



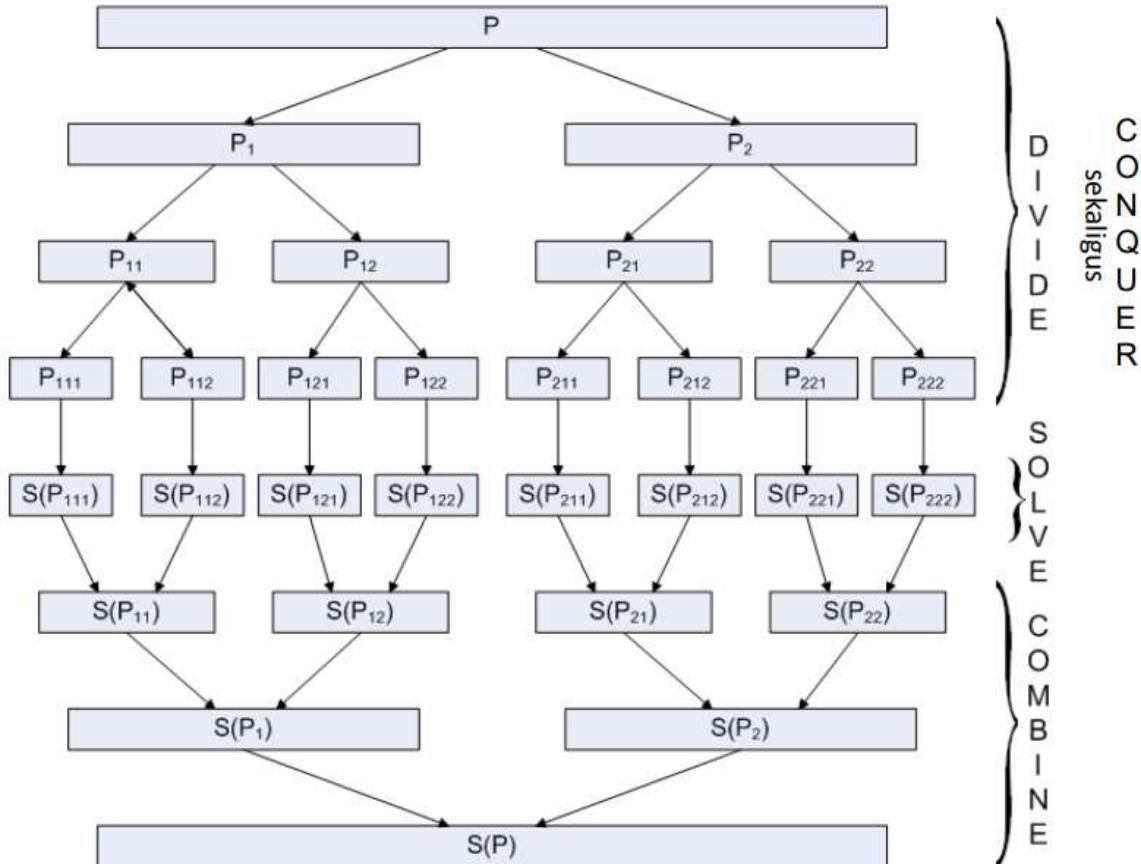
ANALISIS ALGORITMA

Ir. Rismayani, S.Kom., M.T

Definisi Divide and Conquer

- ▶ Divide: membagi persoalan menjadi beberapa upa-persoalan yang memiliki kemiripan dengan persoalan semula namun berukuran lebih kecil (idealnya berukuran hampir sama)
- ▶ Conquer (solve): menyelesaikan masing-masing upa-persoalan (secara langsung jika sudah berukuran kecil atau secara rekursif jika masih berukuran besar).
- ▶ Combine: mengabungkan solusi masing-masing upa-persoalan sehingga membentuk solusi persoalan semula.

LANJUTAN



Keterangan:
 P = persoalan
 S = solusi

LANJUTAN

- ▶ Obyek persoalan yang dibagi :
- ▶ masukan (input) atau instances persoalan yang berukuran n seperti:
 - ▶ - tabel (larik),
 - ▶ - matriks,
 - ▶ - eksponen,
 - ▶ - polinom,
 - ▶ - dll, bergantung persoalannya.
- ▶ Tiap-tiap upa-persoalan memiliki karakteristik yang sama (the same type) dengan karakteristik persoalan semula
- ▶ sehingga metode Divide and Conquer lebih natural diungkapkan dalam skema rekursif.

Skema Umum Algoritma Divide and Conquer

```
procedure DIVIDEandCONQUER(input P : problem, n : integer)
{ Menyelesaikan persoalan P dengan algoritma divide and conquer
  Masukan: masukan persoalan P berukuran n
  Luaran: solusi dari persoalan semula }
```

Deklarasi

r : **integer**

Algoritma

```
if n ≤ n0 then {ukuran persoalan P sudah cukup kecil}
  SOLVE persoalan P yang berukuran n ini
else
  DIVIDE menjadi r upa-persoalan, P1, P2, ..., Pr, yang masing-masing berukuran n1, n2, ..., nr
  for masing-masing P1, P2, ..., Pr, do
    DIVIDEandCONQUER(Pi, ni)
  endfor
  COMBINE solusi dari P1, P2, ..., Pr menjadi solusi persoalan semula
endif
```

Kompleksitas algoritma *divide and conquer*: $T(n) = \begin{cases} g(n) & , n \leq n_0 \\ T(n_1) + T(n_2) \dots + T(n_r) + f(n) & , n > n_0 \end{cases}$

LANJUTAN

Penjelasan:

$$T(n) = \begin{cases} g(n) & , n \leq n_0 \\ T(n_1) + T(n_2) \dots + T(n_r) + f(n) & , n > n_0 \end{cases}$$

- $T(n)$: kompleksitas waktu penyelesaian persoalan P yang berukuran n
- $g(n)$: kompleksitas waktu untuk SOLVE jika n sudah berukuran kecil
- $T(n_1) + T(n_2) \dots + T(n_r)$: kompleksitas waktu untuk memproses setiap upa-persoalan
- $f(n)$: kompleksitas waktu untuk COMBINE solusi dari masing-masing upa-persoalan

LANJUTAN

Jika pembagian selalu menghasilkan dua upa-persoalan yang berukuran sama:

```
procedure DIVIDEandCONQUER(input P : problem, n : integer)
{ Menyelesaikan persoalan dengan algoritma divide and conquer
  Masukan: masukan yang berukuran n
  Luaran: solusi dari persoalan semula
}
Deklarasi
  r : integer

Algoritma
  if n ≤ n0 then {ukuran persoalan sudah cukup kecil }
    SOLVE persoalan P yang berukuran n ini
  else
    DIVIDE menjadi 2 upa-persoalan, P1 dan P2, masing-masing berukuran n/2
    DIVIDEandCONQUER(P1, n/2)
    DIVIDEandCONQUER(P2, n/2)
    COMBINE solusi dari P1 dan P2
  endif
```

Kompleksitas algoritma *divide and conquer*: $T(n) = \begin{cases} g(n) & , n \leq n_0 \\ 2T(n/2) + f(n) & , n > n_0 \end{cases}$



Beberapa persoalan yang diselesaikan dengan D&C

- ▶ 1. Persoalan MinMaks (mencari nilai minimum dan nilai maksimum)
- ▶ 2. Menghitung perpangkatan
- ▶ 3. Persoalan pengurutan (sorting) – Mergesort dan Quicksort
- ▶ 4. Mencari sepasang titik terdekat (closest pair problem)
- ▶ 5. Convex Hull
- ▶ 6. Perkalian matriks
- ▶ 7. Perkalian bilangan bulat besar
- ▶ 8. Perkalian dua buah polinom

Persoalan MinMaks: Mencari Nilai Minimum dan Maksimum

Persoalan: Misalkan diberikan sebuah larik A yang berukuran n elemen dan sudah berisi nilai *integer*.

