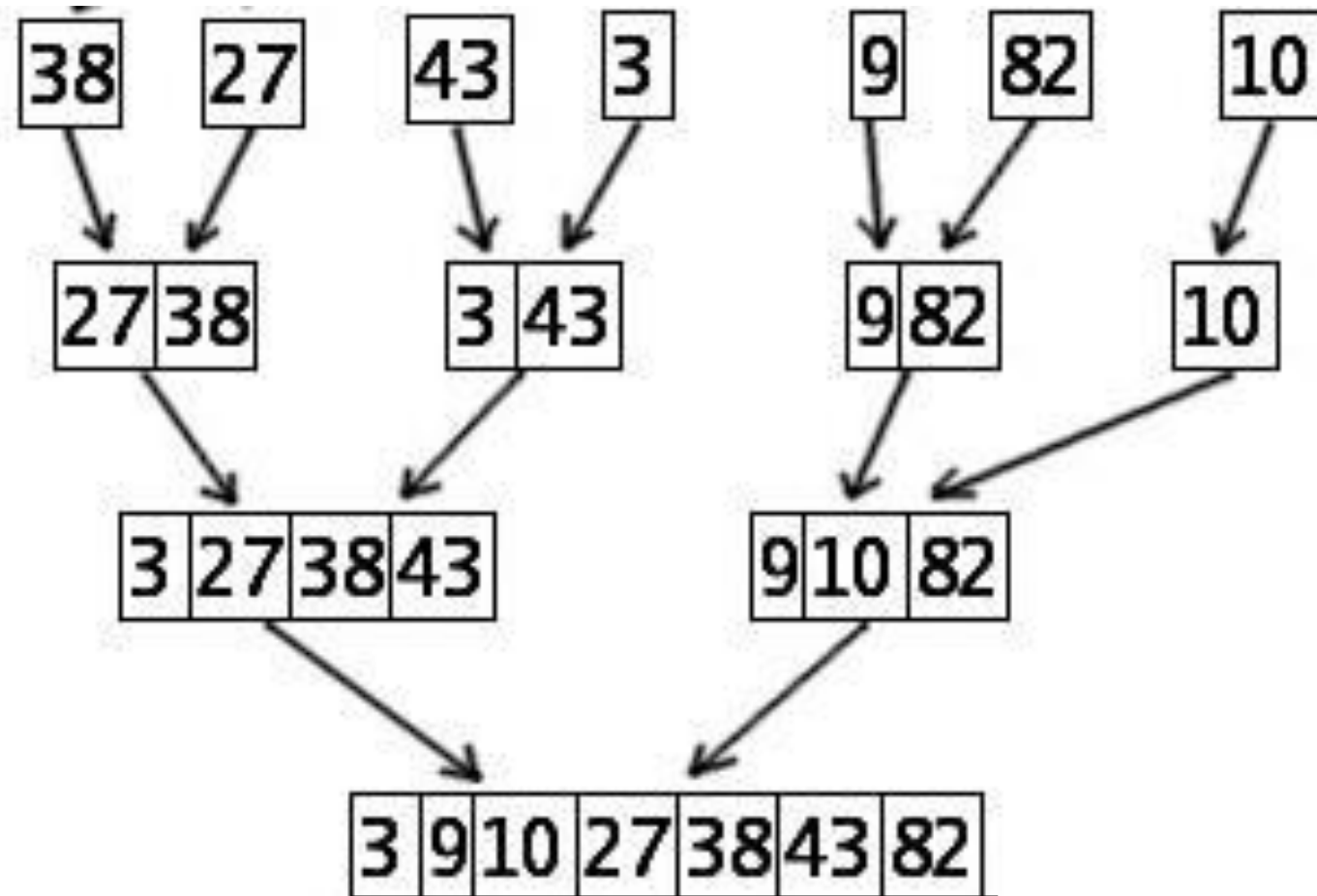
for $r \leftarrow i$ **to** j **do**

$A[r] \leftarrow B[r]$

endfor

{ diperoleh larik A yang terurut membesar }

Contoh proses *merge* di dalam *Merge Sort*:



Contoh 4: Pengurutan larik A di bawah ini dengan *Merge Sort*

4	12	23	9	21	1	35	2
---	----	----	---	----	---	----	---

DIVIDE, CONQUER, dan SOLVE:

4	12	23	9	21	1	35	2
---	----	----	---	----	---	----	---

4	12	23	9
---	----	----	---

21	1	35	2
----	---	----	---

4	12
---	----

23	9
----	---

21	1
----	---

35	2
----	---

4	12
---	----

23	9
----	---

21	1
----	---

35	2
----	---

MERGE:

4	12
---	----

9	23
---	----

1	21
---	----

2	35
---	----

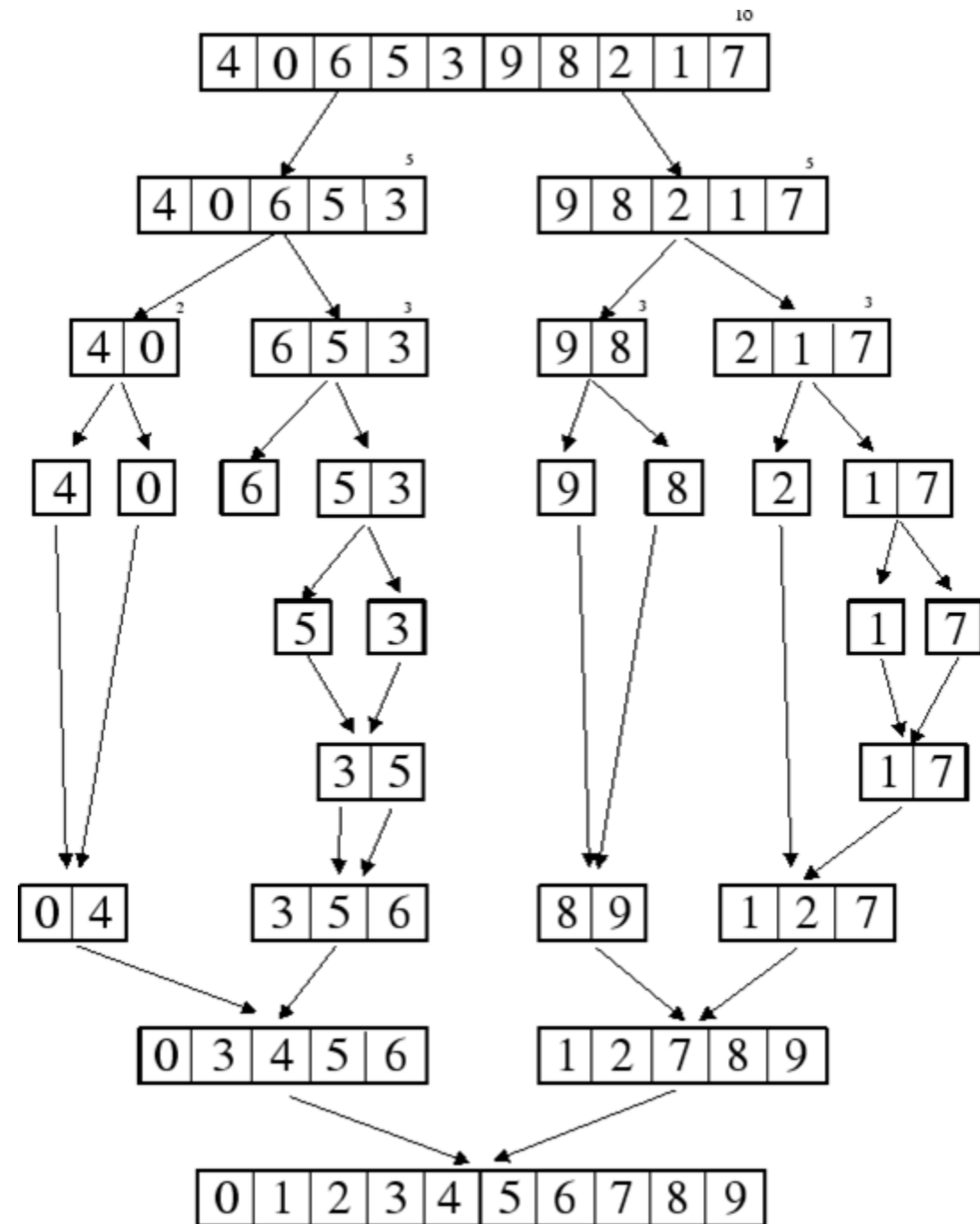
4	9	12	23
---	---	----	----

1	2	21	35
---	---	----	----

1	2	4	9	12	21	23	35
---	---	---	---	----	----	----	----

Terurut!

Contoh 5:



Kompleksitas waktu *Merge Sort*

- Kompleksitas algoritma *Merge Sort* diukur dari jumlah operasi perbandingan elemen-elemen larik.
- Jumlah perbandingan elemen-elemen larik di dalam prosedur *Merge* adalah $O(n)$, yaitu berbanding lurus dengan jumlah elemen larik, atau cn , c konstanta.
- Jumlah perbandingan elemen-elemen larik seluruhnya:

$$\begin{aligned} T(n) &= \text{Mergesort untuk pengurutan dua buah upalarik berukuran } n/2 + \\ &\quad \text{jumlah perbandingan elemen di dalam prosedur } \textit{Merge} \\ &= 2T(n/2) + cn \end{aligned}$$

- Sehingga:
$$T(n) = \begin{cases} a & , n = 1 \\ 2T(n/2) + cn & , n > 1 \end{cases}$$

- Penyelesaian persamaan rekursif secara iteratif :

Untuk menyederhanakan perhitungan, asumsikan $n = 2^k$

$$\begin{aligned}
 T(n) &= 2T(n/2) + cn \\
 &= 2(2T(n/4) + cn/2) + cn = 4T(n/4) + 2cn \\
 &= 4(2T(n/8) + cn/4) + 2cn = 8T(n/8) + 3cn \\
 &= \dots \\
 &= 2^k T(n/2^k) + kcn
 \end{aligned}$$

$$n = 2^k \rightarrow k = {}^2\log n$$

sehingga

$$T(n) = nT(1) + cn {}^2\log n = an + cn {}^2\log n = O(n {}^2\log n)$$

- Jadi, kompleksitas algoritma *Merge Sort* adalah $O(n {}^2\log n)$, lebih baik daripada kompleksitas algoritma pengurutan secara *brute force*.

