Algoritma *Divide and Conquer*

(Bagian 2)

Bahan Kuliah IF2211 Strategi Algoritma

Oleh: Rinaldi Munir

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika ITB
2021

4. Pengurutan Secara *Divide and Conquer*

- Algoritma pengurutan secara *brute force*: algoritma *selection sort*, *bubble sort*, *insertion sort.*

- Ketiganya memiliki kompleksitas algoritma $O(n^2)$.

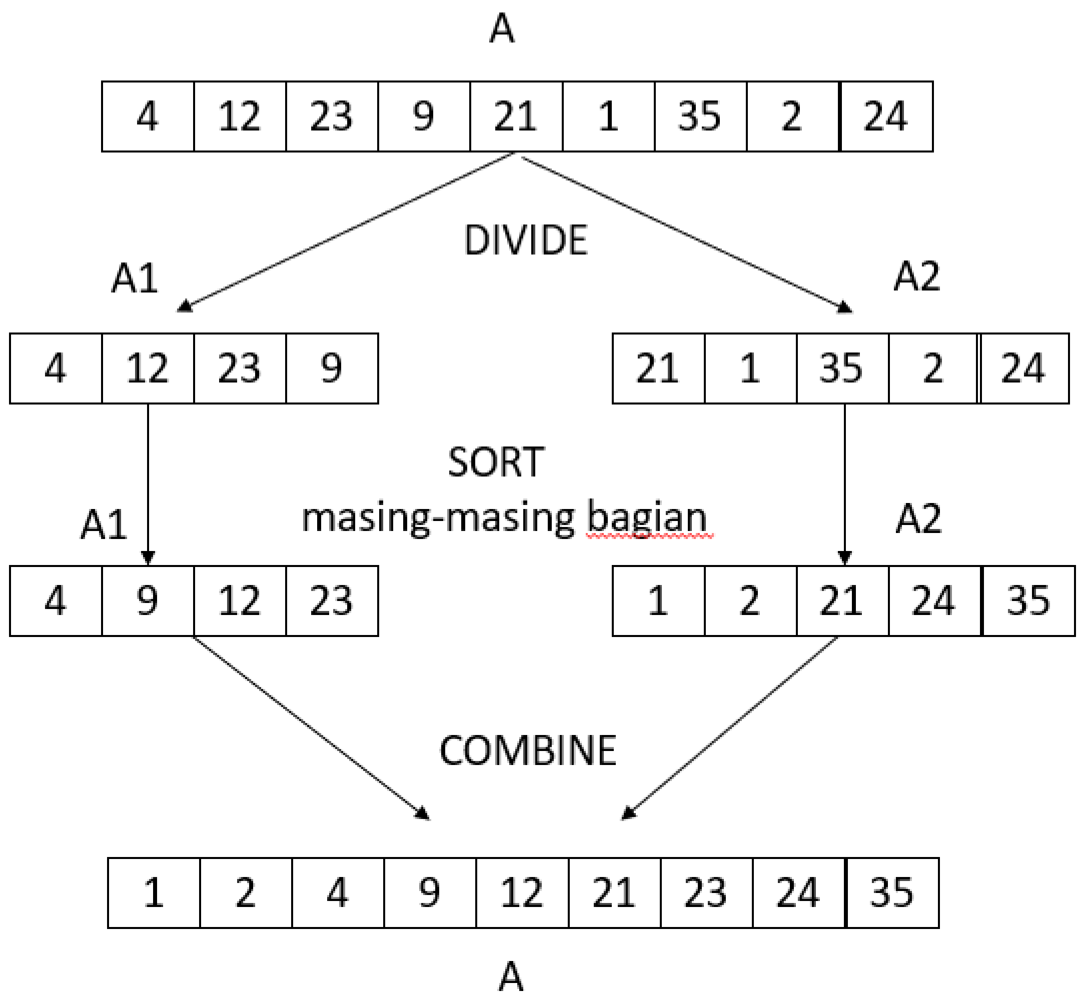
- Dengan metode *divide and conquer*, dapatkah dihasilkan algoritma pengurutan dengan kompleksitas lebih rendah dari n^2?

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 sukar secara komputasi (ditinjau 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* dibagidua berdasarkan posisi elemen:

Divide: *A*1

8	1	4	6

*A*2

9	3	5	7

Sort: *A*1

1	4	6	8

*A*2

3	5	7	9

Combine: *A*1

1	3	4	5	6	7	8	9

Algoritma pengurutan yang termasuk jenis ini:

a. urut-gabung (*Merge Sort*)
b. urut-sisip (*Insertion Sort*)

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

Tabel *A* dibagidua berdasarkan nilai elemennya. Misalkan elemen-elemen *A*1 ≤ elemen-elemen *A*2.

A

8	1	4	6	9	3	5	7

Divide: *A*1

5	1	4	3

*A*2

9	6	8	7

Sort: *A*1

1	3	4	5

*A*2

6	7	8	9

Combine: *A*

1	3	4	5	6	7	8	9

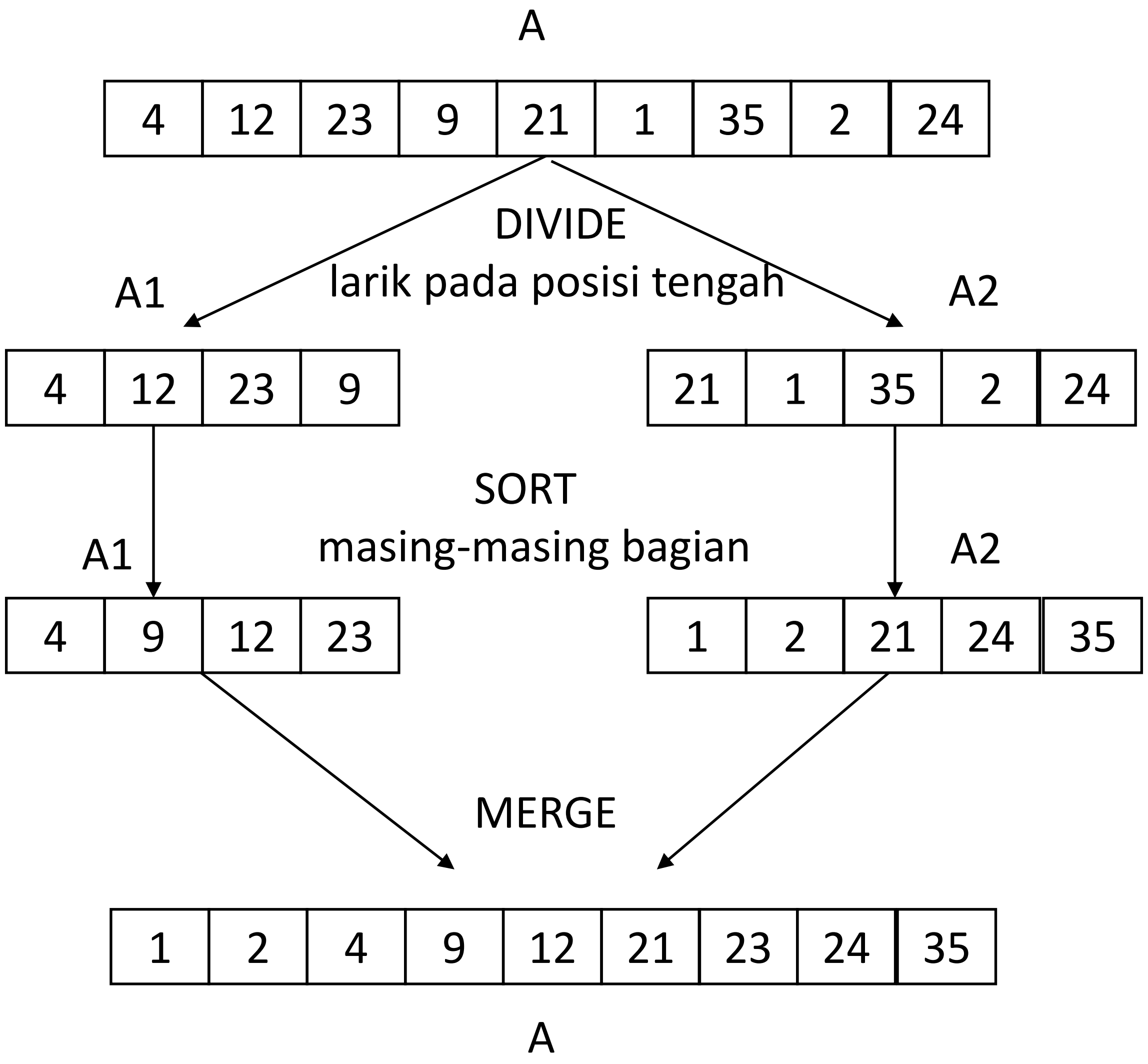
Algoritma pengurutan yang termasuk jenis ini:

a. urut-cepat *(Quick Sort)*

b. urut-seleksi *(Selection 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).

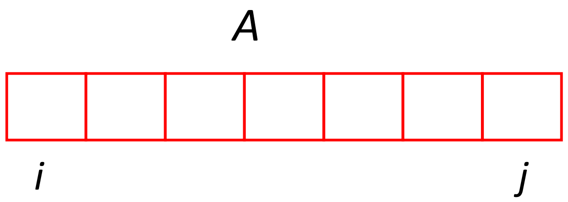
2. Jika n > 1, maka

   (a) DIVIDE: bagi larik A menjadi dua bagian pada posisi pertengahan, masing-masing bagian berukuran $n/2$ elemen.

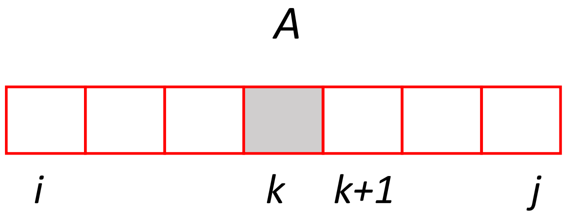
   (b) CONQUER: secara rekursif, terapkan *Merge Sort* pada masing-masing bagian.