Carilah nilai minimum (min) dan nilai maksimum (max) sekaligus di dalam larik tersebut.

Contoh:

4	12	23	9	21	1	35	2	24
---	----	----	---	----	---	----	---	----

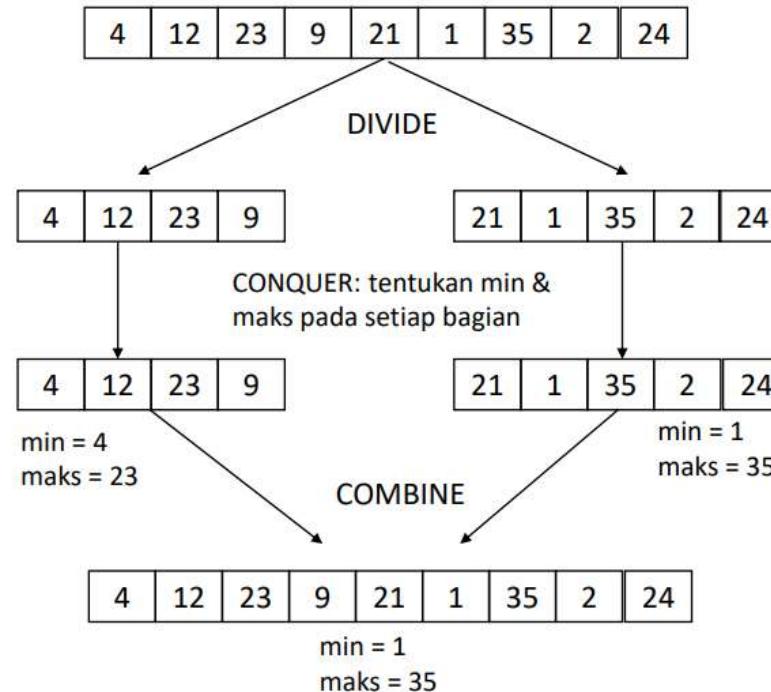
$$\text{min} = 1$$

$$\text{max} = 35$$

LANJUTAN

Penyelesaian dengan *algoritma divide and conquer*

Ide dasar secara *divide and conquer*:



LANJUTAN

- ▶ Ukuran larik hasil pembagian dapat dibuat cukup kecil sehingga mencari minimum dan maksimum dapat diselesaikan (SOLVE) secara trivial.
- ▶ Dalam hal ini, ukuran “kecil” yang didefinisikan apabila larik hanya berisi 1 elemen atau 2 elemen.

LANJUTAN

Penyelesaian dengan *algoritma brute force*

```
procedure MinMaks1(input A : TabellInteger, n : integer, output min, maks : integer)
{ Mencari nilai minimum dan maksimum di dalam larik a yang berukuran n elemen, secara brute force.
Masukan: larik a yang sudah terdefinisi elemen-elemennya
Luaran: nilai maksimum dan nilai minimum tabel
}
Deklarasi
i : integer

Algoritma:
min ← A[1] { asumsikan elemen pertama sebagai nilai minimum sementara}
maks ← A[1] {asumsikan elemen pertama sebagai nilai maksimum sementara}
for i ← 2 to n do
    if A[i] < min then
        min ← A[i]
    endif
    if A[i] > maks then
        maks ← A[i]
    endif
endfor
```

Jumlah perbandingan elemen larik: $T(n) = (n - 1) + (n - 1) = 2n - 2 = O(n)$

LANJUTAN

Prosedur MinMaks($A[1..n]$, min, maks)

Algoritma:

1. Untuk kasus $n = 1$ atau $n = 2$,

SOLVE: Jika $n = 1$, maka $\text{min} = \text{maks} = A[n]$

Jika $n = 2$, maka bandingkan kedua elemen untuk menentukan min dan maks

2. Untuk kasus $n > 2$,

(a) DIVIDE: Bagi dua larik A menjadi dua bagian yang sama, A_1 dan A_2

(b) CONQUER:

MinMaks(A_1 , $n/2$, min1, maks1)

MinMaks(A_2 , $n/2$, min2, maks2)

(c) COMBINE:

if $\text{min1} < \text{min2}$ then $\text{min} \leftarrow \text{min1}$ else $\text{min} \leftarrow \text{min2}$

if $\text{maks1} < \text{maks2}$ then $\text{maks} \leftarrow \text{maks2}$ else $\text{maks} \leftarrow \text{maks1}$

LANJUTAN

procedure MinMaks2(**input** A : LarikInteger, i, j : **integer**, **output** min, maks : **integer**)

{ Mencari nilai maksimum dan minimum di dalam larik A yang berukuran n elemen dengan algoritma divide and Conquer.

Masukan: larik A yang sudah terdefinisi elemen-elemennya

Luaran: nilai maksimum dan nilai minimum larik }

Deklarasi

min1, min2, maks1, maks2 : **integer**

Algoritma:

if $i = j$ **then** { larik berukuran 1 elemen }
 min $\leftarrow A[i]$; maks $\leftarrow A[i]$

else

if $(i = j - 1)$ **then** { larik berukuran 2 elemen }
 if $A[i] < A[j]$ **then**
 min $\leftarrow A[i]$; maks $\leftarrow A[j]$

else

 min $\leftarrow A[j]$; maks $\leftarrow A[i]$

endif

else { larik berukuran lebih dari 2 elemen }
 k $\leftarrow (i + j) \text{ div } 2$ { bagidua larik pada posisi k }

 MinMaks2(A, i, k, min1, maks1)

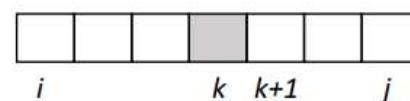
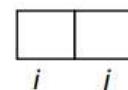
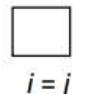
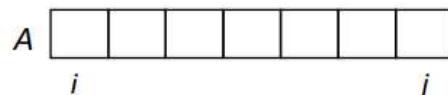
 MinMaks2(A, k + 1, j, min2, maks2)

if min1 < min2 **then** min $\leftarrow min1$ **else** min $\leftarrow min2$ **endif**

if maks1 < maks2 **then** maks $\leftarrow maks2$ **else** maks $\leftarrow maks1$ **endif**