4.2 *Quicksort*

- Algoritma pengurutan *Quicksort* merupakan algoritma pengurutan yang terkenal dan tercepat (sesuai namanya).
- *Quicksort* ditemukan oleh Tony Hoare tahun 1959 dan dipublikasikan tahun 1962.
- *Quicksort* merupakan algoritma pengurutan secara *divide and conquer*, dan termasuk ke dalam pendekatan sulit membagi, mudah menggabung (*hard split/easy join*)

- Di dalam *Quicksort*, larik *A* dibagidua (istilahnya: dipartisi) menjadi dua buah upalarik, *A1* dan *A2*, sedemikian sehingga:

semua elemen di $A1 \leq$ semua elemen di *A2*.

A

8	1	4	6	9	3	5	7
---	---	---	---	---	---	---	---

Divide: *A1*

5	1	4	3
---	---	---	---

A2

9	6	8	7
---	---	---	---

Sort: *A1*

1	3	4	5
---	---	---	---

A2

6	7	8	9
---	---	---	---

Combine: *A*

1	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---

- Terdapat beberapa varian algoritma Quicksort. Versi orisinal adalah dari Hoare seperti di bawah ini:

Misalkan larik A akan diurut menaik (*ascending order*).

Teknik mempartisi larik menjadi dua bagian:

- (i) pilih $x \in \{ A[1], A[2], \dots, A[n] \}$ sebagai *pivot*,
- (ii) pindai larik dari kiri sampai ditemukan elemen $A[p] \geq x$
- (iii) pindai larik dari kanan sampai ditemukan elemen $A[q] \leq x$
- (iv) pertukarkan $A[p] \Leftrightarrow A[q]$
- (v) ulangi (ii), dari posisi $p + 1$, dan (iii), dari posisi $q - 1$, sampai kedua pemindaian bertemu di tengah larik ($p \geq q$)

Contoh 6. Misalkan larik A berisi elemen-elemen berikut:

8 1 4 6 9 3 5 7

Misalkan *pivot* = 6 (elemen tengah larik). Langkah-langkah partisi adalah sbb:

(i): 8 1 4 **6** 9 3 5 7

 pivot

(ii) & (iii):

	awal pindai →									← awal pindai
	8	1	4	6	9	3	5			7
	↑ p						↑ q			

(iv):

5	1	4	6	9	3	8	7
↑						↑	
tukar							

(ii) & (iii): 5 1 4 6 9 3 8 7

awal pindai
→

←
awal pindai

↑ p ↑ q

(iv): 5 1 4 3 9 6 8 7

↑ ↑

tukar

(ii) & (iii): 5 1 4 3 9 6 8 7

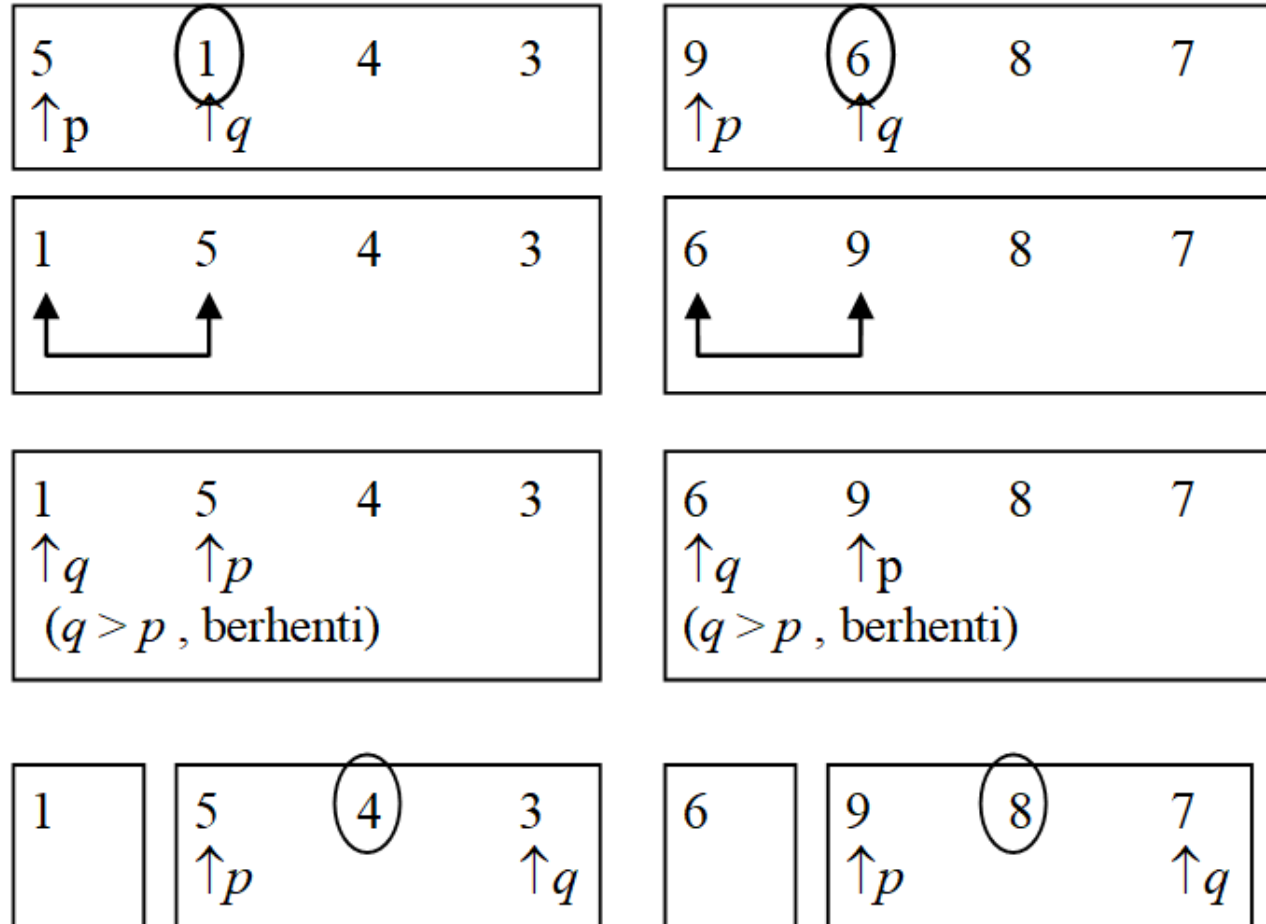
↑ q ↑ p ($q < p$, berhenti)

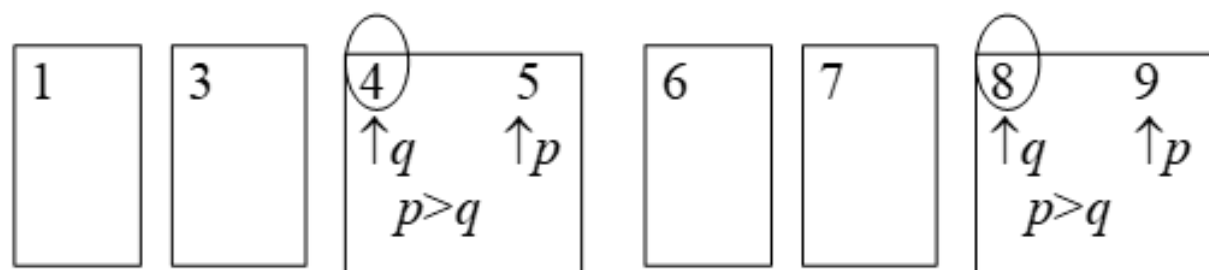
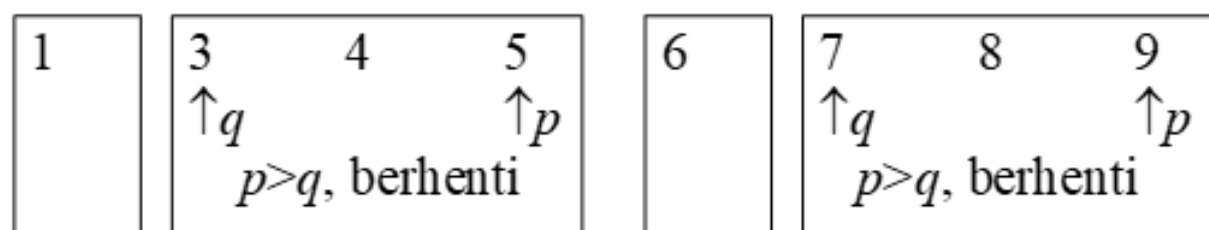
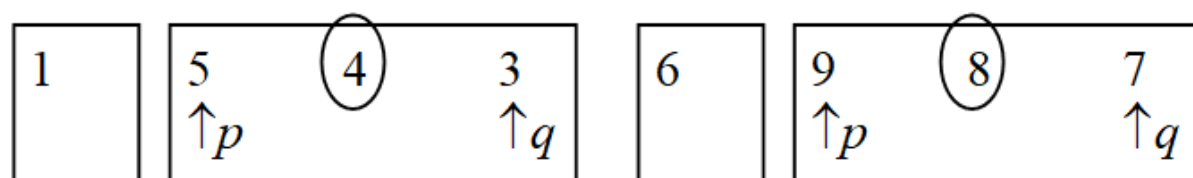
← →

Hasil partisi pertama:

kiri:	5	1	4	3	(< 6)
kanan:	9	6	8	7	(≥ 6)

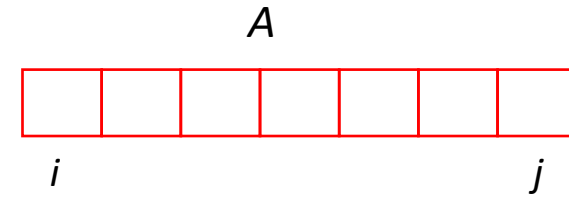
Teruskan partisi untuk setiap bagian sampai berukuran satu elemen:





(terurut)

- *Pseudo-code* algoritma Quicksort:



```

procedure QuickSort(input/output  $A$  : LarikInteger, input  $i, j$  : integer)
{ Mengurutkan larik  $A[i..j]$  dengan algoritma Quicksort.
  Masukan: Larik  $A[i..j]$  yang sudah terdefinisi elemen-elemennya
  Luaran: Larik  $A[i..j]$  yang terurut
}

```

Kamus

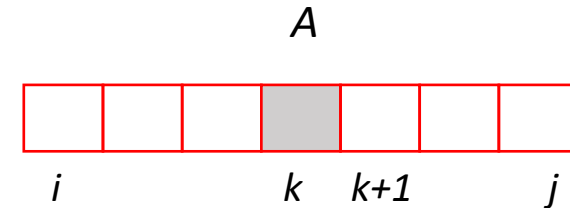
k : integer

Algoritma:

```

if  $i < j$  then           {  $Ukuran(A) > 1$  }
  Partisi( $A, i, j, k$ )      { Larik dipartisi pada indeks  $k$  }
  QuickSort( $A, i, k$ )      { Urut  $A[i..k]$  dengan Quick Sort }
  QuickSort( $A, k+1, j$ )    { Urut  $A[k+1..j]$  dengan Quick Sort }
endif

```



procedure *Partisi*(input/output A : *LarikInteger*, input i, j : integer, output q : integer)

{ Membagi larik $A[i..j]$ menjadi upalarik $A[i..q]$ dan $A[q+1..j]$

Masukan: Larik $A[i..j]$ udah terdefinisi harganya.