   (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
}
Deklarasi
     k : integer

Algoritma:
  if   i  < j  then                    { ukuran(A) > 1 }
       k ← (i + j) div 2               { bagi A pada posisi pertengahan }
       MergeSort(A, i, k)              { urut upalarik A[i..k] }
       MergeSort(A, k + 1, j)          { urut upalarik A[k+1, j] }
       Merge(A, i, k, j)               { gabung hasil pengurutan A[i..k] dan A[k+1..j] menjadi A[i..j] }
  end
```

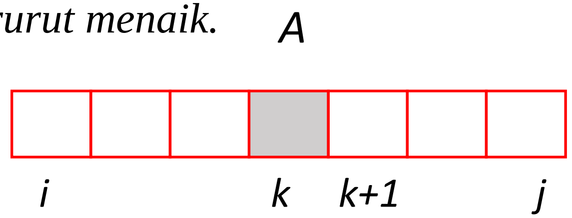


Pemanggilan pertama kali: *MergeSort*(*A*, 1, *n*)

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

*A*1			*A*2				*B*					
1	13	24	2	15	27	$1 < 2 \rightarrow 1$	1					
1	13	24	2	15	27	$2 < 13 \rightarrow 2$	1	2				
1	13	24	2	15	27	$13 < 15 \rightarrow 13$	1	2	13			
1	13	24	2	15	27	$15 < 24 \rightarrow 15$	1	2	13	15		
1	13	24	2	15	27	$24 < 27 \rightarrow 24$	1	2	13	15	24	
1	13	24	2	15	27	$27 \rightarrow$	1	2	13	15	24	27

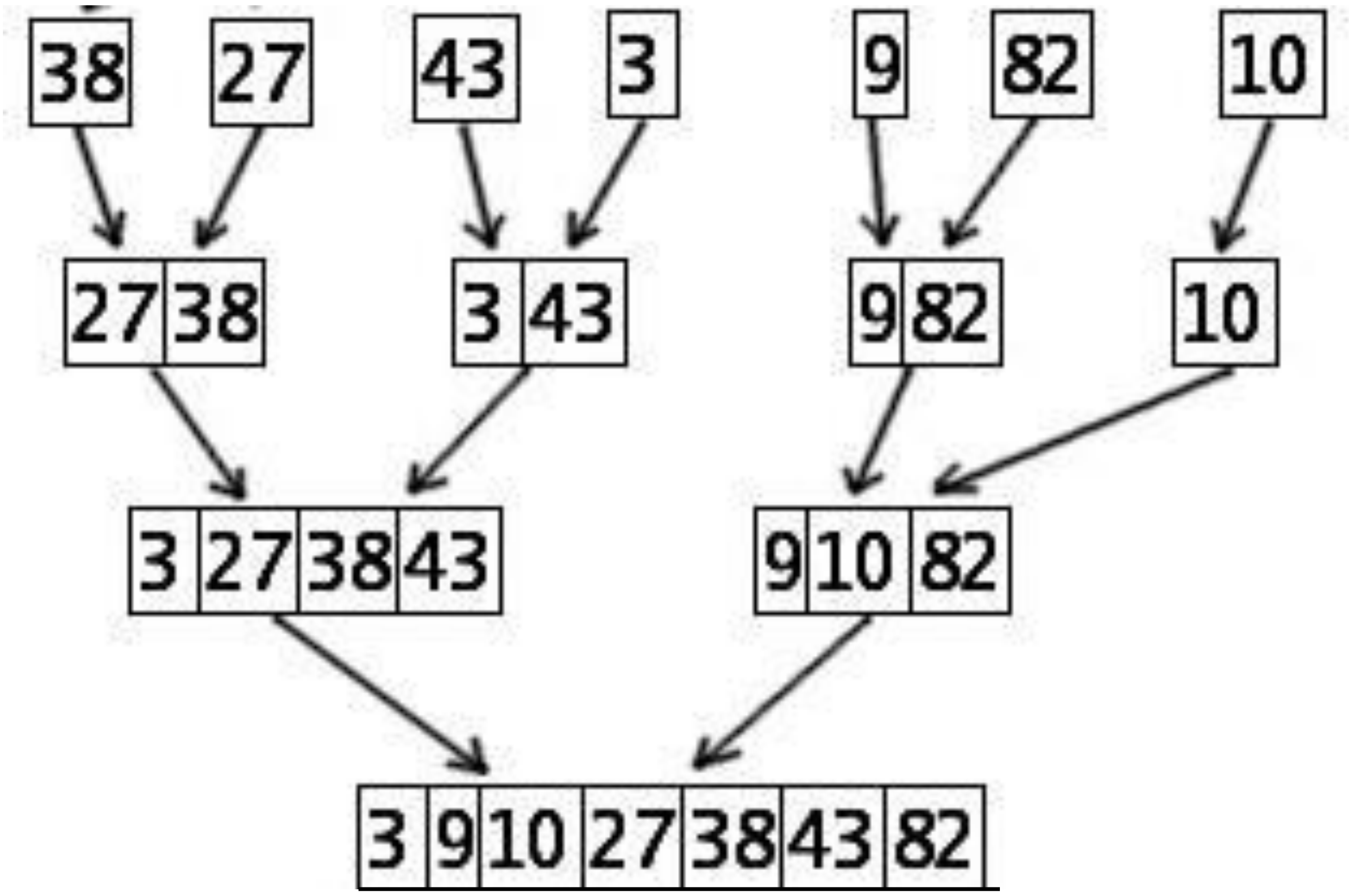
```
procedure Merge(input/output A : LarikInteger, input  i, k, j : integer)
{ Menggabung larikA[i..k] dan larik A[k+1..j] menjadi larik A[i..j] yang terurut menaik.
  Masukan: A[i..k] dan A[k+1..j] sudah terurut menaik.
  Luaran: A[k+1..j] yang terurut menaik.  }
```




```
Deklarasi
    B : LarikInteger        { larik temprorer untuk menyimpan hasil penggabungan }
    p, q, r : integer

Algoritma:
    p ← i                   { A[i .. k] }
    q ← k + 1               { A[k+1 .. j] }
    r ← i
    while (p ≤ k) and (q ≤ j) do
       if  A[p] ≤  A[q]  then
          B[r] ← A[p]        { salin elemen A[p] dari larik bagian kiri ke dalam larik B }
          p ← p + 1
      else
         B[r] ← A[q]         { salin elemen A[q] dari larik bagian kanan ke dalam larik B }
         q ← q + 1
      endif
      r ← r + 1
   endwhile
   { p  > k   or q  > j }
    …… continued
```

```
{ salin sisa larik A bagian kiri ke larik B, jika masih  ada }
while (p ≤ k) do
      B[r] ← A[p]
      p ← p + 1
      r ← r + 1
endwhile
{ p > k }

{ salin sisa larik A bagian kanan ke larik B, jika masih ada }
while (q ≤ j) do
      B[r] ← A[q]
      q ← q + 1
      r ← r + 1
endwhile
{ q > j }