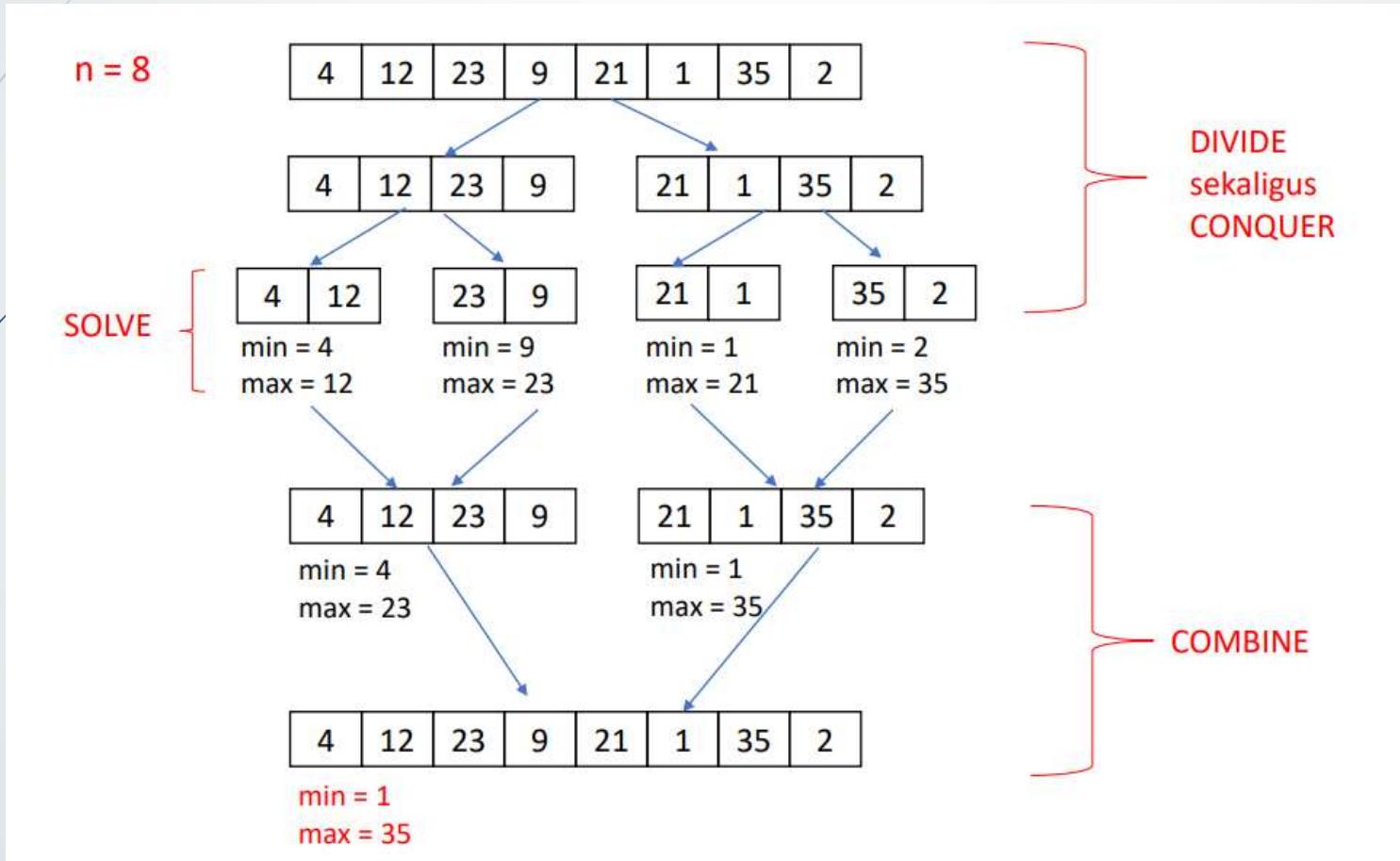
endif

endif

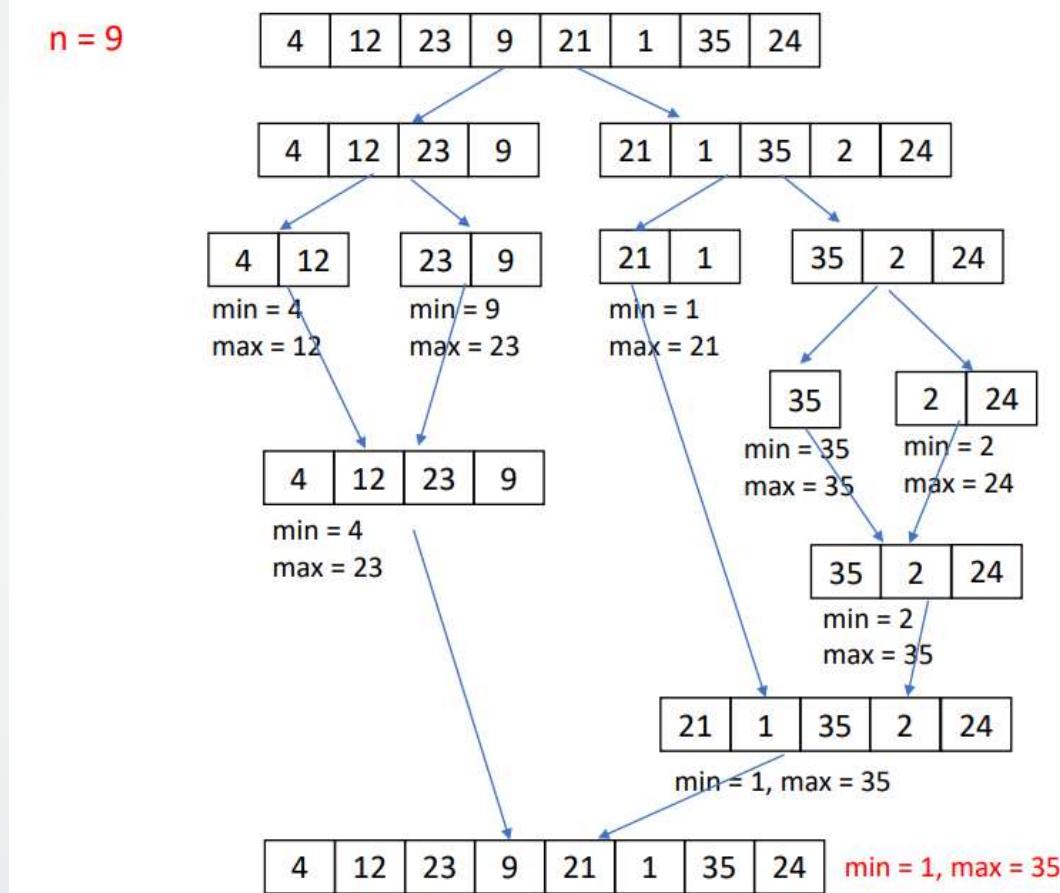


Pemanggilan pertama kali: MinMaks2(A, 1, n, min, maks)

Contoh 1: Mencari nilai minimum dan maksimum di dalam larik berikut



Contoh 2: Mencari nilai minimum dan maksimum di dalam larik berikut



LANJUTAN

Kompleksitas waktu algoritma *MinMaks2*, dihitung dari jumlah operasi perbandingan elemen-elemen larik:

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

Penyelesaian:

Asumsi: $n = 2^k$, dengan k bilangan bulat positif, maka

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 = 4T(n/4) + 4 + 2 \\ &= 4T(2T(n/8) + 2) + 4 + 2 = 8T(n/8) + 8 + 4 + 2 \\ &= \dots \\ &= 2^{k-1} T(2) + \sum_{i=1}^{k-1} 2^i \\ &= 2^{k-1} \cdot 1 + 2^k - 2 \\ &= n/2 + n - 2 \\ &= 3n/2 - 2 \\ &= O(n) \end{aligned}$$

Perpangkatan an

- Misalkan $a \in R$ dan n adalah bilangan bulat tidak negatif, maka perpangkatan a^n didefinisikan sebagai berikut:

$$a^n = \begin{cases} 1, & n = 0 \\ a \times a \times \cdots \times a, & n > 0 \end{cases}$$

Bagaimana algoritma menghitung perpangkatan a^n secara *brute force* dan secara *divide and conquer*?