Luaran: upalarik $A[i..q]$ dan upalarik $A[q+1..j]$ sedemikian sehingga $A[i..q]$ lebih kecil dari larik $A[q+1..j]$ }



Kamus

pivot, temp : integer

Algoritma:

$pivot \leftarrow$ pilih sembarang elemen larik sebagai pivot, misalkan $pivot =$ elemen tengah

$p \leftarrow i$ *{awal pemindaian dari kiri }*

$q \leftarrow j$ *{ awal pemindaian dari kanan }*

repeat

while $A[p] < pivot$ **do**

$p \leftarrow p + 1$

endwhile

{ $A[p] \geq pivot$ }

while $A[q] > pivot$ **do**

$q \leftarrow q - 1$

endwhile

{ $A[q] \leq pivot$ }

if $p < q$ **then**

$swap(A[p], A[q])$ *{pertukarkan $A[p]$ dengan $A[q]$ }*

$p \leftarrow p + 1$ *{awal pemindaian berikutnya dari kiri }*

$q \leftarrow q - 1$ *{awal pemindaian berikutnya dari kanan }*

endif

until $p \geq q$

Versi kedua *Quicksort*: Partisi sedemikian rupa sehingga elemen-elemen larik kiri $\leq$ pivot dan elemen-elemen larik kanan $\geq$ dari pivot.

$$\underbrace{a_{i_1} \cdots a_{i_{s-1}}}_{\leq p} \quad p \quad \underbrace{a_{i_{s+1}} \cdots a_{i_n}}_{\geq p}$$

p = pivot = elemen pertama.

Contoh:

pivot
↓
5, 3, 1, 9, 8, 2, 4, 7

Partisi

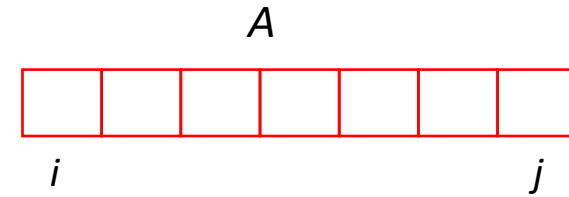
pivot
↓
2, 3, 1, 4, **5**, 8, 9, 7
↔ ↔
semua $\leq$ pivot semua $\geq$ pivot

Contoh 7 (Levitin, 2003):





- *Pseudo-code* algoritma Quicksort versi 2:



```

procedure QuickSort2(input/output A : LarikInteger, input i, j : integer)
{ Mengurutkan larik A[i..j] dengan algoritma Quicksort versi 2
  Masukan: Larik A[i..j] yang sudah terdefinisi elemen-elemennya
  Luaran: Larik A[i..j] yang terurut
}

```

Kamus

k : integer

Algoritma:

```

if i < j then                                { Ukuran(A) > 1 }
  Partisi2(A, i, j, k)                          { Larik dipartisi pada indeks k, partisi versi 2 }
  QuickSort2(A, i, k - 1)                       { Urut A[i..k - 1] dengan Quick Sort2 }
  QuickSort2(A, k + 1, j)                       { Urut A[k + 1..j] dengan Quick Sor2t }
endif

```

procedure *Partisi2*(**input/output** A : *LarikInteger*, **input** i, j : **integer**, **output** q : **integer**)

{ Membagi larik $A[i..j]$ menjadi upalarik $A[i..q-1]$ dan $A[q+1..j]$

Masukan: Larik $A[i..j]$ udah terdefinisi harganya.

Luaran: upalarik $A[i..q-1]$ dan upalarik $A[q+1..j]$ sedemikian sehingga $A[i..q-1]$ lebih kecil dari larik $A[q+1..j]$ }

Deklarasi

pivot, temp : **integer**

Algoritma:

$pivot \leftarrow A[i]$ *{ pivot = elemen pertama }*

$p \leftarrow i$ *{ awal pemindaian dari kiri }*

$q \leftarrow j + 1$ *{ awal pemindaian dari kanan }*

repeat

repeat

$p \leftarrow p + 1$

until $A[p] >= pivot$

repeat

$q \leftarrow q - 1$

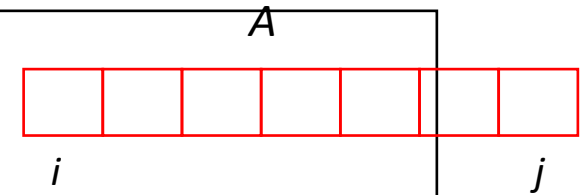
until $A[q] <= pivot$

$swap(A[p], A[q])$ *{ pertukarkan $A[p]$ dengan $A[q]$ }*

until $p \geq q$

$swap(A[p], A[q])$ *{ undo last swap when $p \geq q$ }*

$swap(A[i], A[q])$ *{ pertukarkan pivot dengan $A[q]$ }*

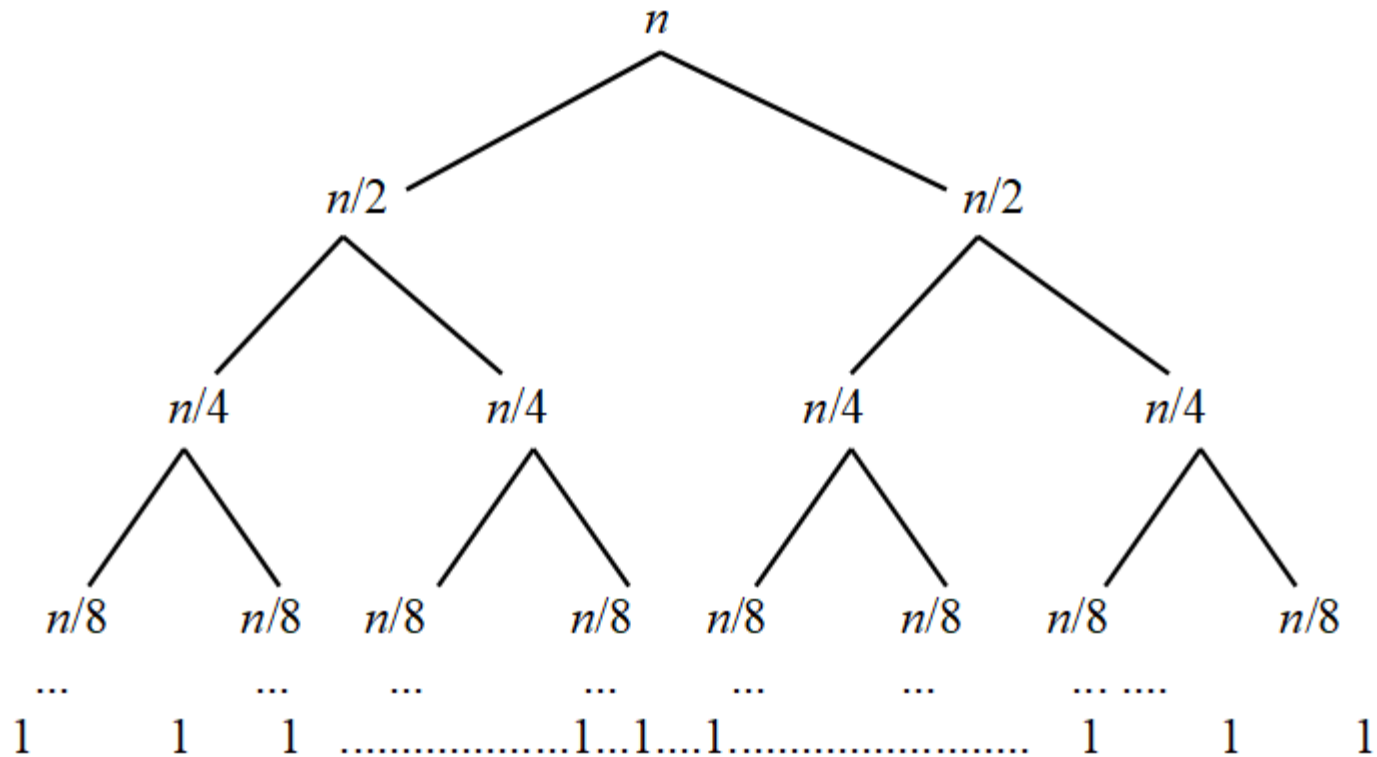


- Cara pemilihan *pivot* (khusus pada *Quicksort* versi 1):
 1. *Pivot* = elemen pertama/elemen terakhir/elemen tengah larik
 2. *Pivot* dipilih secara acak dari salah satu elemen larik.
 3. *Pivot* = elemen median larik
- Cara pemilihan pivot menentukan kompleksitas algoritma *Quicksort*

Kompleksitas Algoritma *Quicksort*:

1. Kasus terbaik (*best case*)

- Kasus terbaik terjadi bila *pivot* adalah elemen median larik sehingga larik selalu terbagi menjadi dua upalarik yang berukuran relatif sama setiap kali proses partisi.