{ salin kembali elemen-elemen larik B ke dalam A }
for r ← i to j do
    A[r] ← B[r]
endfor
{ diperoleh larik A yang terurut membesar }
```

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

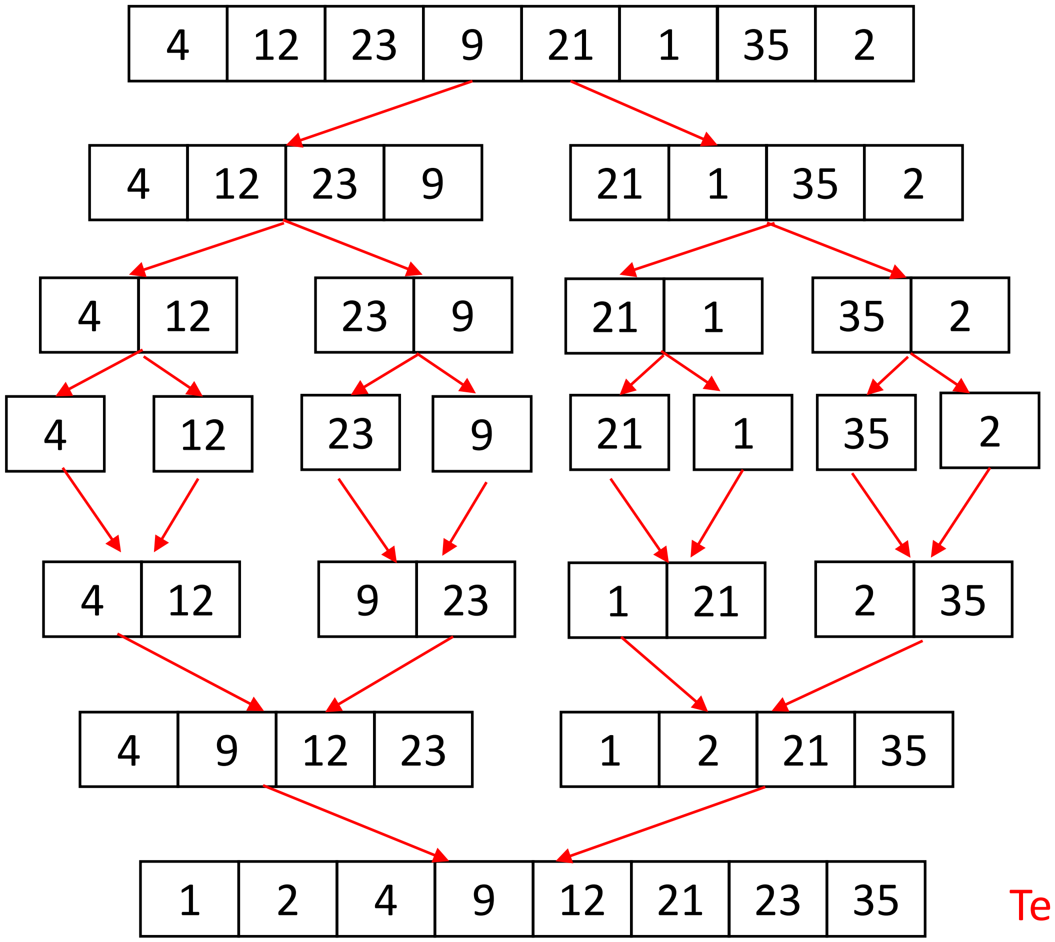


Contoh 4: Pengurutan larik A di bawah ini dengan *Merge Sort*

4	12	23	9	21	1	35	2

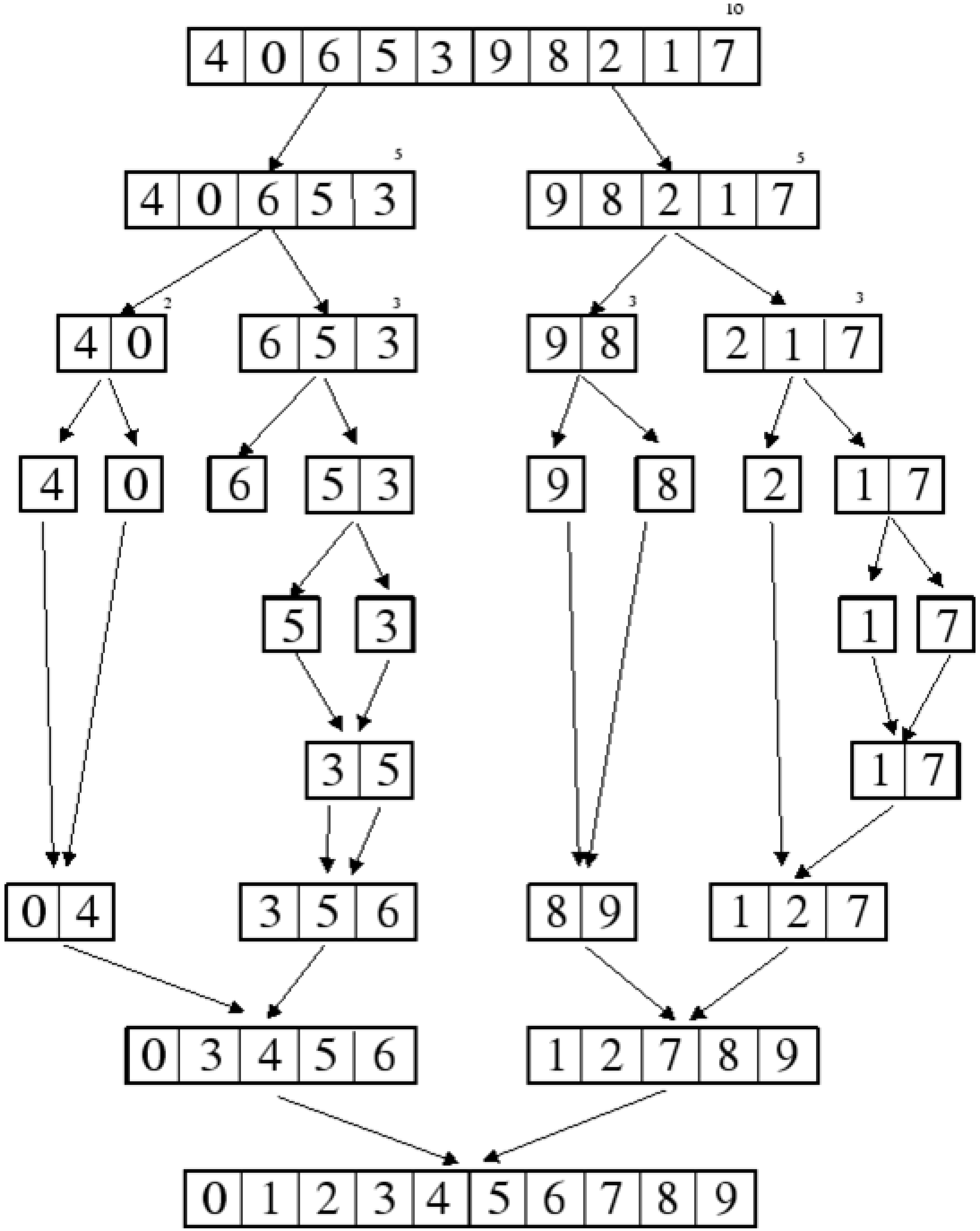
DIVIDE, CONQUER, dan SOLVE:

MERGE:



Terurut!

Contoh 5:


10
4 0 6 5 3 9 8 2 1 7
5
4 0 6 5 3
5
9 8 2 1 7
2
4 0
3
6 5 3
3
9 8
3
2 1 7
4
0
6
5 3
9
8
2
1 7
5
3
1
7
3 5
1 7
0 4
3 5 6
8 9
1 2 7
0 3 4 5 6
1 2 7 8 9
0 1 2 3 4 5 6 7 8 9

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:

$T(n)$ = *Mergesort* untuk pengurutan dua buah upalarik berukuran $n/2$ +
jumlah perbandingan elemen di dalam prosedur *Merge*
= $2T(n/2) + cn$

- 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 $n = 2^k$

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

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

  sehingga

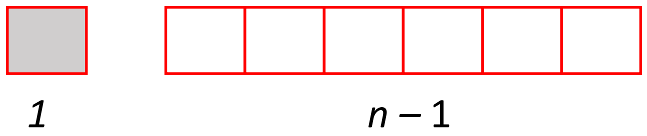
  $T(n) = nT(1) + cn\,{}^2\log n = an + cn\,{}^2\log n = O(n\,{}^2\log n)$

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

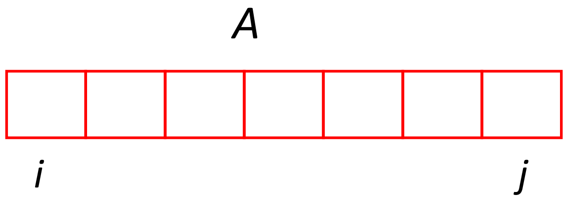
4.2 *Insertion Sort*

- *Insertion Sort* adalah pengurutan *easy split/hard join* dengan cara membagi larik menjadi dua buah upalarik yang tidak sama ukurannya,

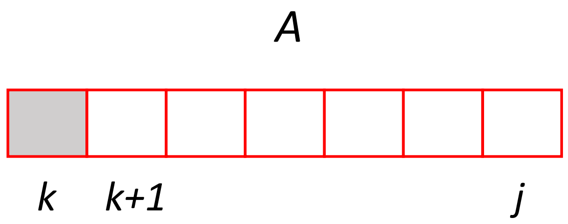
- yaitu, upalarik pertama hanya satu elemen, sedangkan upalarik kedua berukuran $n - 1$ elemen.



- *Insertion Sort* dapat dipandang sebagai kasus khusus dari *Merge Sort* dengan hasil pembagian terdiri dari 1 elemen dan $n - 1$ elemen.




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

Deklarasi:
   k : integer

Algoritma:
    if  i  < j  then                    { ukuran(A) > 1 }
        k ← i                           { bagi A pada posisi elemen pertama }
        InsertionSort(A, i, k)          { urut upalarik A[i..k] }
        InsertionSort(A, k + 1, j)      { urut upalarik A[k+1, j] }
        Merge(A, i, k, j)               { gabung hasil pengurutan A[i..k] dan A[k+1..j] menjadi A[i..j] }
   end
```



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

- **Perbaikan**: karena upalarik pertama hanya berisi satu elemen, maka kita tidak perlu melakuan pemanggilan rekursif untuk upalarik ini. Algoritma menjadi sbb:

```
procedure InsertionSort(input/output A : LarikInteger, input  i, j : integer)
{ Mengurutkan larik A[i..j] dengan algoritma Insertion Sort.
  Masukan: Larik A[i..j] yang sudah terdefinisi elemen-elemennya
  Luaran: Larik A[i..j] yang terurut
}
Deklarasi:
   k : integer

Algoritma:
    if   i  < j  then                     { ukuran(A) > 1 }
        k ← i                             { bagi A pada posisi elemen pertama }
        InsertionSort(A, k + 1, j)        { urut upalarik A[k+1, j] }
        Merge(A, i, k, j)                 { gabung hasil pengurutan A[i..k] dan A[k+1..j] menjadi A[i..j] }
   end
```

- Algoritma di atas dapat dianggap sebagai versi rekursif algoritma *Insertion Sort*
- Selain menggunakan prosedur *Merge,* kita dapat mengganti *Merge* dengan prosedur penyisipan sebuah elemen pada larik yang terurut (seperti pada algoritma *Insertion Sort* versi iteratif).

Contoh 6 (Insertion Sort): Misalkan larik *A* berisi elemen-elemen berikut:

4	12	23	9	21	1	5	2

DIVIDE, CONQUER, dan SOLVE:

4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2
4	12	3	9	1	21	5	2

MERGE:

4	12	3	9	1	21	5	2
4	12	3	9	1	21	2	5
4	12	3	9	1	2	5	21
4	12	3	9	1	2	5	21
4	12	3	1	2	5	9	21
4	12	1	2	3	5	9	21
4	1	2	3	5	9	12	21
1	2	3	4	5	9	12	21

Kompleksitas waktu algoritma *Insertion Sort*:

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

Penyelesaian:

$T(n) = cn + T(n-1)$
$= cn + \{ c(n-1) + T(n-2) \}$
$= cn + c(n-1) + \{ c(n-2) + T(n-3) \}$
$= cn + c(n-1) + c(n-2) + \{c(n-3) + T(n-4) \}$
$= \ldots$
$= cn + c(n-1) + c(n-2) + c(n-3) + \ldots + c2 + T(1)$
$= c\{ n + (n-1) + (n-2) + (n-3) + \ldots + 2 \} + a$
$= \mathrm{c}\{ (n-1)(n+2)/2 \} + a$
$= cn^2/2 + cn/2 + (a - c)$
$= O(n^2) \rightarrow$ sama seperti komplesitas algoritma *Insertion Sort* versi iteratif

4.3 *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, *A*1 dan *A*2, sedemikian sehingga:

semua elemen di *A*1 $\leq$ semua elemen di *A*2.