LANJUTAN

Penyelesaian dengan *algoritma brute force*

```
function Exp1(a : real, n : integer)→ real  
{ Menghitung  $a^n$ ,  $a > 0$  dan  $n$  bilangan bulat tak-negatif }
```

Deklarasi

```
k : integer  
hasil : real
```

Algoritma:

```
hasil ← 1  
for k ← 1 to n do  
    hasil ← hasil * a  
endfor
```

```
return hasil
```

Kompleksitas algoritma, dihitung dari jumlah operasi perkalian: $T(n) = n = O(n)$

LANJUTAN

Penyelesaian dengan algoritma *Divide and Conquer*

Ide dasar: bagi dua pangkat n menjadi $n = n/2 + n/2$

$$a^n = a^{(n/2 + n/2)} = a^{n/2} \cdot a^{n/2}$$

Algoritma *divide and conquer* untuk menghitung a^n :

1. Untuk kasus $n = 0$, maka $a^n = 1$.
2. Untuk kasus $n > 0$, bedakan menjadi dua kasus lagi:
 - (i) jika n genap, maka $a^n = a^{n/2} \cdot a^{n/2}$
 - (ii) jika n ganjil, maka $a^n = a^{n/2} \cdot a^{n/2} \cdot a$

Contoh 3. Menghitung 3^{16} dengan metode Divide and Conquer:

$$\begin{aligned}3^{16} &= 3^8 \cdot 3^8 = (3^8)^2 \\&= ((3^4)^2)^2 \\&= (((3^2)^2)^2)^2 \\&= ((((3^1)^2)^2)^2)^2 \\&= (((((3^0)^2 \cdot 3)^2)^2)^2)^2 \\&= (((((1)^2 \cdot 3)^2)^2)^2)^2 \quad \rightarrow \text{Hanya membutuhkan enam operasi perkalian} \\&\qquad\qquad\qquad (\text{operasi perpangkatan dua} = \text{perkalian}) \\&= (((3)^2)^2)^2 \\&= ((9)^2)^2 \\&= (81)^2 \\&= (6561)^2 \\&= 43046721\end{aligned}$$

LANJUTAN

Pseudo-code menghitung a^n dengan divide and conquer:

```
function Exp2(a : real, n : integer) → real
{ mengembalikan nilai  $a^n$ , dihitung dengan metode Divide and Conquer }

Algoritma:
if n = 0 then
    return 1
else
    if odd(n) then { kasus n ganjil }
        return Exp2(a, n div 2) * Exp2(a, n div 2) * a      { $a^n = a^{n/2} \cdot a^{n/2} \cdot a$ }
    else           { kasus n genap }
        return Exp2(a, n div 2) * Exp2(a, n div 2)      { $a^n = a^{n/2} \cdot a^{n/2}$ }
    endif
endif
```

Fungsi *Exp2* tidak mangkus, sebab terdapat dua kali pemanggilan rekursif untuk nilai parameter yang sama → *Exp2(a, n div 2) * Exp2(a, n div 2)*

LANJUTAN

Perbaikan: simpan hasil $Exp2(a, n \text{ div } 2)$ di dalam sebuah peubah (misalkan x), lalu gunakan x untuk menghitung a^n pada kasus n genap dan n ganjil.

```
function Exp3(a : real, n : integer) → real
{ mengembalikan nilai  $a^n$ , dihitung dengan metode Divide and Conquer }

Deklarasi
x : real
Algoritma:
if n = 0 then
    return 1
else
    x ← Exp3(a, n div 2)
    if odd(n) then { kasus n ganjil }
        return x * x * a
    else { kasus n genap }
        return x * x
    endif
endif
```

LANJUTAN

- Kompleksitas algoritma *Exp3* dihitung dari jumlah operasi perkalian:

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

$x * x * a$
 $x * x$

- Dalam menghitung $T(n)$ ini ada sedikit kesulitan, yaitu nilai n mungkin ganjil atau genap, sehingga penyelesain relasi rekurens menjadi lebih rumit.
- Namun, perbedaan ini dianggap kecil sehingga dapat kita abaikan. Sebagai implikasinya, kita membuat asumsi penghampiran bahwa untuk n genap atau ganjil, jumlah operasi perkalian relatif sama.

LANJUTAN

- Sehingga, kompleksitas algoritma *Exp3* menjadi:

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

- Asumsikan n adalah perpangkatan dari 2, atau $n = 2^k$, maka

$$\begin{aligned} T(n) &= 1 + T(n/2) \\ &= 1 + (1 + T(n/4)) = 2 + T(n/4) \\ &= 2 + (1 + T(n/8)) = 3 + T(n/8) \\ &= \dots \\ &= k + T(n/2^k) \end{aligned}$$

Karena $n = 2^k$ maka $k = \log_2 n$, sehingga

$$\begin{aligned} &= k + T(n/2^k) = \log_2 n + T(1) \\ &= \log_2 n + (1 + T(0)) = \log_2 n + 1 + 0 \\ &= \log_2 n + 1 = O(\log n) \rightarrow \text{lebih baik daripada algoritma } \textit{brute force}! \end{aligned}$$

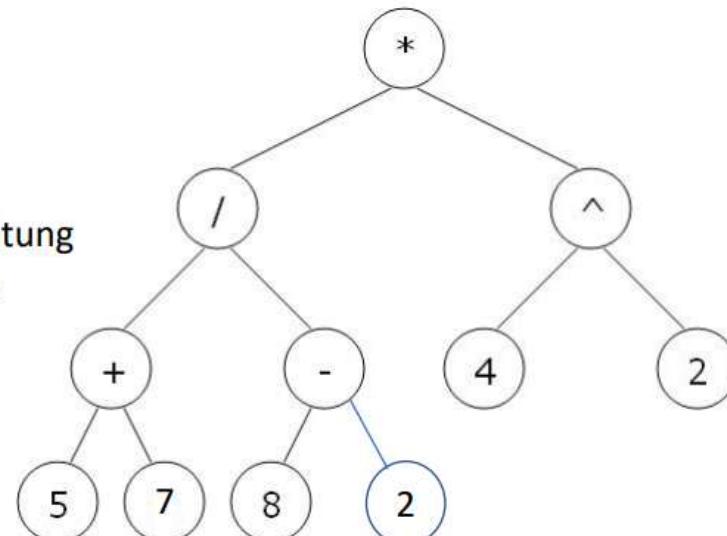
Mengevaluasi Pohon Ekspresi

- Di dalam *compiler* bahasa pemrograman, ekspresi aritmetika direpresentasikan dalam pohon biner yaitu pohon ekspresi (*expression tree*)

Contoh: $(5 + 7) / (8 - 2) * (4 ^ 2)$

- Mengevaluasi pohon ekspresi artinya menghitung nilai ekspresi aritmetika yang dinyatakannya.

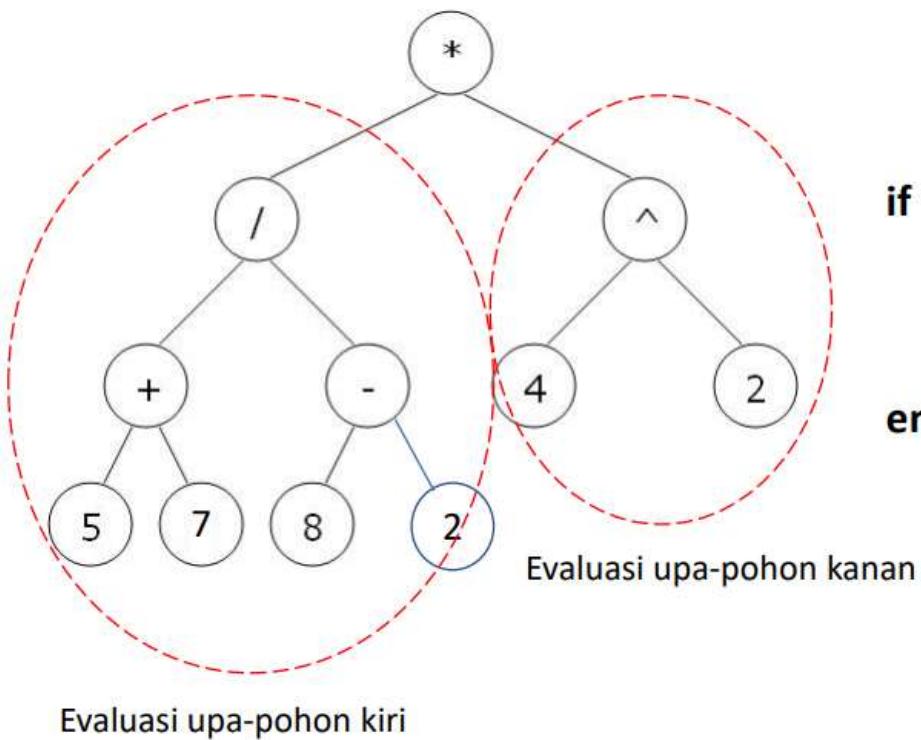
Contoh: $(5 + 7) / (8 - 2) * (4 ^ 2) = 32$



Pohon ekspresi

LANJUTAN

- Algoritma *divide and conquer*:



```
if pohon tidak kosong then
    nilai1 ← Evaluasi(upa-pohon kiri)
    nilai2 ← Evaluasi(upa-pohon kanan)
    Gabungkan nilai1 dan nilai2 dengan operatornya
end
```

LANJUTAN

- Misalkan pohon ekspresi direpresentasikan dengan senarai berkait (*linked list*).