- Kompleksitas algoritma *Quicksort* untuk kasus terbaik:

$$T(n) = \begin{cases} a & , n = 1 \\ 2T(n/2) + cn & , n > 1 \end{cases}$$

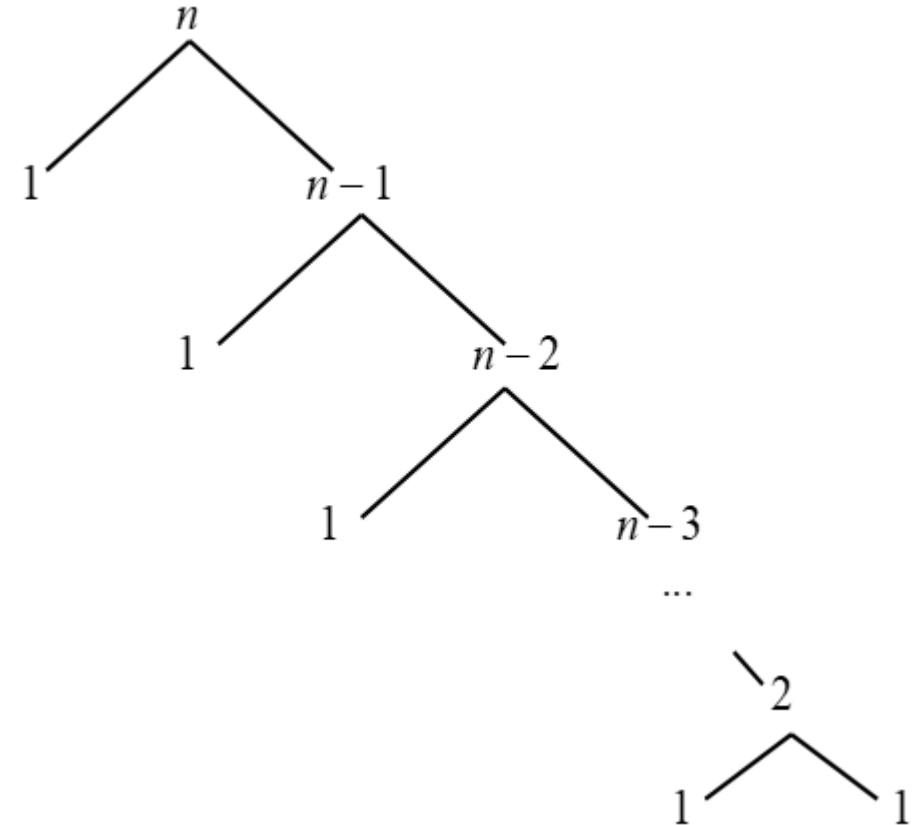
Penyelesaiannya sama seperti pada *Merge Sort*:

$$T(n) = 2T(n/2) + cn = na + cn \log n = O(n \log n).$$

- Kasus terbaik menghasilkan kompleksitas algoritma *Quicksort* yang lebih baik daripada algoritma pengurutan secara *brute force*.

2. Kasus terburuk (worst case)

- Kasus ini terjadi bila pada awalnya larik sudah terurut (menaik atau menurun), dan *pivot* selalu elemen pertama larik (elemen pertama merupakan elemen maksimum atau elemen minimum larik).
- Akibatnya, proses partisi menghasilkan ketidakseimbangan ukuran, upalarik pertama berukuran satu elemen, upalarik kedua berukuran $n - 1$ elemen.



- Kompleksitas algoritma *Quicksort* untuk kasus terburuk:

$$T(n) = \begin{cases} a & , n = 1 \\ T(n-1) + cn & , n > 1 \end{cases}$$

- Penyelesaian relasi rekursif di atas adalah pada halaman berikut

Kompleksitas waktu *Quicksort* kasus terburuk:

$$T(n) = \begin{cases} a & , n = 1 \\ T(n-1) + cn & , n > 1 \end{cases}$$

Penyelesaian:

$$\begin{aligned} T(n) &= cn + T(n-1) \\ &= cn + \{ c(n-1) + T(n-2) \} \\ &= cn + c(n-1) + \{ c(n-2) + T(n-3) \} \\ &= cn + c(n-1) + c(n-2) + \{ c(n-3) + T(n-4) \} \\ &= \dots \\ &= cn + c(n-1) + c(n-2) + c(n-3) + \dots + c2 + T(1) \\ &= c\{ n + (n-1) + (n-2) + (n-3) + \dots + 2 \} + a \\ &= c\{ (n-1)(n+2)/2 \} + a \\ &= cn^2/2 + cn/2 + (a - c) \\ &= O(n^2) \rightarrow \text{sama seperti kompleksitas algoritma } \textit{sorting} \text{ secara } \textit{brute force} \end{aligned}$$

3. Kasus rata-rata (*average case*)

- Kasus ini terjadi jika *pivot* dipilih secara acak dari elemen-elemen larik, dan peluang setiap elemen dipilih menjadi *pivot* adalah sama.
- Kompleksitas algoritma *Quicksort* untuk kasus rata-rata:

$$T_{\text{avg}}(n) = O(n^2 \log n).$$

5. Teorema Master

- Teorema Master dapat digunakan untuk menentukan notasi asimptotik kompleksitas waktu yang berbentuk relasi rekurens dengan mudah tanpa harus menyelesaikannya secara iteratif.
- Misalkan $T(n)$ adalah fungsi monoton menaik yang memenuhi relasi rekurens:

$$T(n) = aT(n/b) + cn^d$$

yang dalam hal ini $n = b^k$, $k = 1, 2, \dots$, $a \geq 1$, $b \geq 2$, $c \geq 0$ dan $d \geq 0$, maka

$$T(n) \text{ adalah } \begin{cases} O(n^d) & \text{jika } a < b^d \\ O(n^d \log n) & \text{jika } a = b^d \\ O(n^{\log_b a}) & \text{jika } a > b^d \end{cases}$$

Contoh 10. Misalkan sebuah algoritma memiliki kompleksitas waktu sebagai berikut:

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\frac{n}{2}\right) + 4, & n > 1 \end{cases}$$

Tentukan estimasi notasi Big-Oh nya.

Jawaban:

Perhatikan rekasi rekurens $T(n) = T(n/2) + 4$. Dari Teorema Master, $T(n) = aT(n/b) + cn^d$, maka $a = 1$, $b = 2$, $c = 4$, dan $d = 0$, sehingga memenuhi $a = b^d = 1$ (yaitu $1 = 2^0$).

maka relasi rekurens

$$T(n) = T(n/2) + 4$$

memenuhi *case 2* (jika $a = b^d$)

$$\begin{cases} O(n^d) & \text{jika } a < b^d \text{ Case 1} \\ O(n^d \log n) & \text{jika } a = b^d \text{ Case 2} \\ O(n^{\log_b a}) & \text{jika } a > b^d \text{ Case 3} \end{cases}$$

sehingga

$$T(n) = O(n^0 \log n) = O(\log n)$$

Contoh 11. Misalkan sebuah algoritma memiliki kompleksitas waktu sebagai berikut:

$$T(n) = \begin{cases} 1, & n = 1 \\ 7T\left(\frac{n}{5}\right) + 9n^2, & n > 1 \end{cases}$$

Tentukan estimasi notasi Big-Oh nya.

Jawaban:

Perhatikan rekasi rekurens $T(n) = 7T(n/5) + 9n^2$. Dari Teorema Master, $T(n) = aT(n/b) + cn^d$, maka $a = 7$, $b = 5$, $c = 9$, dan $d = 2$, sehingga memenuhi $a < b^d$ (yaitu $7 < 5^2$).

maka relasi rekurens

$$T(n) = 7T(n/5) + 9n^2$$

memenuhi *case 1* (jika $a < b^d$)

$$\begin{cases} O(n^d) & \text{jika } a < b^d \text{ Case 1} \\ O(n^d \log n) & \text{jika } a = b^d \text{ Case 2} \\ O(n^{\log_b a}) & \text{jika } a > b^d \text{ Case 3} \end{cases}$$

sehingga

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

Contoh 12: Pada algoritma *Mergesort/Quick Sort*,

$$T(n) = \begin{cases} a & , n = 1 \\ 2T(n/2) + cn & , n > 1 \end{cases}$$

Menurut Teorema Master, $T(n) = aT(n/b) + cn^d$, diperoleh $a = 2$, $b = 2$, $d = 1$, dan memenuhi $a = b^d$ (yaitu $2 = 2^1$) maka relasi rekurens:

$$T(n) = 2T(n/2) + cn$$

memenuhi *case 2* (jika $a = b^d$)

$$\begin{cases} O(n^d) & \text{jika } a < b^d & \text{Case 1} \\ O(n^d \log n) & \text{jika } a = b^d & \text{Case 2} \\ O(n^{\log_b a}) & \text{jika } a > b^d & \text{Case 3} \end{cases}$$

sehingga

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

Catatan: basis logaritma tidak penting di dalam notasi Big-O, sebab fungsi logaritma tumbuh pada laju yang sama untuk sembarang basis.

Contoh 13: Pada algoritma perpangkatan a^n ,

$$T(n) = \begin{cases} 1, & n = 0 \\ T\left(\frac{n}{2}\right) + 1, & n > 0 \end{cases}$$

Menurut Teorema Master, $T(n) = aT(n/b) + cn^d$, diperoleh $a = 1$, $b = 2$, $d = 0$, dan memenuhi $a = b^d$ (yaitu $1 = 2^0$) maka relasi rekurens:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

memenuhi *case 2* (jika $a = b^d$)

$O(n^d)$	jika $a < b^d$	Case 1
$O(n^d \log n)$	jika $a = b^d$	Case 2
$O(n^{\log_b a})$	jika $a > b^d$	Case 3

sehingga

$$T(n) = O(n^0 \log n) = (\log n)$$

Contoh 14. Misalkan sebuah algoritma memiliki kompleksitas waktu sebagai berikut:

$$T(n) = \begin{cases} 1, & n = 2 \\ 2T(\sqrt{n}) + 1, & n > 2 \end{cases}$$

Diasumsikan n adalah perpangkatan dari 2, atau $n = 2^m$. Tentukan estimasi notasi Big-Oh nya.

Jawaban:

Misalkan $n = 2^m$, maka $m = \log_2 n$

Maka, $T(n) = 2T(\sqrt{n}) + 1 \rightarrow T(2^m) = 2T(2^{m/2}) + 1 \rightarrow T(2^m) = 2 T(2^{m/2}) + 1$

Misalkan $S(m) = T(2^m)$, maka $S(m) = 2S(m/2) + 1$. Dari teorema master, $a = 2$, $b = 2$, $c = 1$, $d = 0$, sehingga $2 > 2^0$ (case 3),

$S(m)$ adalah $O(m^{\log_2 2}) = O(m)$.