A	8	1	4	6	9	3	5	7

Divide:

*A*1	5	1	4	3
*A*2	9	6	8	7

Sort:

*A*1	1	3	4	5
*A*2	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 $x \in \{ A[1], A[2], ..., A[n] \}$ 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 7. 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

awal pindai $\rightarrow$ awal pindai $\leftarrow$

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

$\uparrow p$ $\uparrow q$

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

$\uparrow$ $\uparrow$

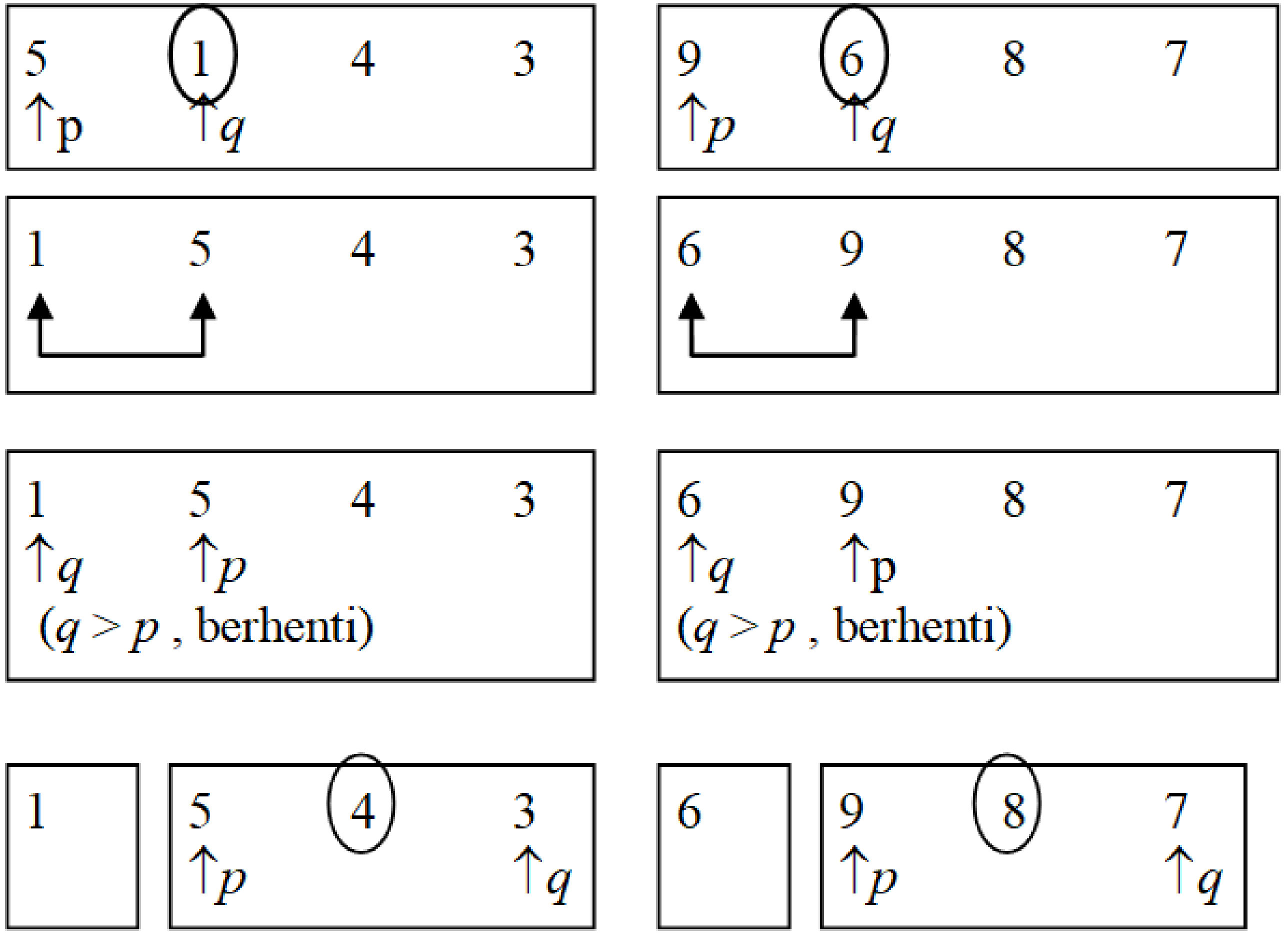
tukar

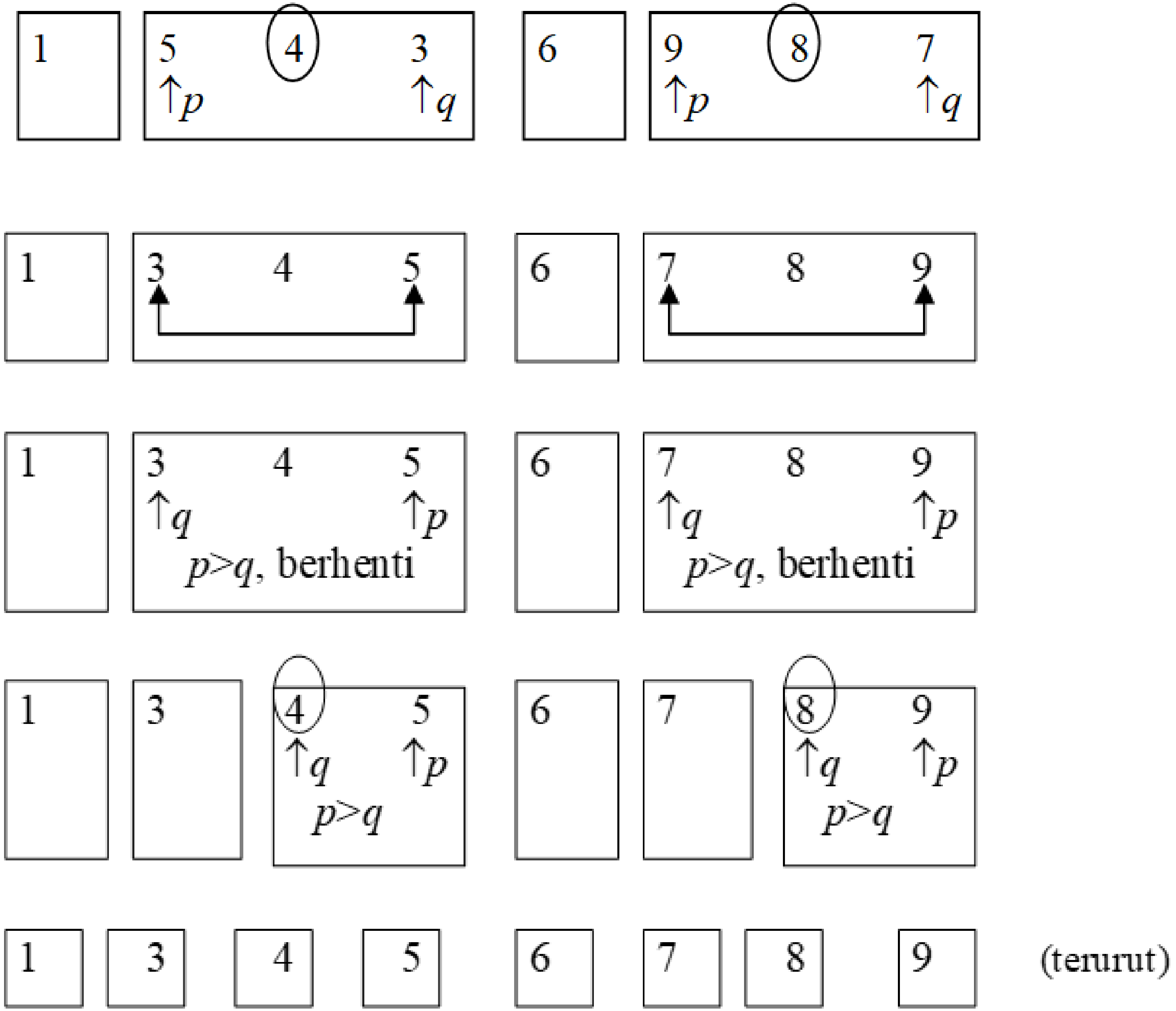
		awal pindai $\rightarrow$				awal pindai $\leftarrow$		
(ii) & (iii):	5	1	4	6	9	3	8	7
				$\uparrow p$		$\uparrow q$		
(iv):	5	1	4	3	9	6	8	7
				$\uparrow$	tukar	$\uparrow$		
					$\leftarrow \rightarrow$			
(ii) & (iii):	5	1	4	3	9	6	8	7
				$\uparrow q$	$\uparrow p$	($q < p$, berhenti)		

Hasil partisi pertama:

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

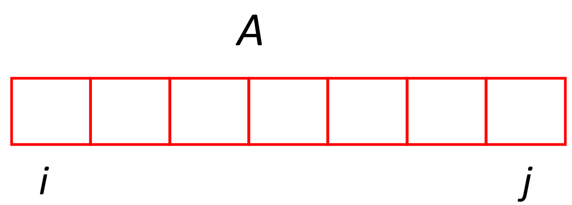
Teruskan partisi untuk setiap bagian sampai berukuran satu elemen:



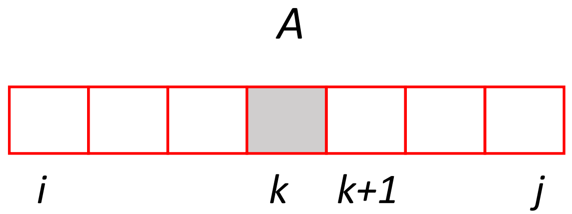


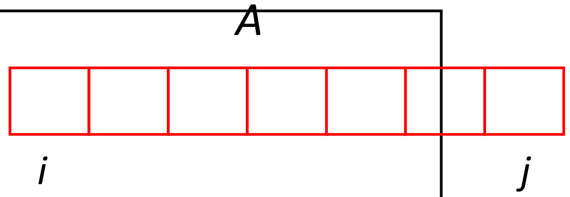
1
5 4 3
↑p ↑q
6
9 8 7
↑p ↑q
1
3 4 5
6
7 8 9
1
3 4 5
↑q ↑p
p>q, berhenti
6
7 8 9
↑q ↑p
p>q, berhenti
1
3
4 5
↑q ↑p
p>q
6
7
8 9
↑q ↑p
p>q
1 3 4 5 6 7 8 9
(terurut)

- *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
}

Deklarasi
   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]  }
Deklarasi
   pivot, temp : integer
Algoritma:
  pivot ← pilih sembarang elemen larik sebagai pivot, misalkan pivot = elemen tengah
  p ← i        {awal pemindaian dari kiri }
  q ← j        { awal pemindaian dari kanan }
  repeat
    while A[p] < pivot  do
        p ← p + 1
    endwhile
    { A[p] ≥ pivot}
     while  A[q] > pivot  do
         q ← q − 1
    endwhile
    { A[q] ≤ pivot}
     if  p  < q  then
       swap(A[p], A[q])      {pertukarkan A[p] dengan A[q] }
       p ← p + 1              {awal pemindaian berikutnya dari kiri }
       q ← q − 1             {awal pemindaian berikutnya dari kanan }
    endif
  until  p  ≥ q
```

Versi kedua *Quicksort*: Partisi sedemikian rupa sehingga elemen-elemen larik kiri ≤ pivot dan elemen-elemen larik kanan ≥ dari pivot.

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

p = pivot = elemen pertama.

Contoh:

$$\begin{array}{l} \text{pivot} \\ \downarrow \\ \textcolor{red}{\mathbf{5}}, 3, 1, 9, 8, 2, 4, 7 \end{array}$$

Partisi

$$\begin{array}{c} \text{pivot} \\ \downarrow \\ \underbrace{2,\ 3, 1, 4,}_{\text{semua} \leq \text{pivot}}\ \textcolor{red}{\mathbf{5}}, \underbrace{8, 9, 7}_{\text{semua} \geq \text{pivot}} \end{array}$$

Contoh 8 (Levitin, 2003):

	→					←	
5	3	1	9	8	2	4	7
			↑p			↑q	
5	3	1	4	8	2	9	7
			tukar				
5	3	1	4	8	2	9	7
				↑p	↑q		
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
				↑q	↑p	→ stop sebab $p > q$	
2	3	1	4	**5**	8	9	7

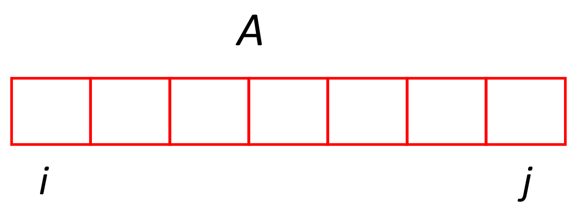
tukar posisi pivot dengan $A[q]$

2	3	1	4	**5**	8	9	7

	→		←			→	←
2	3	1	4		**8**	9	7
	↑*p*	↑*q*				↑*p*	↑*q*
2	1	3	4		**8**	7	9
2	1	3	4		**8**	7	9
	↑*q*	↑*p*				↑*q*	↑*p*
1	**2**	3	4		7	**8**	9
1		**3**	4		7		9
		↑*q*	↑*p*				
			4				

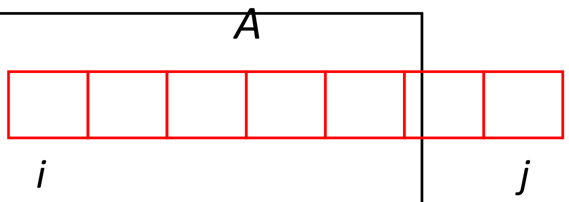
Terurut:

1	**2**	**3**	**4**	**5**	**7**	**8**	**9**



- *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
}

Deklarasi
   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 ← A[i]          { pivot = elemen pertama }
   p ← i                 {awal pemindaian dari kiri }
   q ← j + 1             { awal pemindaian dari kanan }
   repeat
      repeat
           p ← p + 1
      until  A[p] >= pivot

      repeat
           q ← q – 1
      until  A[q] <= pivot

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

   until  p ≥  q
   swap(A[p], A[q])    { undo last swap when p ≥ 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.

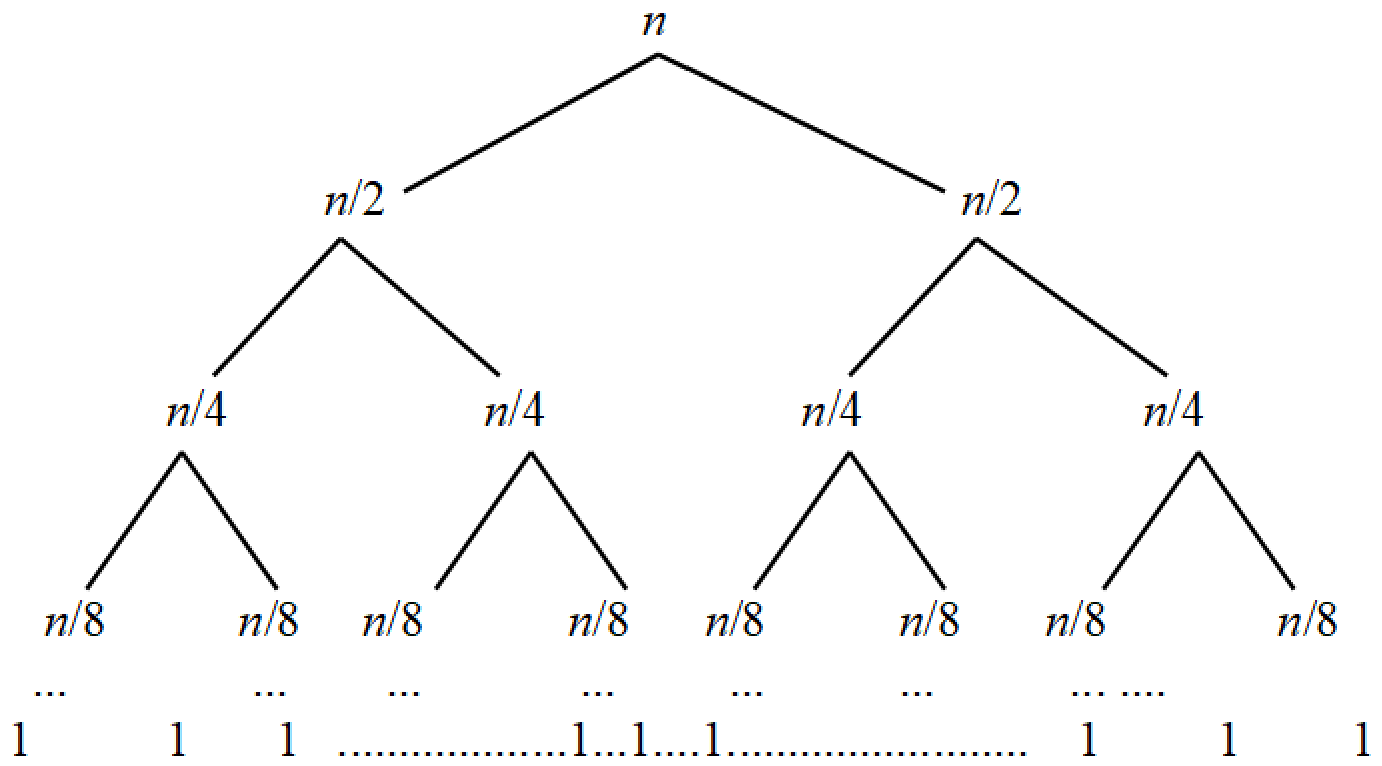
    3. *Pivot* = elemen median larik

- 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 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*:

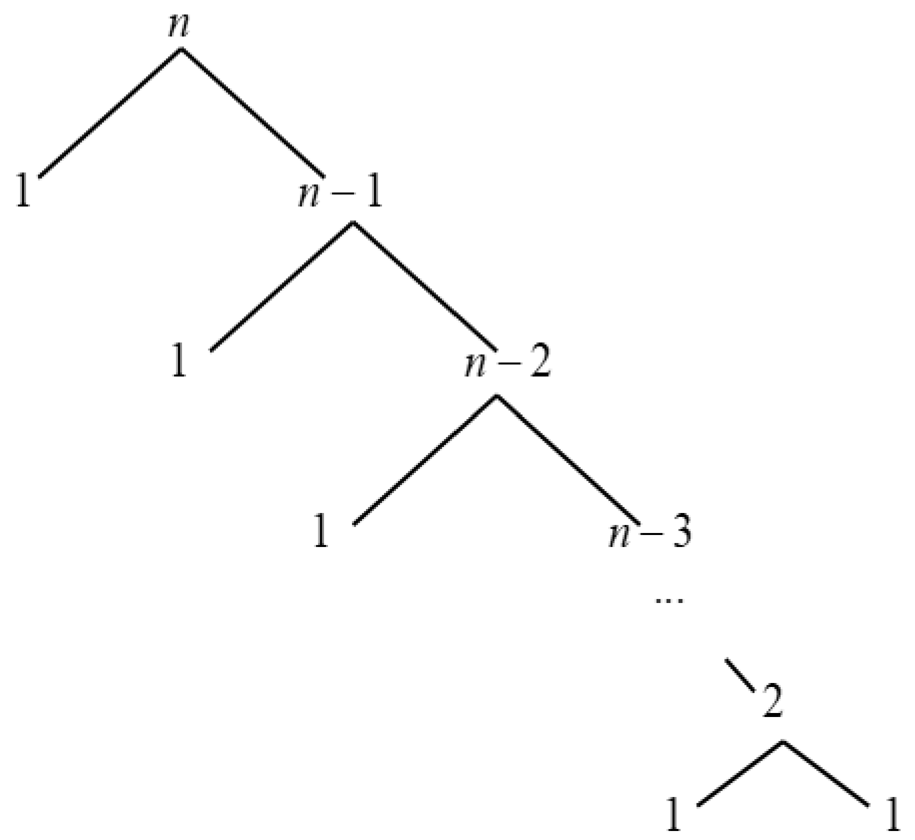
$T(n) = 2T(n/2) + cn = na + \mathrm{cn}\ {}^2\log n = O(n\ {}^2\log n)$.

- 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:

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

Penyelesaiannya sama seperti pada *Insertion Sort*:

$T(n) = T(n - 1) + cn = O(n^2)$.

- Kasus terburuk menghasilkan kompleksitas algoritma *Quicksort* yang sama dengan algoritma pengurutan secara *brute force.*

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.

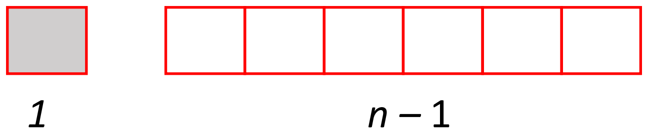
- Kompleksitas algoritma *Quicksort* untuk kasus rata-rata:

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

4.4 Selection Sort

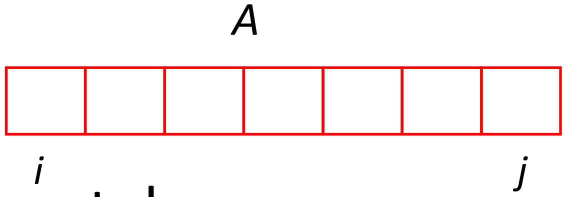
- *Selection Sort* adalah pengurutan *hard split/easy join* dengan cara membagi larik menjadi dua buah upalarik yang tidak sama ukurannya,

- yaitu, upalarik pertama hanya satu elemen, sedangkan upalarik kedua berukuran $n - 1$ elemen.



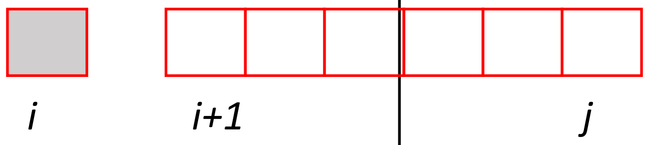
- *Selection Sort* dapat dipandang sebagai kasus khusus dari *Quick Sort* dengan hasil pembagian terdiri dari 1 elemen dan $n - 1$ elemen, namun proses partisinya dilakukan dengan cara berbeda.

- Proses partisi di dalam *Selection Sort* dilakukan dengan mencari elemen bernilai minimum (atau bernilai maksimum) di dalam larik *A*[*i*..*j*]



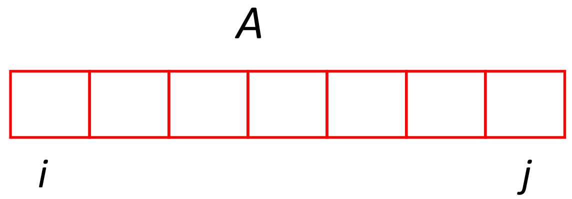
- lalu elemen minimum ditempatkan pada posisi *A*[*i*] dengan cara pertukaran.