Simpul daun → *operand*, contoh: 4, -2, 0, dst

Simpul dalam → operator, contoh: +, -, *, /, ^

Struktur setiap simpul:

left	info	right
------	------	-------

info: *operand* atau operator

Pada simpul daun → left = NIL dan right = NIL

- Algoritma *divide and conquer*:

if simpul adalah daun **then**

return info

else

secara rekursif evaluasi upa-pohon kiri dan return nilainya

secara rekursif evaluasi upa-pohon kanan dan return nilainya

gabungkan kedua nilai tersebut sesuai dengan operator dan return nilainya

endif

LANJUTAN

- Algoritma evaluasi pohon ekspresi dalam bentuk prosedur:

```
procedure EvaluasiPohon(input T : Pohon, output nilai : real)
{ Mengevaluasi pohon ekspresi T
  Masukan: Pohon ekspresi T, asumsik T tidak kosong
  Luaran: nilai berisi hasil evaluasi ekspresi
}

Deklarasi
  nilai1, nilai2 : real
Algoritma:
  if left(T) = NIL and right(T) = NIL { simpul daun}
    nilai ← info(T)
  else { simpul dalam }
    EvaluasiPohon(left(T), nilai1);
    EvaluasiPohon(right(T), nilai2);
    case info(T) of
      "+" : nilai ← nilai1 + nilai2
      "-" : nilai ← nilai1 - nilai2
      "*" : nilai ← nilai1 * nilai2
      "/" : nilai ← nilai1 / nilai2      {dengan syarat nilai2 ≠ 0 }
      "^" : nilai ← nilai1 ^ nilai2      {dengan syarat nilai1 ≠ 0 dan nilai2 ≠ 0 }
    end
  end
```

LANJUTAN

- Algoritma evaluasi pohon ekspresi dalam bentuk fungsi:

```
function EvaluasiPohon(T : Pohon) → real
{ mengevaluasi pohon ekspresi T }

Deklarasi
    nilai1, nilai2 : real

Algoritma:
    if left(T) = NIL and right(T) = NIL { simpul daun }
        return info(T)
    else { simpul dalam }
        case info(T) of
            "+" : return EvaluasiPohon(left(T), nilai1) + EvaluasiPohon(right(T), nilai2);
            "-" : return EvaluasiPohon(left(T), nilai1) - EvaluasiPohon(right(T), nilai2);
            "*" : return EvaluasiPohon(left(T), nilai1) * EvaluasiPohon(right(T), nilai2);
            "/" : return EvaluasiPohon(left(T), nilai1) / EvaluasiPohon(right(T), nilai2); {nilai2 ≠ 0}
            "^" : return EvaluasiPohon(left(T), nilai1) ^ EvaluasiPohon(right(T), nilai2); {nilai1 ≠ 0 dan nilai2 ≠ 0}
        end
    end
```

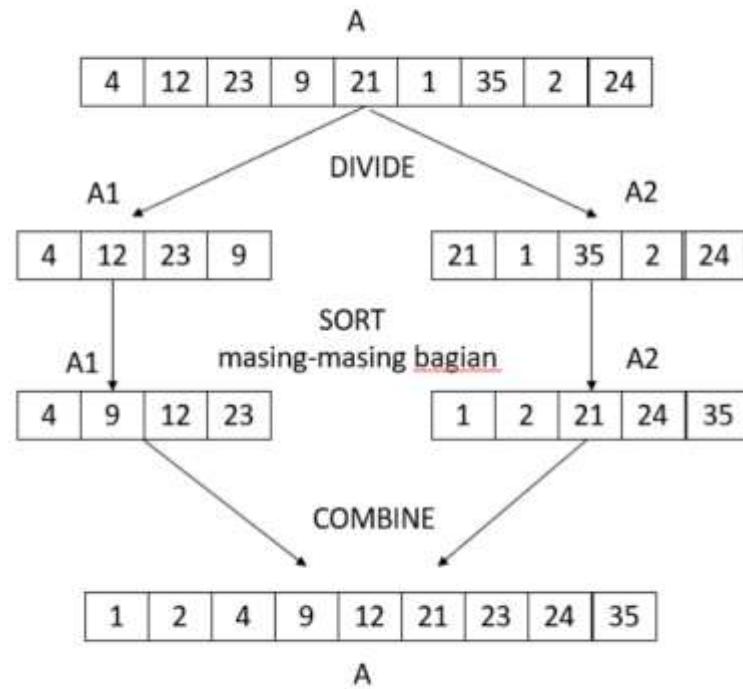
Pengurutan Secara Divide and Conquer

- 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 ?

LANJUTAN

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.



LANJUTAN

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

LANJUTAN

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

LANJUTAN

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: A1 [8 1 4 6] A2 [9 3 5 7]

Sort: A1 [1 4 6 8] A2 [3 5 7 9]

Combine: A1 [1 3 4 5 6 7 8 9]

Algoritma pengurutan yang termasuk jenis ini:

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

LANJUTAN

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

Tabel A dibagidua berdasarkan nilai elemennya. Misalkan elemen-elemen $A_1 \leq$ 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]

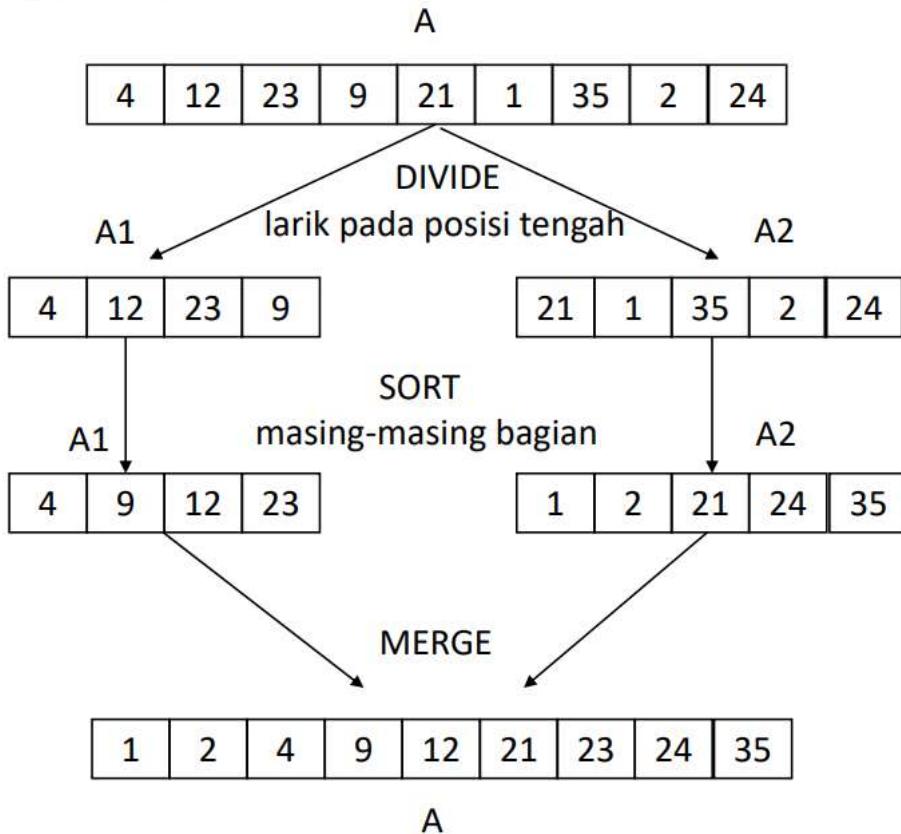
Combine: A [1 | 3 | 4 | 5 | 6 | 7 | 8 | 9]

Algoritma pengurutan yang termasuk jenis ini:

- urut-cepat (*Quick Sort*)
- urut-seleksi (*Selection Sort*)

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*

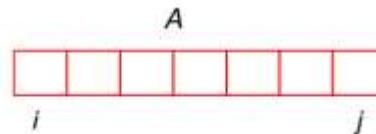
LANJUTAN

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.