Karena $n = 2^m$, maka $m = \log_2 n$.

Jadi, $T(n)$ adalah $O(\log n)$

$$\begin{cases} O(n^d) & \text{jika } a < b^d & \text{Case 1} \\ O(n^d \log n) & \text{jika } a = b^d & \text{Case 2} \\ O(n^{\log_b a}) & \text{jika } a > b^d & \text{Case 3} \end{cases}$$

Latihan Soal Divide and Conquer

(Soal UTS 2011) Misalkan anda mempunyai larik $A[1..n]$ yang telah berisi n elemen *integer*. *Elemen mayoritas di dalam A adalah elemen yang muncul lebih dari $n/2$ kali* (jadi, jika $n = 6$ atau $n = 7$, elemen mayoritas terdapat paling sedikit 4 kali).

Rancanglah algoritma *divide and conquer* (tidak dalam bentuk *pseudo-code*, tapi dalam bentuk uraian deskriptif) untuk menemukan elemen mayoritas di dalam A (atau menentukan tidak terdapat elemen mayoritas).

Jelaskan algoritma anda dengan contoh sebuah larik berukuran 8 elemen. Selanjutnya, perkirakan kompleksitas algoritmanya dalam hubungan rekursif (misalnya $T(n) = bT(n/p) + h(n)$), lalu selesaikan $T(n)$ tersebut.

Jawaban:

1. Jika $n = 1$, maka elemen tunggal tersebut adalah mayoritasnya sendiri.
2. Jika $n > 1$, maka bagi larik menjadi dua bagian (kiri dan kanan) yang masing-masing berukuran sama ($n/2$), lalu cari mayoritas pada setiap bagian (CONQUER)
3. Tahap *combine*. Ada empat kemungkinan kasus:

Kasus 1: tidak ada mayoritas pada setiap bagian, sehingga larik gabungan keduanya tidak memiliki mayoritas.

Return: “no majority”

Contoh: 4 3 4 2 7 5 2 1

4 3 4 2 7 5 2 1

Ingat definisi
mayoritas!

no majority

no majority

4 3 4 2 7 5 2 1
“no majority”

Kasus 2: bagian kanan memiliki mayoritas, bagian kiri tidak. Pada larik gabungan, hitung kemunculan elemen mayoritas bagian kanan tersebut;

Jika elemen tersebut mayoritas pada larik gabungan, *return* elemen tersebut, kalau tidak *return* “no majority”

Contoh:

4 3 4 2 7 4 4 4

4 3 4 2 7 4 4 4

no majority

majority = 4

4 3 4 2 7 4 4 4

Jumlah elemen 4 = 5 buah → mayoritas

“majority = 4”

Ingat definisi
mayoritas!

Ingat definisi
mayoritas!

Contoh lain (tidak ada mayoritas):

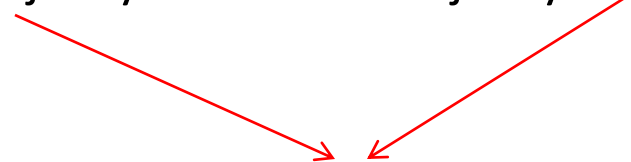
4 3 5 2 7 4 4 4



4 3 5 2 7 4 4 4

no majority

majority = 4



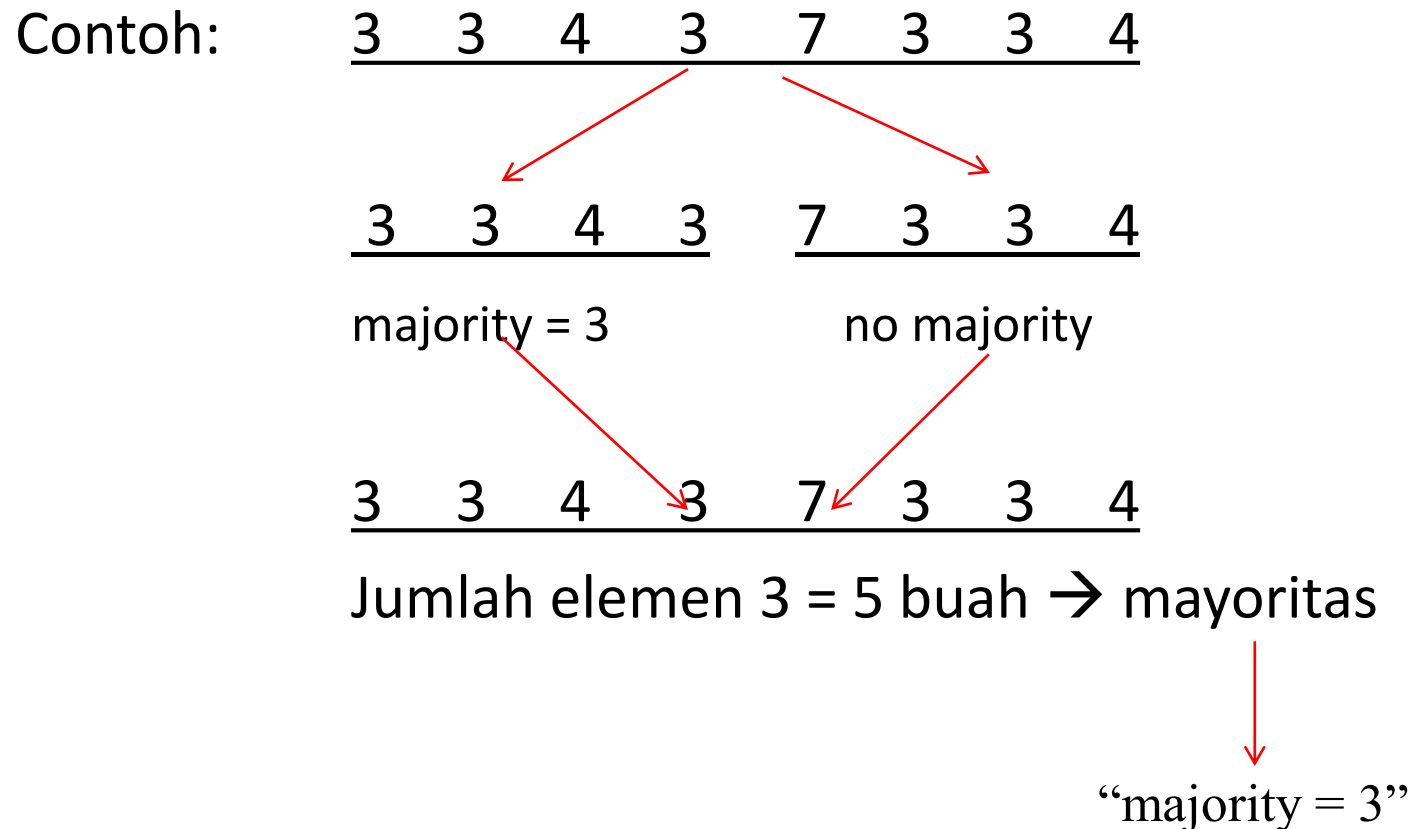
4 3 5 2 7 4 4 4

Jumlah elemen 4 = 4 buah → bukan mayoritas



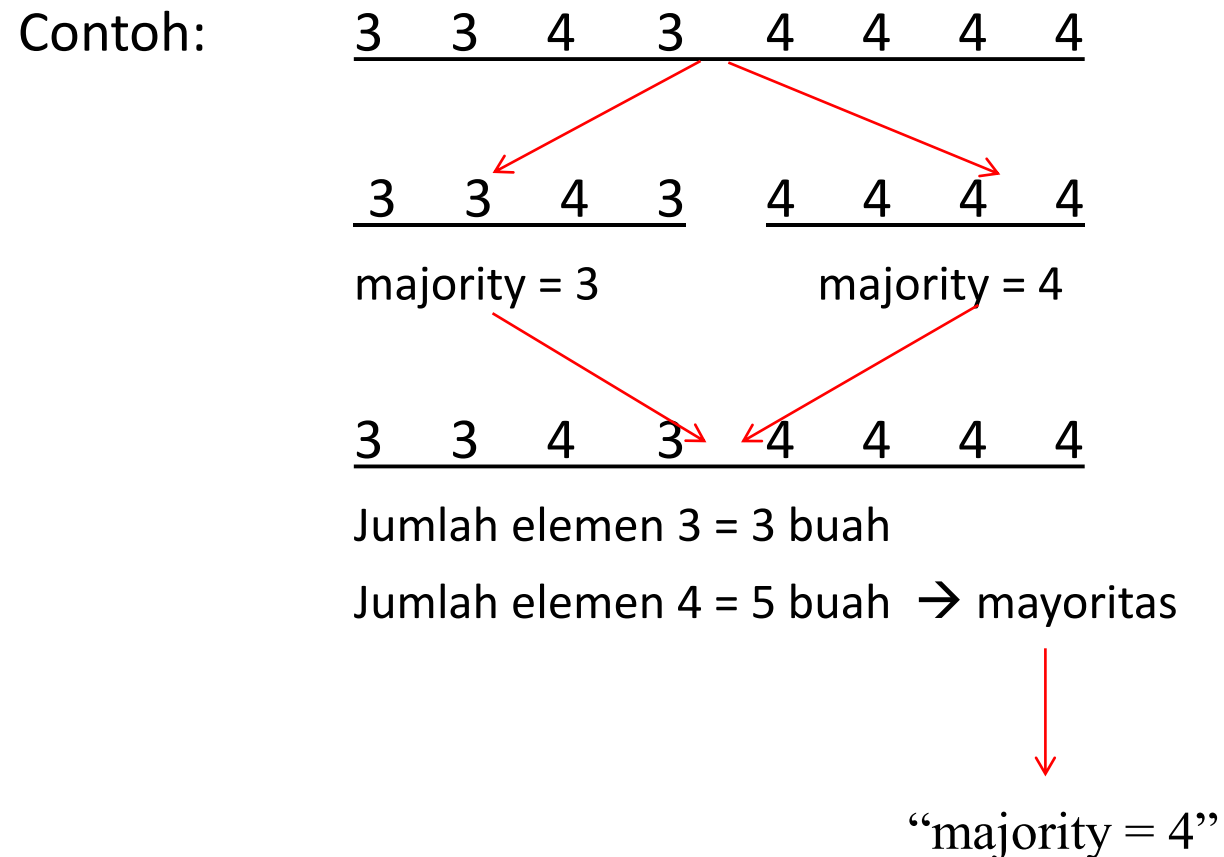
“no majority”