```
procedure SelectionSort(input/output A : LarikInteger, input  i, j : integer)
{ Mengurutkan larik A[i..j] dengan algoritma Selection Sort
  Masukan: Larik A[i..j] yang sudah terdefinisi elemen-elemennya
  Luaran: Larik A[i..j] yang terurut
}
Deklarasi
   k : integer

Algoritma:
    if  i < j  then                   { Ukuran(A) > 1 }
        Partisi3(A, i, j)             { Partisi menjadi 1 elemen dan n – 1 elemen }
        SelectionSort(A, i+1, j)   { Urut hanya upalarik A[i+1..j] dengan Selection Sort }
    endif
```



- Algoritma di atas dapat dianggap sebagai versi rekursif algoritma *Selection Sort*

procedure *Partisi3*(**input/output** *A* : *LarikInteger*, **input** *i*, *j* : **integer**)
{ Menmpartisi larik A[i..j] dengan cara mencari elemen minimum di dalam A[i..j], dan menempatkan elemen terkecil sebagai elemen pertama larik.
Masukan: A[i..j] sudah terdefinisi elemen-elemennya
Luaran: A[i..j] dengan A[i] adalah elemen minimum.
}



Deklarasi
    idxmin, *k* : **integer**

Algoritma:
  idxmin ← i
  for *k* ← *i*+1 **to** *j* **do**
      if *A*[*k*] < *A*[*idxmin*] **then**
          idxmin ← *k*
      endif
  endfor
  swap(*A*[*i*], *A*[*idxmin*])   *{ pertukarkan A[i] dengan A[idxmin] }*

Contoh 9. Misalkan tabel A berisi elemen-elemen berikut:

4 12 3 9 1 21 5 2

Langkah-langkah pengurutan dengan *Selection Sort*:

4	12	3	9	1	21	5	2	
1	12	3	9	4	21	5	2	
1	2	3	9	4	21	5	12	
1	2	3	9	4	21	5	12	
1	2	3	4	9	21	5	12	
1	2	3	4	5	21	9	12	
1	2	3	4	5	9	12	21	
1	2	3	4	5	9	12	21	
1	2	3	4	5	9	12	21	→ terurut!

Kompleksitas waktu algoritma *Selection Sort*:

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

Penyelesaiannya sama seperti pada *Insertion Sort*:

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

Moral of the story: pembagian larik menjadi dua buah upalarik yang seimbang (masing-masing $n/2$) akan menghasiljan kinerja algoritma yang terbaik (pada kasus *Merge Sort* dan *Quicksort*, $O(n \log n)$), sedangkan pembagian yang tidak seimbang (masing-masing 1 elemen dan $n - 1$ elemen) menghasilkan kinerja algoritma yang buruk (pada kasus *insertion sort* dan *selection sort*, $O(n^2)$)

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 $n = b^k$, $k = 1, 2, \ldots$, $a \geq 1$, $b \geq 2$, c 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: 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 $a = b^d$)

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

sehingga

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

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