Kasus 3: bagian kiri memiliki mayoritas, bagian kanan tidak. Pada larik gabungan, hitung jumlah kemunculan elemen mayoritas bagian kiri tersebut. Jika elemen tersebut mayoritas pada larik gabungan, *return* elemen tersebut, kalau tidak *return* “no majority”

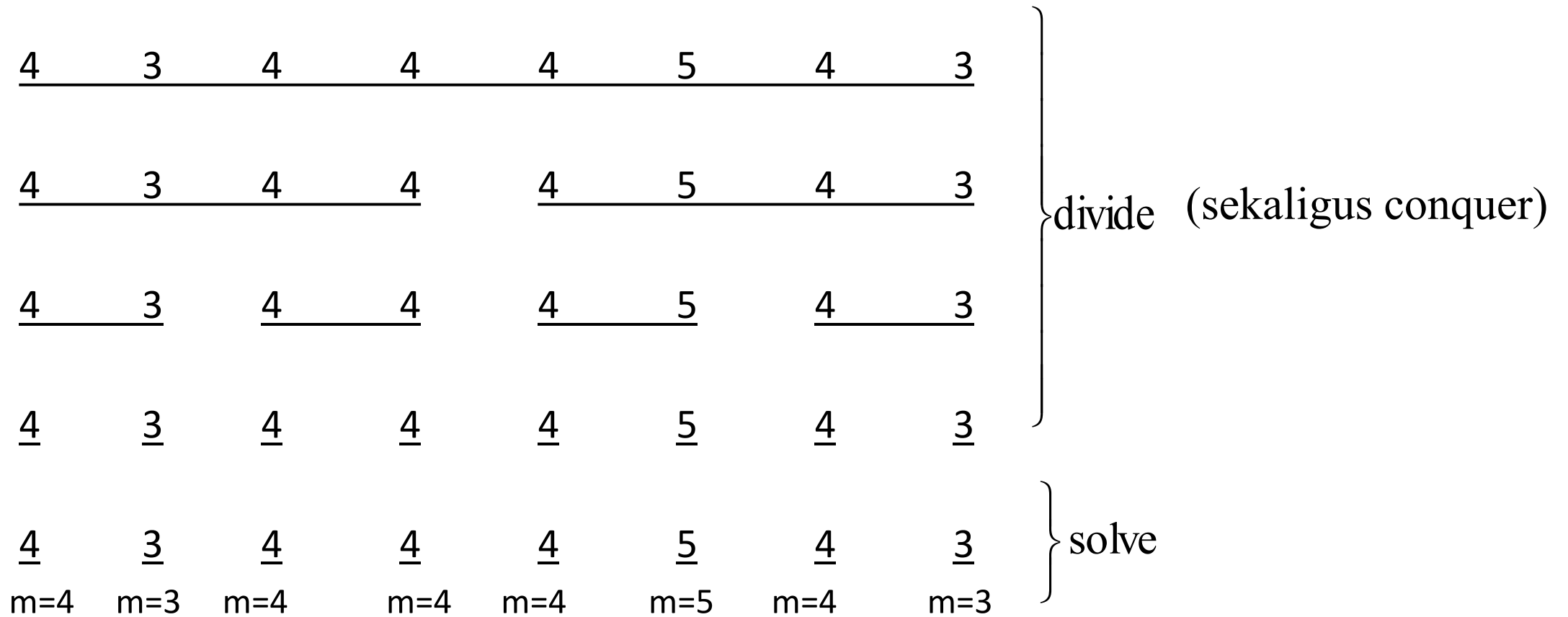


Kasus 4: bagian kiri dan bagian kanan memiliki mayoritas, Pada larik gabungan, hitung jumlah kemunculan kedua elemen kandidat mayoritas tersebut.

Jika salah satu kandidat adalah elemen mayoritas, *return* elemen tersebut, kalau tidak *return* “no majority”



Contoh keseluruhan:



4 3 4 4 4 5 4 3
 m=4 m=3 m=4 m=4 m=4 m=5 m=4 m=3

4 3 4 4 4 5 4 3
 nm m = 4 nm nm

4 3 4 4 4 5 4 3
 m = 4 nm

4 3 4 4 4 5 4 3
 m = 4

} combine

Kompleksitas waktu algoritma mayoritas:

$T(n)$ = jumlah operasi perbandingan elemen yang terjadi
(pada saat menghitung jumlah elemen yang sama dengan kandidat mayoritas)

Untuk $n = 1$, jumlah perbandingan = 0, secara umum = a .

Pada $n > 1$, terdapat dua pemanggilan rekursif, masing-masing untuk $n/2$ elemen larik.

Jumlah perbandingan elemen yang terjadi paling banyak $2n$ (*upper bound*) yaitu pada kasus 4, untuk *array* berukuran n . Secara umum jumlah perbandingan = cn .

Jadi,

$$T(n) = \begin{cases} a & , n = 1 \\ 2T(n/2) + cn & , n > 1 \end{cases}$$

Bila diselesaikan dengan Teorema Master, $T(n) = aT(n/b) + cn^d$, diperoleh $a = 2$, $b = 2$, $d = 1$, dan memenuhi $a = b^d$ (yaitu $2 = 2^1$) maka relasi rekurens

$$T(n) = 2T(n/2) + cn$$

memenuhi case 2 (jika $a = b^d$)

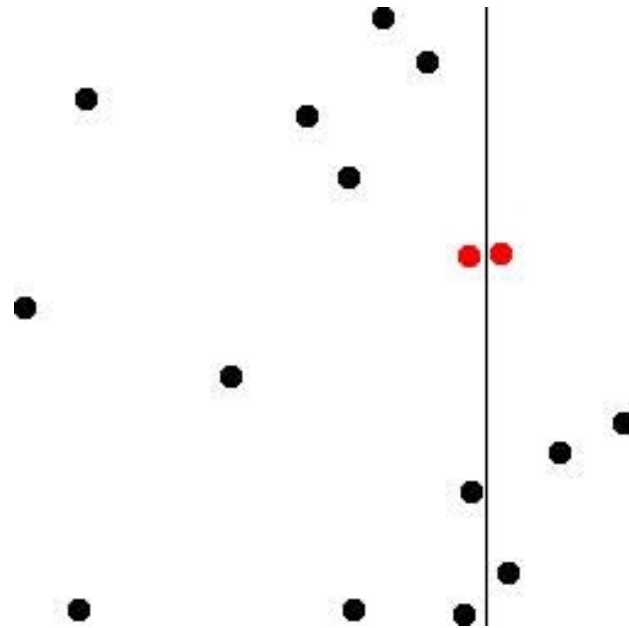
$$\begin{cases} O(n^d) & \text{jika } a < b^d \\ O(n^d \log n) & \text{jika } a = b^d \\ O(n^{\log_b a}) & \text{jika } a > b^d \end{cases}$$

sehingga

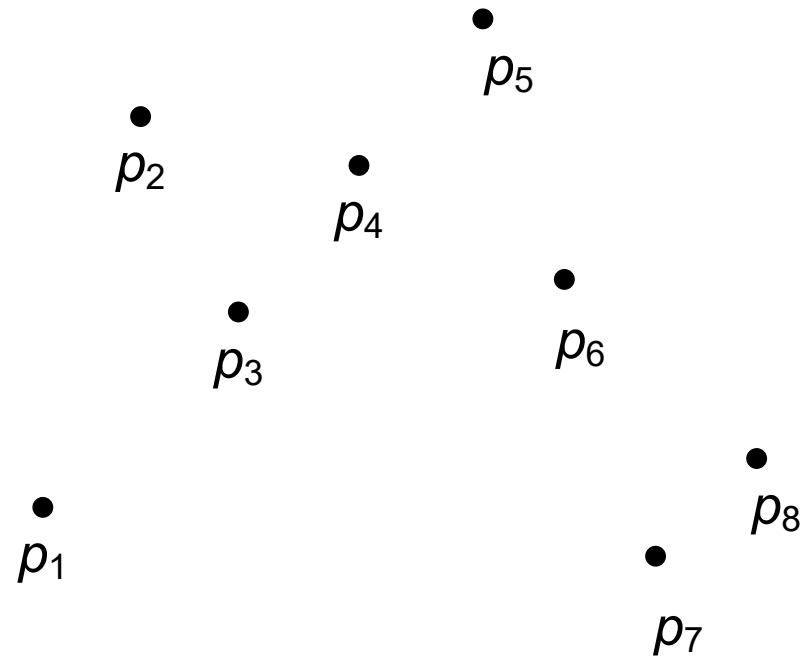
$$T(n) = O(n^1 \log n) = O(n \log n)$$

6. Mencari Pasangan Titik Terdekat (*Closest Pair*)

Persoalan: Diberikan himpunan titik, P , yang terdiri dari n buah titik pada bidang 2-D, (x_i, y_i) , $i = 1, 2, \dots, n$. Tentukan sepasang titik di dalam P yang jaraknya terdekat satu sama lain.



$n = 8$



Jarak dua buah titik $p_1 = (x_1, y_1)$ dan $p_2 = (x_2, y_2)$ dihitung dengan rumus Euclidean:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Penyelesaian secara *Brute Force*

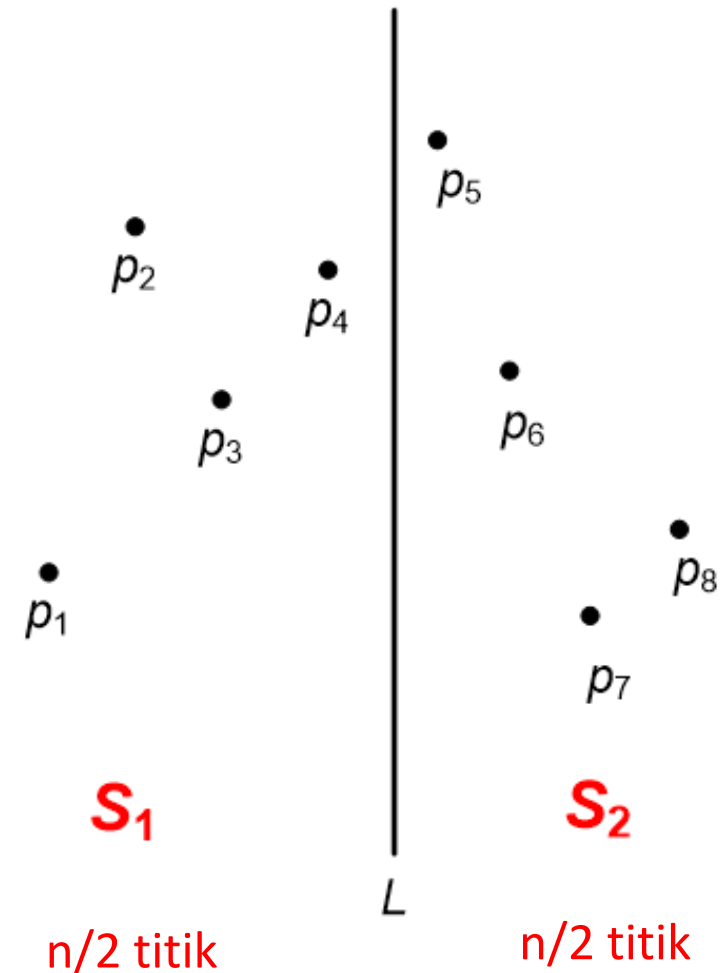
- Hitung jarak setiap pasang titik. Terdapat sebanyak $C(n, 2) = n(n - 1)/2$ pasangan titik yang harus dihitung jaraknya. (C = notasi kombinasi)
- Pilih pasangan titik yang mempunyai jarak terkecil sebagai solusinya.
- Kompleksitas algoritma adalah $O(n^2)$.

Penyelesaian secara *Divide and Conquer*

- Asumsi: $n = 2^k$ (jumlah titik adalah perpangkatan dari dua)
- Praproses: titik-titik di dalam P diurut menaik berdasarkan nilai absisnya (x).
- Algoritma *Closest Pair*:
 1. SOLVE: jika $n = 2$, maka jarak kedua titik dihitung langsung dengan rumus Euclidean.

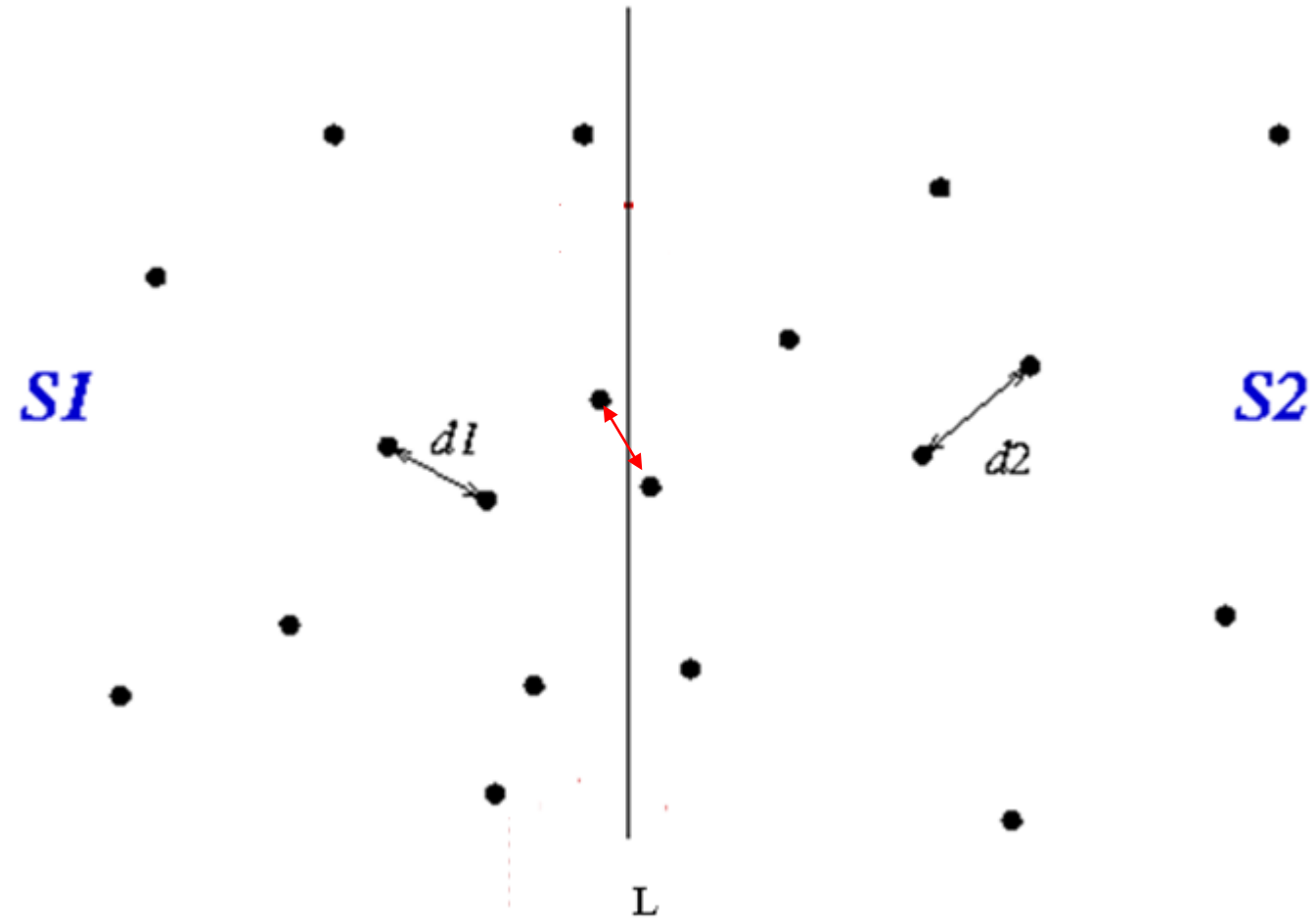
2. DIVIDE: Bagi himpunan titik ke dalam dua bagian, S_1 dan S_2 , setiap bagian mempunyai jumlah titik yang sama. L adalah garis maya yang membagi dua himpunan titik ke dalam dua sub-himpunan, masing-masing $n/2$ titik.

Garis maya L dapat dihampiri sebagai $y = x_{n/2}$ (ingatlah titik-titik sudah diurut menaik berdasarkan absis (x)).



3. CONQUER: Secara rekursif, terapkan algoritma *D-and-C* pada masing-masing bagian untuk mencari sepasang titik terdekat.
4. COMBINE: Pasangan titik yang jaraknya terdekat ada tiga kemungkinan letaknya:
 - (a) Pasangan titik terdekat terdapat di dalam bagian S_1 .
 - (b) Pasangan titik terdekat terdapat di dalam bagian S_2 .
 - (c) Pasangan titik terdekat dipisahkan oleh garis batas L , yaitu satu titik di S_1 dan satu titik di S_2 .

Jika kasusnya adalah (c), maka lakukan tahap ketiga (akan dijelaskan kemudian) untuk mendapatkan jarak dua titik terdekat sebagai solusi persoalan semula.



procedure *FindClosestPair*(**input** $P : \text{SetOfPoint}$, $n : \text{integer}$, **output** $d : \text{real}$)

{ Mencari jarak terdekat sepasang titik di dalam himpunan P

Masukan: $P = \{p_1, p_2, \dots, p_n\}$, titik-titik di dalam P sudah terurut menaik berdasarkan absisnya (x)

Luaran: d adalah jarak sepasang titik terdekat }

Kamus:

$d1, d2 : \text{real}$

Algoritma:

if $n = 2$ **then**

$d \leftarrow$ jarak kedua titik dengan rumus Euclidean

else

$S1 \leftarrow \{p_1, p_2, \dots, p_{n/2}\}$

$S2 \leftarrow \{p_{n/2+1}, p_{n/2+2}, \dots, p_n\}$

FindClosestPair($S1, n/2, d1$)

FindClosestPair($S2, n/2, d2$)

$d \leftarrow \text{MIN}(d1, d2)$ { bandingkan dulu $d1$ dengan $d2$ untuk menentukan yang terkecil }

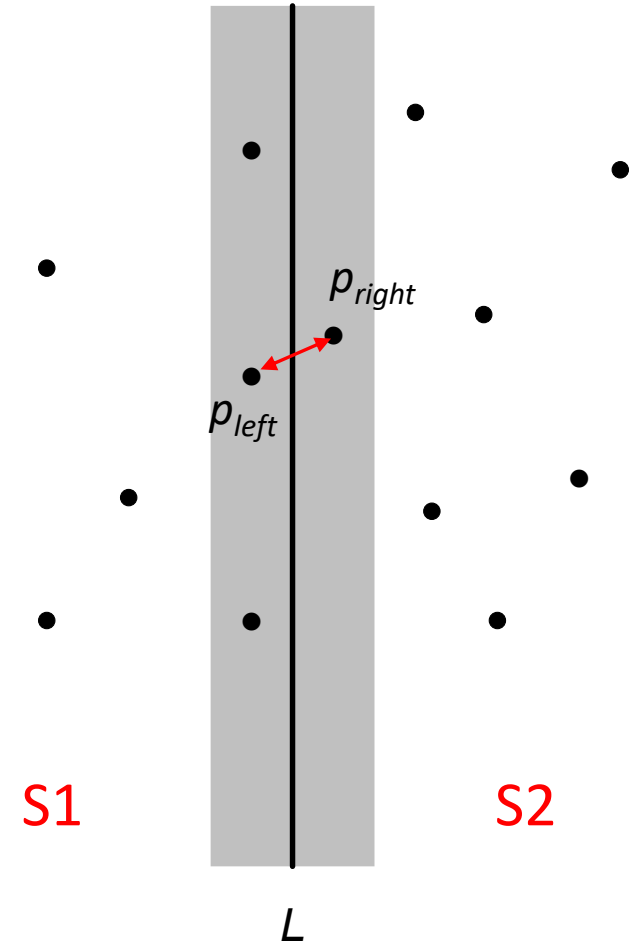
{ ****

Tentukan apakah terdapat titik p_{left} di $S1$ dan p_{right} di $S2$ dengan $\text{jarak}(p_{\text{left}}, p_{\text{right}}) < d$. Jika ada, maka set d dengan jarak terkecil tersebut.