Contoh 11: 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 $a = b^d$)

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

sehingga

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

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:

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

   Return: "no majority"

   Contoh: <u>4 3 4 2 7 5 2 1</u>

<u>4 3 4 2</u> <u>7 5 2 1</u>

Ingat definisi mayoritas! → no majority   no majority

<u>4 3 4 2 7 5 2 1</u>

"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

Ingat definisi mayoritas!

4 3 4 2 7 4 4 4

Jumlah elemen 4 = 5 buah → mayoritas

Ingat definisi mayoritas!

"majority = 4"

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"

Contoh: 3 3 4 3 7 3 3 4

3 3 4 3   7 3 3 4

majority = 3   no majority

3 3 4 3 7 3 3 4

Jumlah elemen 3 = 5 buah → mayoritas

"majority = 3"

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: 3 3 4 3 4 4 4 4

3 3 4 3   4 4 4 4

majority = 3   majority = 4

3 3 4 3 4 4 4 4

Jumlah elemen 3 = 3 buah

Jumlah elemen 4 = 5 buah → mayoritas

"majority = 4"

Contoh keseluruhan:

4 3 4 4 4 5 4 3

4 3 4 4 | 4 5 4 3

4 3 | 4 4 | 4 5 | 4 3

4 | 3 | 4 | 4 | 4 | 5 | 4 | 3

divide (sekaligus conquer)

4 | 3 | 4 | 4 | 4 | 5 | 4 | 3

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

solve

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 $a = b^d$)

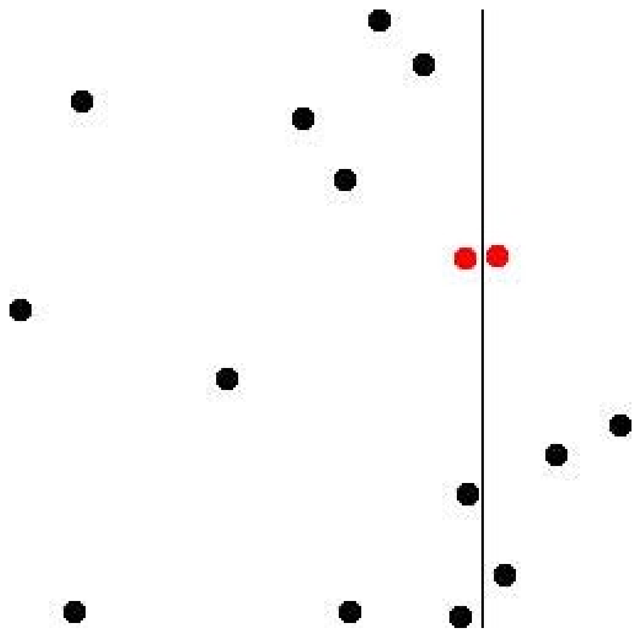
$$\begin{cases} O(n^d) & \text{jika } a < b^d \\ O(n^d \log n) & \text{jika } a = b^d \\ O(n^{\log_b a}) & \text{jika } a > b^d \end{cases}$$

sehingga

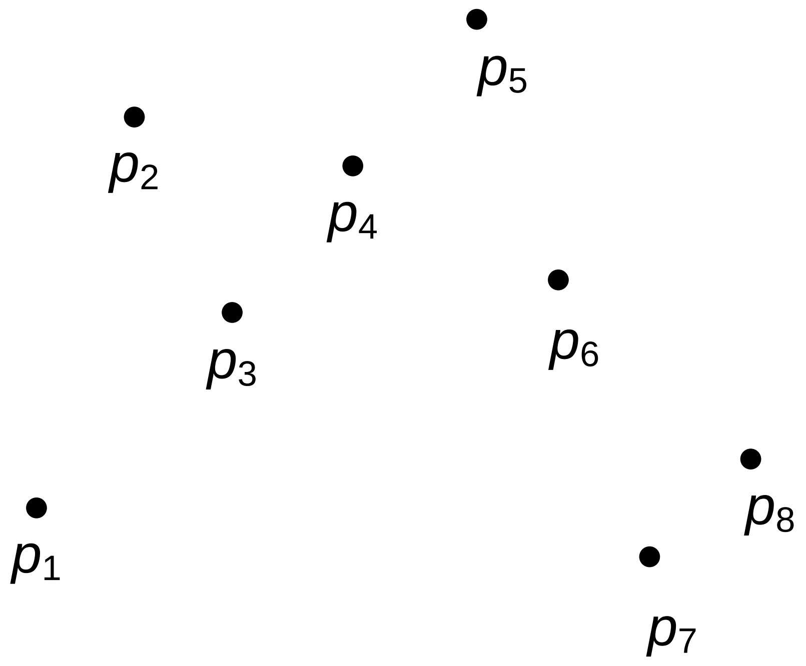
$T(n) = O(n^1 \log n) = O(n \log n)$

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, ..., 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 $O(n^2)$.

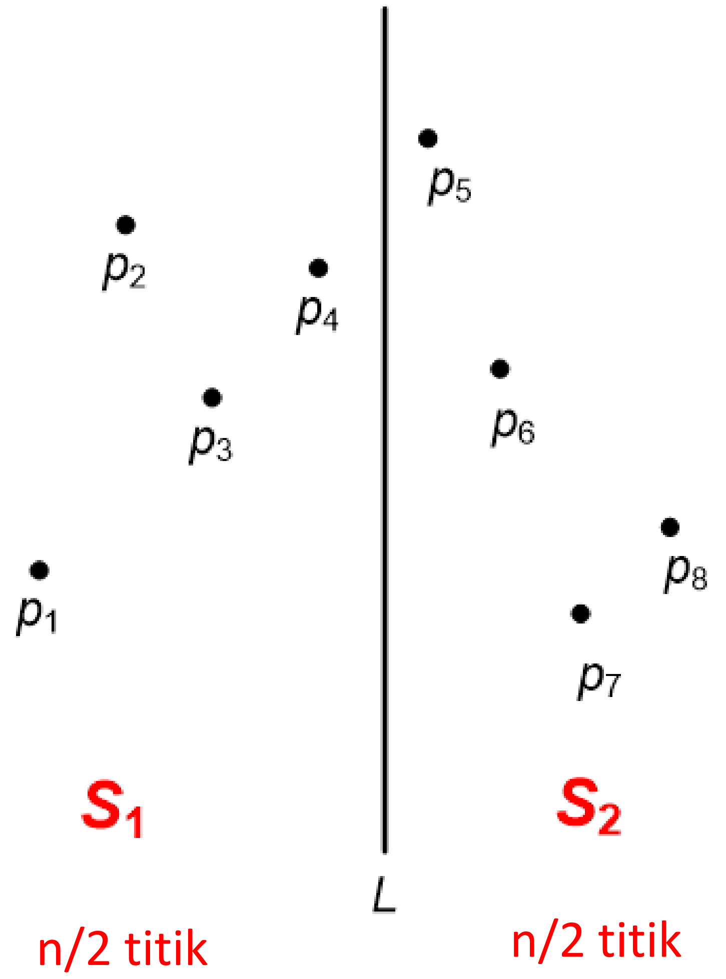
Penyelesaian secara *Divide and Conquer*

- 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.

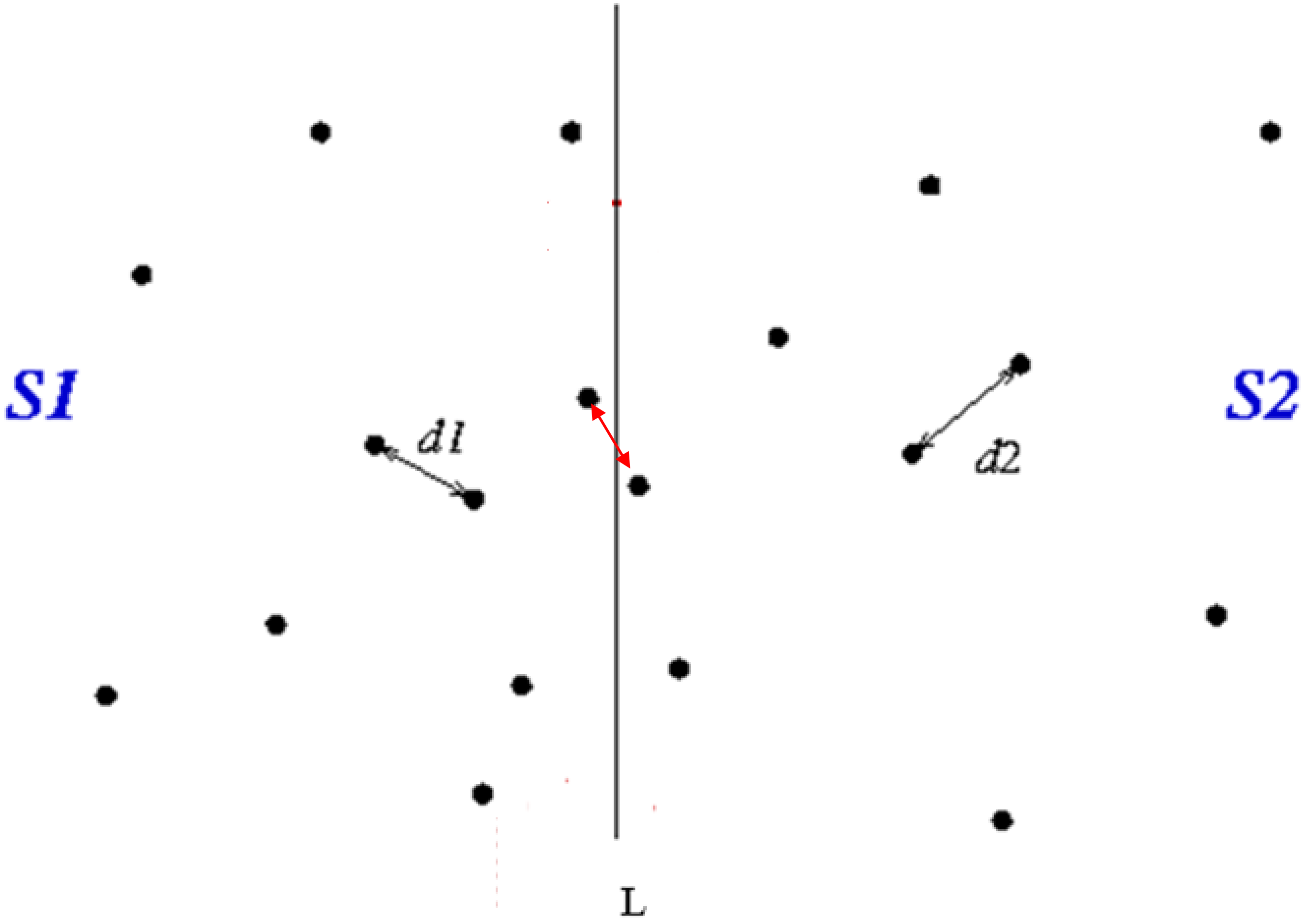
Garis maya L dapat dihampiri sebagai $y = x_{n/2}$ (ingatlah titik-titik sudah diurut menaik berdasarkan absis (x)).



3. CONQUER: Secara rekursif, terapkan algoritma *D-and-C* pada masing-masing bagian untuk mencari sepasang titik terdekat.

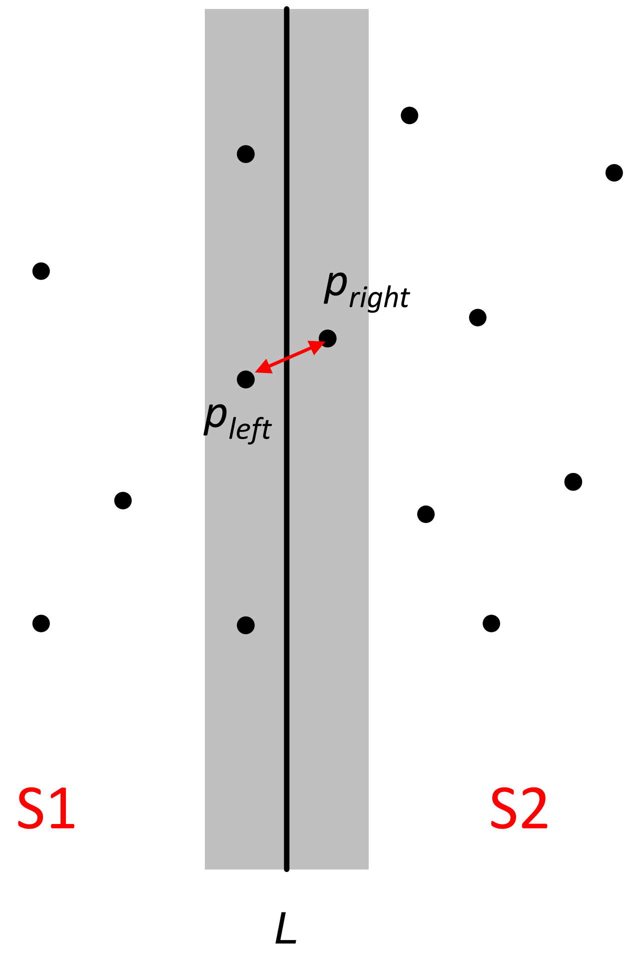
4. COMBINE: Pasangan titik yang jaraknya terdekat ada tiga kemungkinan letaknya:
   (a) Pasangan titik terdekat terdapat di dalam bagian S_1.
   (b) Pasangan titik terdekat terdapat di dalam bagian S_2.
   (c) Pasangan titik terdekat dipisahkan oleh garis batas L, yaitu satu titik di S_1 dan satu titik di S_2.

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


S1
d1
d2
S2
L