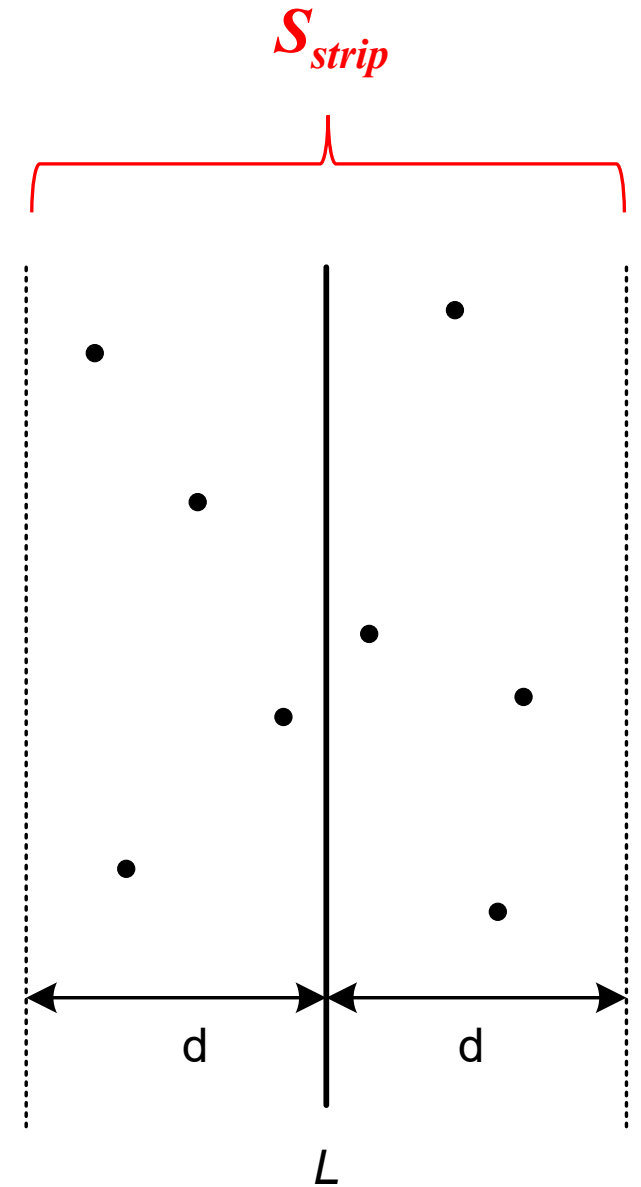
**** }

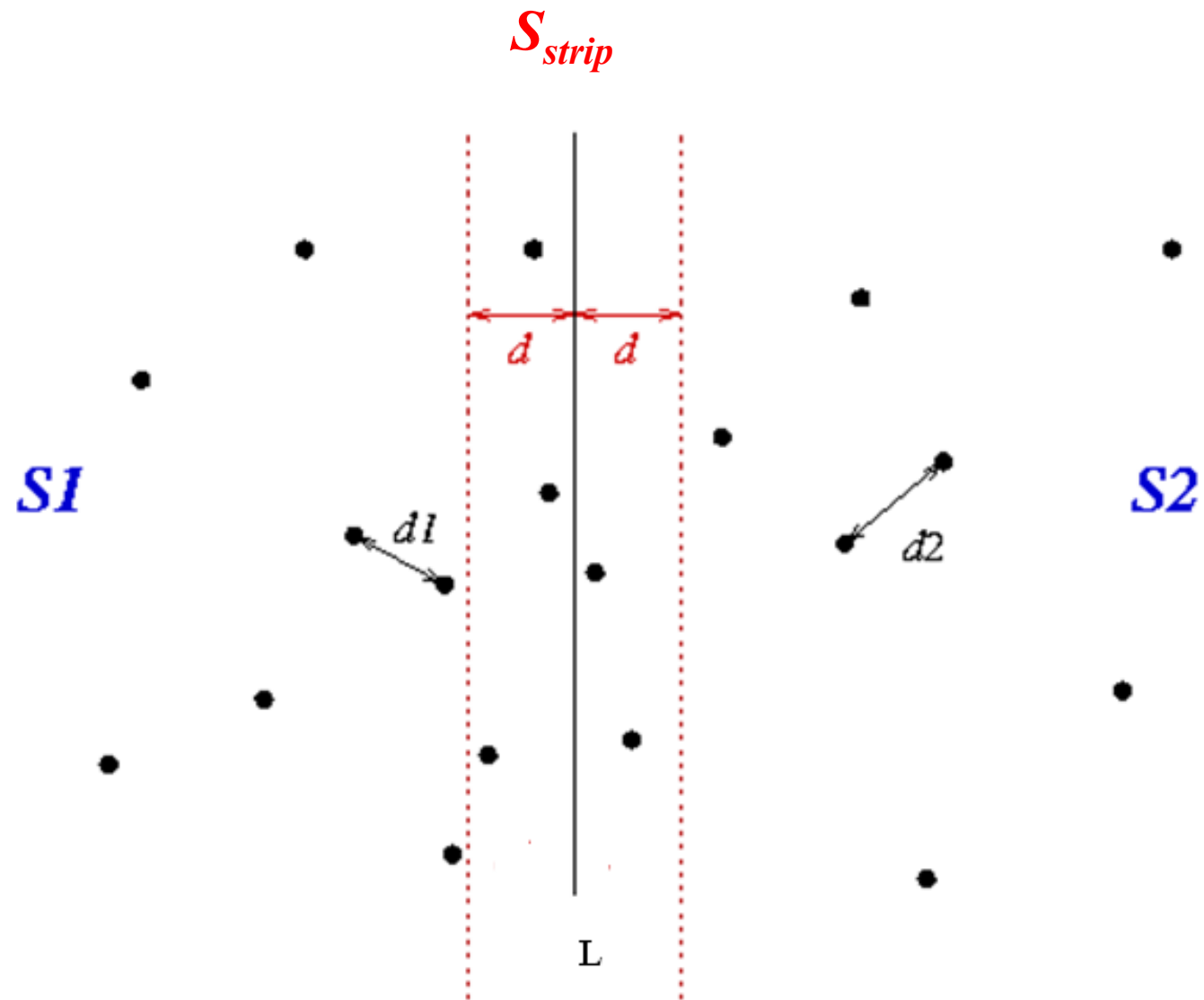
endif

- Jika terdapat pasangan titik p_{left} and p_{right} yang jaraknya lebih kecil dari d , maka kasusnya adalah:
 - (i) Absis x dari p_{left} dan p_{right} berbeda paling banyak sebesar d .
 - (ii) Ordinat y dari p_{left} dan p_{right} berbeda paling banyak sebesar d .
- Ini berarti p_{left} and p_{right} adalah sepasang titik yang berada di daerah sekitar garis vertikal L (daerah abu-abu)
- Berapa lebar strip abu-abu tersebut?



- Kita membatasi titik-titik di dalam *strip* selebar $2d$
- Oleh karena itu, implementasi tahap ketiga adalah sbb:
 - (i) Temukan semua titik di S_1 yang memiliki absis x minimal $x_{n/2} - d$.
 - (ii) Temukan semua titik di S_2 yang memiliki absis x maksimal $x_{n/2} + d$.
- Sebut semua titik-titik yang ditemukan pada langkah (i) dan (ii) tersebut sebagai himpunan S_{strip} yang berisi s buah titik.





Keterangan: $d = \text{MIN}(d1, d2)$

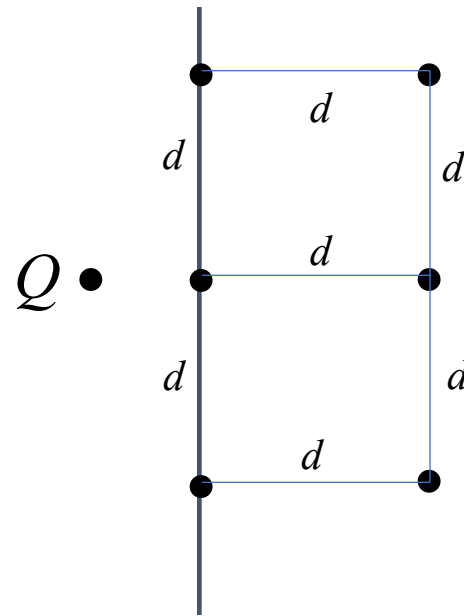
- Urutkan titik-titik di dalam S_{strip} dalam urutan ordinat y yang menaik. Misalkan $q_1, q_2, \dots, q_s$ menyatakan hasil pengurutan.
- Hitung jarak setiap pasang titik di dalam S_{strip} dan bandingkan apakah jaraknya lebih kecil dari d dengan algoritma berikut:

```

for  $i \leftarrow 1$  to  $s$  do
  for  $j \leftarrow i+1$  to  $s$  do
    if  $(ABS(q_i.x - q_j.x) > d \text{ or } ABS(q_i.y - q_j.y) > d)$  then
      { tidak diproses }
    else
       $d3 \leftarrow EUCLIDEAN(q_i, q_j)$  { hitung jarak  $q_i$  dan  $q_j$  dengan rumus Euclidean }
      if  $d3 < d$  then { bandingkan apakah  $d3$  lebih kecil dari  $d$  }
         $d \leftarrow d3$ 
      endif
    endif
  endfor
endfor

```

- Jika diamati, kita tidak perlu memeriksa semua titik di dalam area strip abu-abu tersebut.
- Untuk sebuah titik Q di sebelah kiri garis L , kita hanya perlu memeriksa paling banyak enam buah titik saja yang jaraknya sebesar d dari ordinat Q (ke atas dan ke bawah), serta titik-titik yang berjarak d dari garis L .



Kompleksitas Algoritma *Closest Pair*

- Pengurutan titik-titik dalam absis x dan ordinat y dilakukan sebelum menerapkan algoritma *Divide and Conquer*.
- Pemrosesan titik-titik di dalam S_{strip} memerlukan waktu $t(n) = cn = O(n)$.
- Kompleksitas algoritma *closest pair*:

$$T(n) = \begin{cases} 2T(n/2) + cn & , n > 2 \\ a & , n = 2 \end{cases}$$

Solusi dari persamaan di atas dengan Teorema Master adalah $T(n) = O(n \log n)$
→ Lebih baik dari algoritma *brute force* yang $O(n^2)$

BERSAMBUNG