```
procedure FindClosestPair(input P : SetOfPoint, n : integer, output d : real)
{ Mencari jarak terdekat sepasang titik di dalam himpunan P
  Masukan: P = {p1, p2, ..., pn}, titik-titik di dalam P sudah terurut menaik berdasarkan absisnya (x)
  Luaran: d adalah jarak sepasang titik terdekat }

Deklarasi:
  d1, d2 : real

Algoritma:
  if n = 2 then
     d ← jarak kedua titik dengan rumus Euclidean
  else
    S1 ← {p1, p2 ,..., pn/2 }
    S2 ← {p n/2+1, pn/2+2 ,..., pn }
    FindClosestPair(S1, n/2, d1)
    FindClosestPair(S2, n/2, d2)
    d ← MIN(d1, d2)       { bandingkan dulu d1 dengan d2 untuk menentukan yang terkecil }
    {********************************************************************************}
     Tentukan apakah terdapat titik pleft di S1 dan pright di S2 dengan jarak(pleft, pright) < 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?



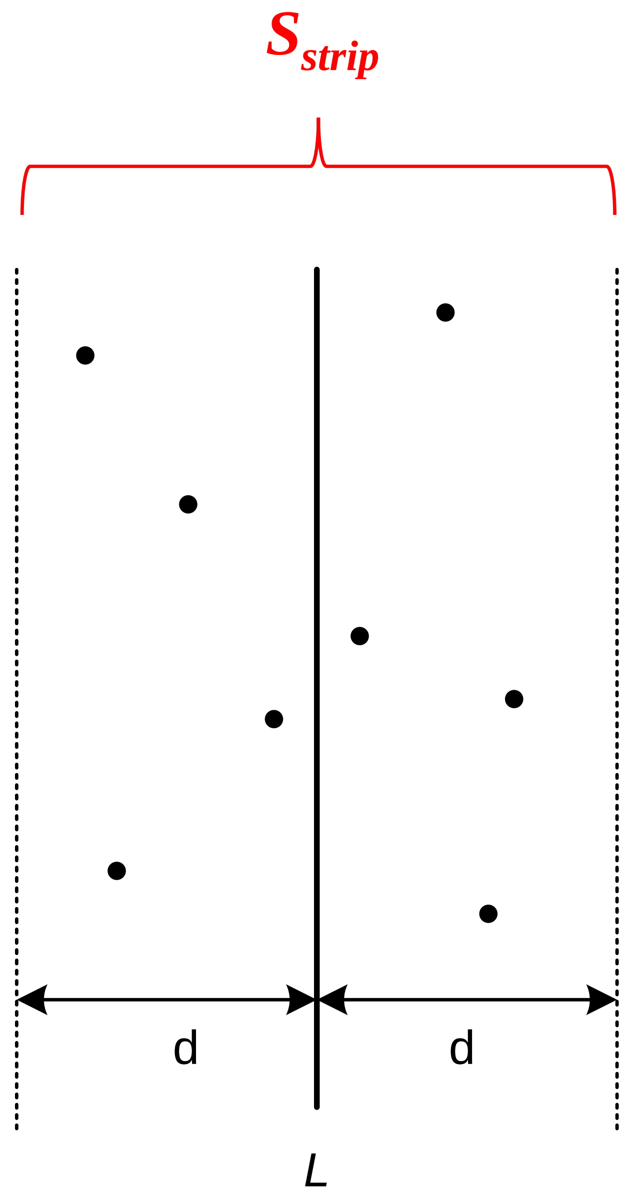
- Kita membatasi titik-titik di dalam *strip* selebar 2*d*

- Oleh karena itu, implementasi tahap ketiga adalah sbb:

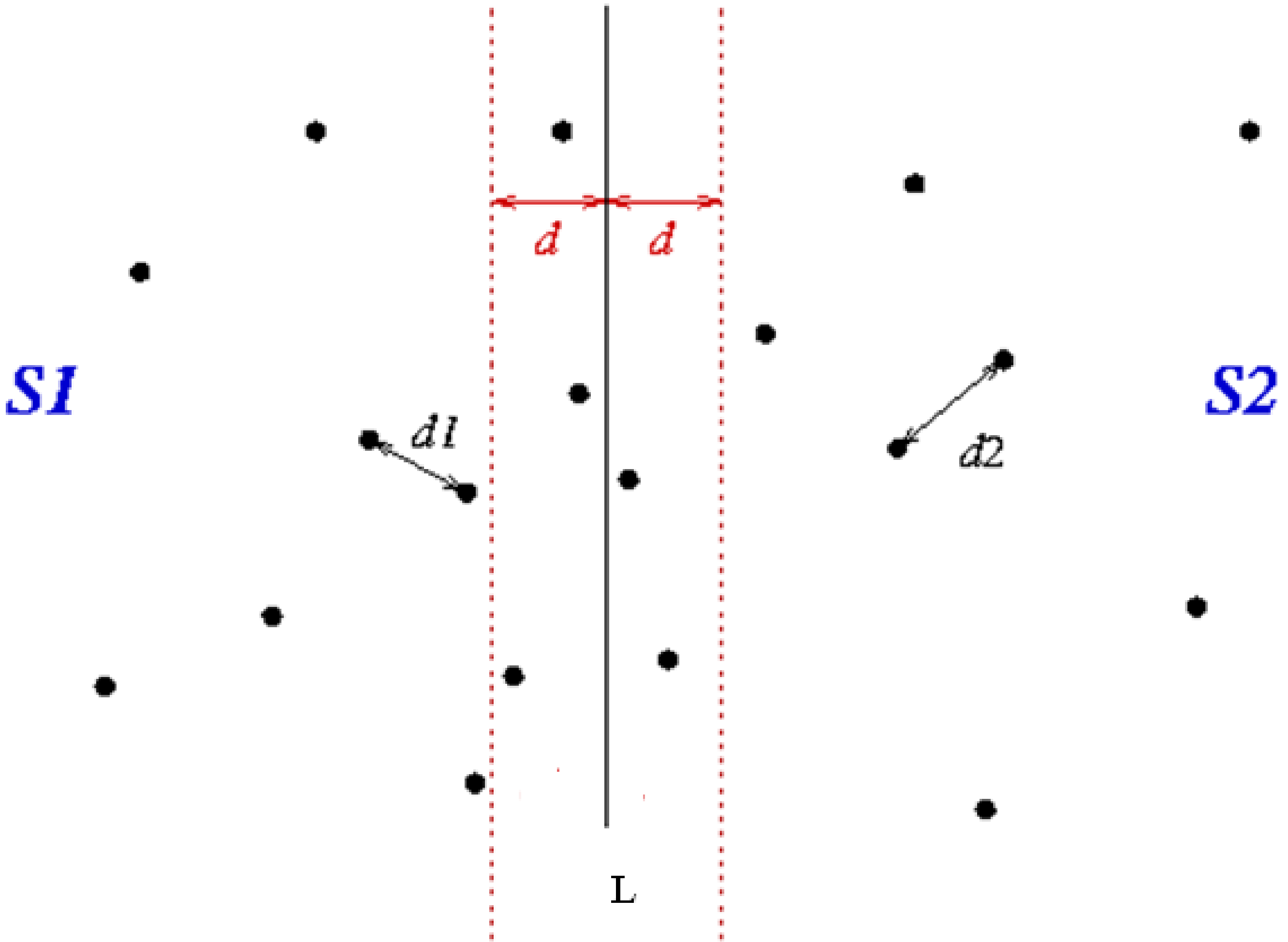
  (i) Temukan semua titik di *S1* yang memiliki absis *x* minimal $x_{n/2} - d$.

  (ii) Temukan semua titik di *S2* 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.



S_{strip}



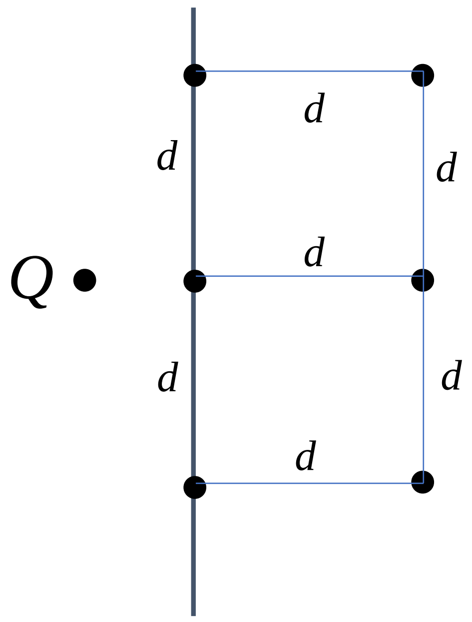
Keterangan: d = MIN($d1$, $d2$)

- Urutkan titik-titik di dalam S_{strip} dalam urutan ordinat *y* yang menaik. Misalkan q_1, q_2 , ..., 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←1 to s do
   for  j←i+1 to s do
      if (ABS(qi.x – qj.x ) > d or  ABS(qi.y – qj.y) > d then
         { tidak diproses  }
      else
         d3 ← EUCLIDEAN(qi, qj)   { hitung jarak qi dan qj dengan rumus Euclidean }
         if  d3 < d then          { bandingkan apakah d3 lebih kecil dari d }
            d ← 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 $T(n) = O(n \log n)$
$\rightarrow$ Lebih baik dari algoritma *brute force* yang $O(n^2)$

BERSAMBUNG