LANJUTAN

- 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
```

A diagram showing the same array structure as above, but with the element at index 'k' shaded gray. Below the array, the indices 'i', 'k', 'k+1', and 'j' are labeled under their respective boxes.

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

LANJUTAN

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

<i>A1</i>
1 13 24

<i>A2</i>
2 15 27

$$1 < 2 \rightarrow 1$$

<i>B</i>
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

LANJUTAN

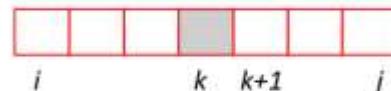
```
procedure Merge(input/output A : LarikInteger, input i, k, j : integer)
{ Menggabung larik A[i..k] dan larik A[k+1..j] menjadi larik A[i..j] yang terurut menaik. A
  Masukan: A[i..k] dan A[k+1..j] sudah terurut menaik.
  Luaran: A[i..j] yang terurut menaik. }
```

Deklarasi

```
B : LarikInteger      { larik temporer 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
```



LANJUTAN

{ 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**

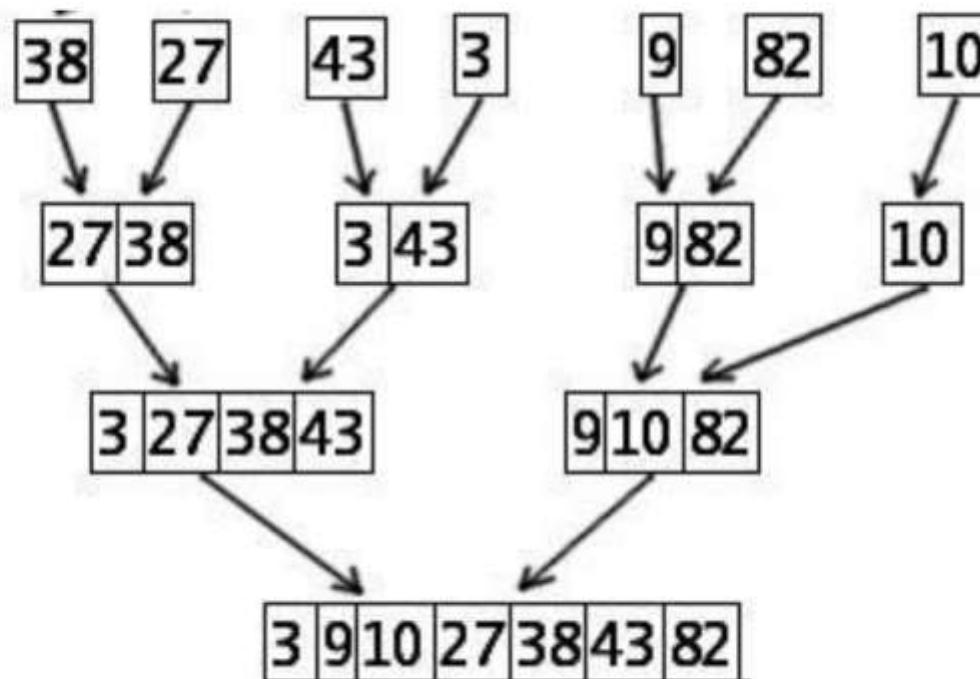
$A[r] \leftarrow B[r]$

endfor

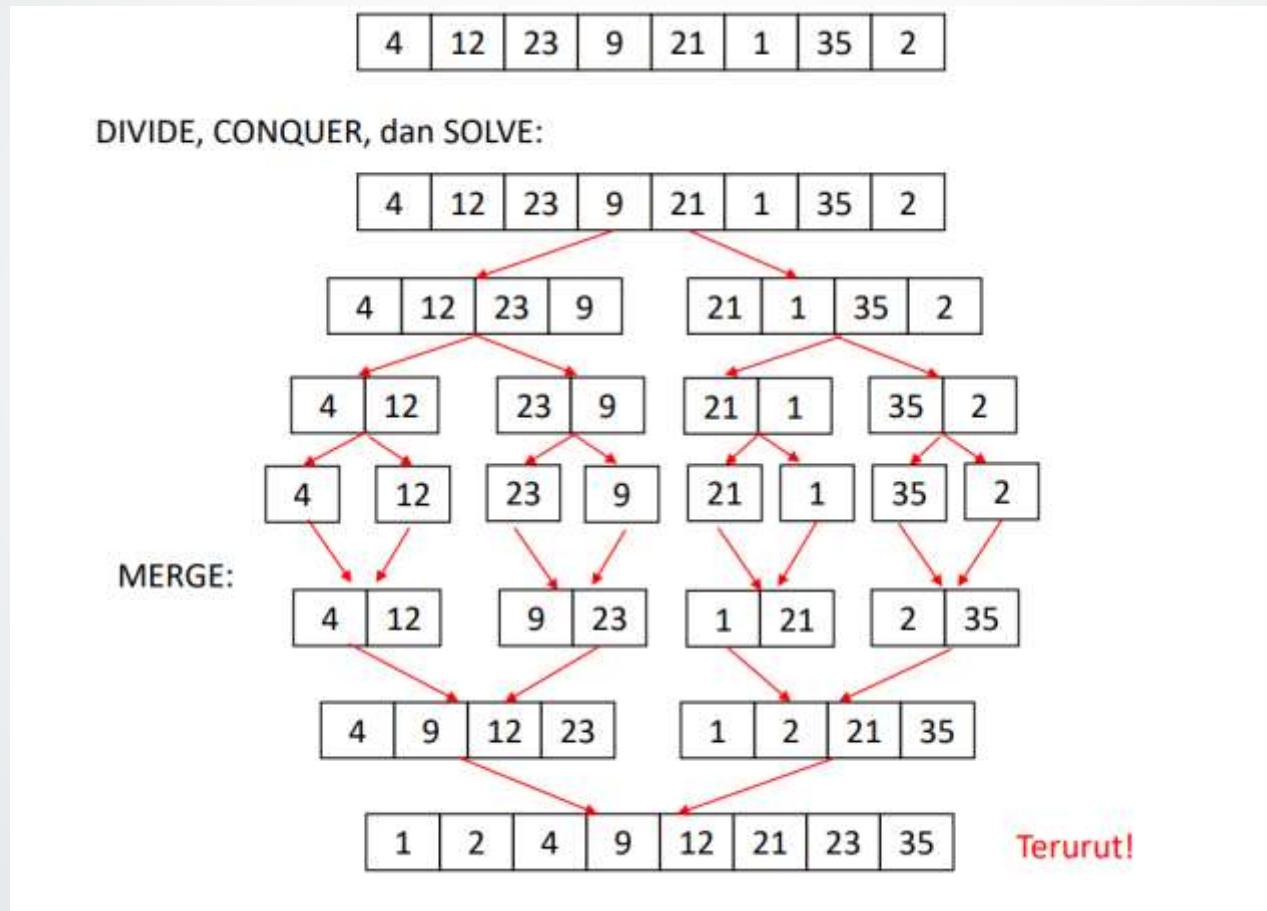
{ diperoleh larik A yang terurut membesar }

LANJUTAN

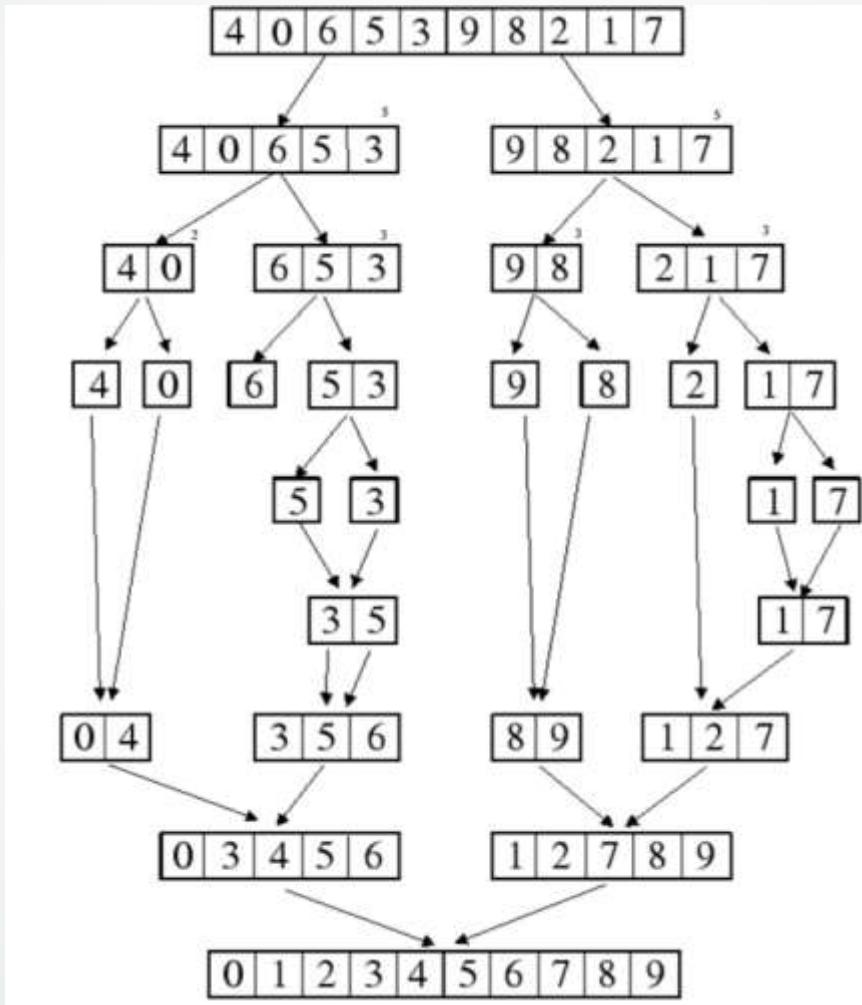
Contoh proses *merge* di dalam *Merge Sort*:



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



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

LANJUTAN

- Penyelesaian persamaan rekursif secara iteratif :

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

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

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

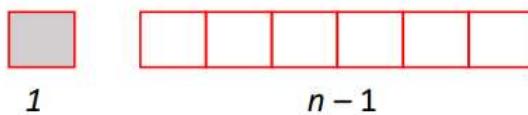
sehingga

$$T(n) = nT(1) + cn \log n = an + cn \log n = O(n \log n)$$

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

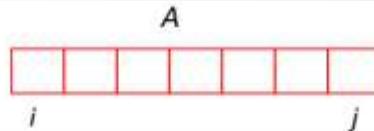
Insertion Sort

- *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.

LANJUTAN



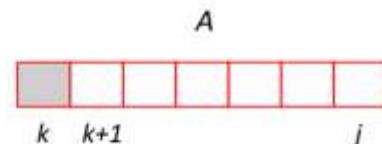
```
procedure InsertionSort(input/output A : LarikInteger, input i,j : integer)
{ Mengurutkan larik A[i..j] dengan algoritma Insertion Sort.
  Masukan: Larik A[i..j] yang sudah terdefinisi elemen-elemennya
  Luaran: Larik A[i..j] yang terurut
}
```

Deklarasi:

k : integer

Algoritma:

```
if i < j then
  k ← i
  InsertSort(A, i, k)           { ukuran(A) > 1 }
  InsertSort(A, k + 1, j)        { bagi A pada posisi elemen pertama }
  InsertSort(A, k + 1, j)        { urut upalarik A[i..k] }
  Merge(A, i, k, 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)*

LANJUTAN

- **Perbaikan:** karena upalarik pertama hanya berisi satu elemen, maka kita tidak perlu melakukan 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
---	----	---	---	---	----	---	---

LANJUTAN

MERGE:

<u>4</u>	<u>12</u>	<u>3</u>	<u>9</u>	<u>1</u>	<u>21</u>	<u>5</u>	<u>2</u>
<u>4</u>	<u>12</u>	<u>3</u>	<u>9</u>	<u>1</u>	<u>21</u>	<u>2</u>	<u>5</u>
<u>4</u>	<u>12</u>	<u>3</u>	<u>9</u>	<u>1</u>	<u>2</u>	<u>5</u>	<u>21</u>
<u>4</u>	<u>12</u>	<u>3</u>	<u>9</u>	<u>1</u>	<u>2</u>	<u>5</u>	<u>21</u>
<u>4</u>	<u>12</u>	<u>3</u>	<u>1</u>	<u>2</u>	<u>5</u>	<u>9</u>	<u>21</u>
<u>4</u>	<u>12</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>5</u>	<u>9</u>	<u>21</u>
<u>4</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>5</u>	<u>9</u>	<u>12</u>	<u>21</u>
<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>9</u>	<u>12</u>	<u>21</u>

LANJUTAN

Kompleksitas waktu algoritma *Insertion Sort*:

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

Penyelesaian:

$$\begin{aligned} T(n) &= cn + T(n-1) \\ &= cn + \{ c(n-1) + T(n-2) \} \\ &= cn + c(n-1) + \{ c(n-2) + T(n-3) \} \\ &= cn + c(n-1) + c(n-2) + \{ c(n-3) + T(n-4) \} \\ &= \dots \\ &= cn + c(n-1) + c(n-2) + c(n-3) + \dots + c2 + T(1) \\ &= c\{ n + (n-1) + (n-2) + (n-3) + \dots + 2 \} + a \\ &= c\{ (n-1)(n+2)/2 \} + a \\ &= cn^2/2 + cn/2 + (a - c) \\ &= O(n^2) \rightarrow \text{sama seperti kompleksitas algoritma } \textit{Insertion Sort} \\ &\quad \text{versi iteratif} \end{aligned}$$

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)

LANJUTAN

- 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
-----	---	---	---	---	---	---	---	---

LANJUTAN

- 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 7. Misalkan larik A berisi elemen-elemen berikut:

8 1 4 6 9 3 5 7

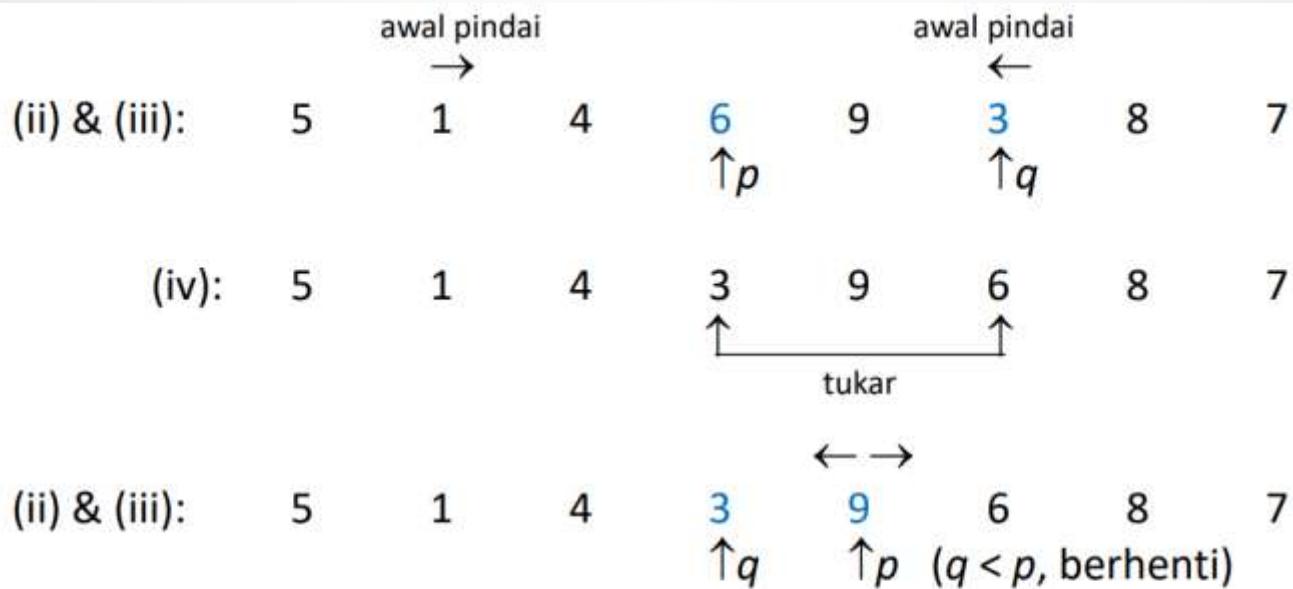
Misalkan $pivot = 6$ (elemen tengah larik). Langkah-langkah partisi adalah sbb:

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

(iv):

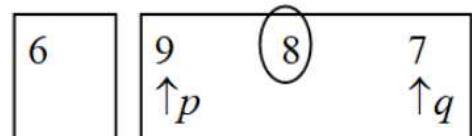
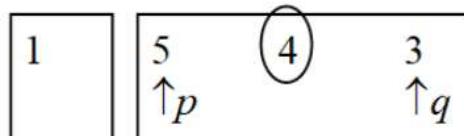
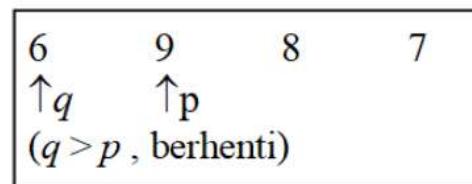
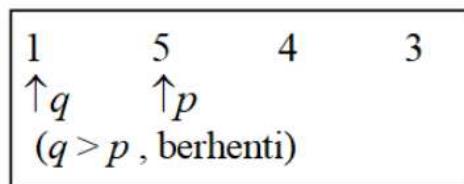
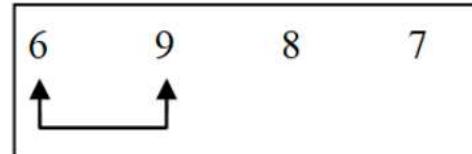
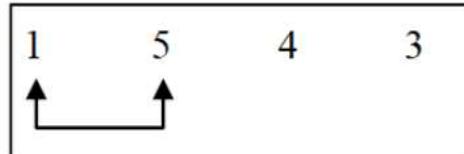
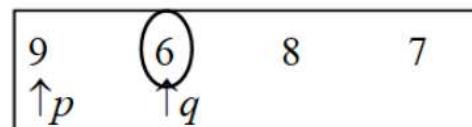
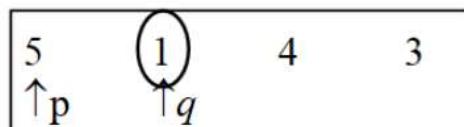
tukar

LANJUTAN

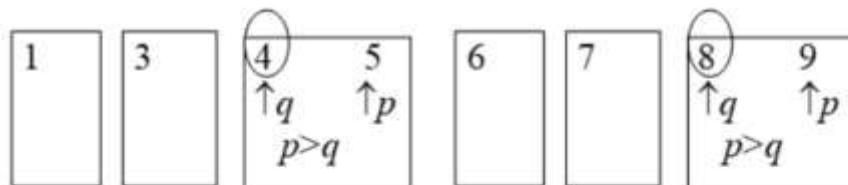
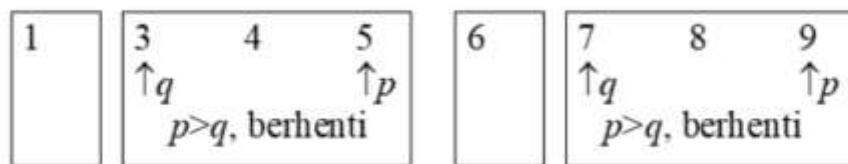
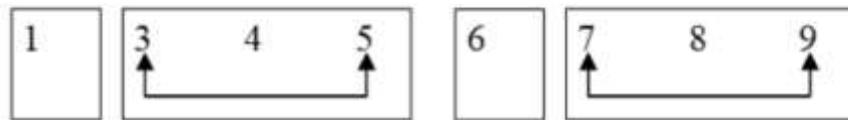
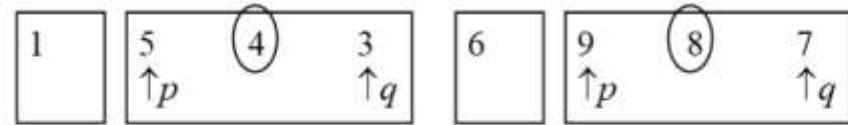


LANJUTAN

Teruskan partisi untuk setiap bagian sampai berukuran satu elemen:



LANJUTAN



1 3 4 5 6 7 8 9 (tenurut)

LANJUTAN

- *Pseudo-code algoritma Quicksort:*

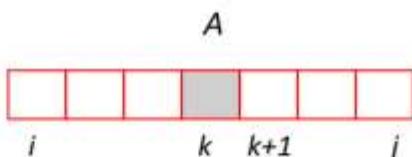
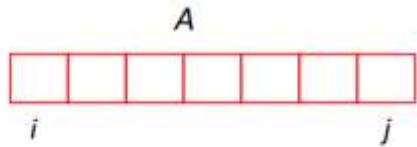
```
procedure QuickSort(input/output A : LarikInteger, input i,j : integer)
{ Mengurutkan larik A[i..j] dengan algoritma Quicksort.
  Masukan: Larik A[i..j] yang sudah terdefinisi elemen-elemennya
  Luaran: Larik A[i..j] yang terurut
}
```

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
```



LANJUTAN

```
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:

$pivot \leftarrow$ 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 $p \leq q$ then

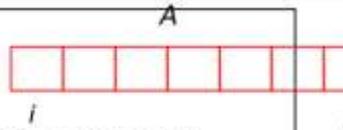
 swap($A[p], A[q]$) {pertukarkan $A[p]$ dengan $A[q]$ }

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

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

 endif

until $p > q$



LANJUTAN

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

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

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 8 (Levitin, 2003):



LANJUTAN

	→		←
2	3	1	4
	↑p	↑q	
2	1	3	4
	↓	↓	
2	1	3	4
	↑q	↑p	
1	2	3	4
1	3	4	
	↑q	↑p	
			4

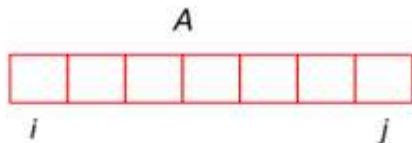
	→		←
8	9	7	
	↑p	↑q	
8	7	9	
	↓	↓	
8	7	9	
	↑q	↑p	
7	8	9	
7			9

Terurut:

1 2 3 4 5 7 8 9

LANJUTAN

- *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 Sort2 }
endif
```

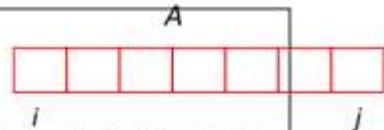
LANJUTAN

```
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 pemindai dari kiri }
  q ← j + 1        { awal pemindai 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] }
```



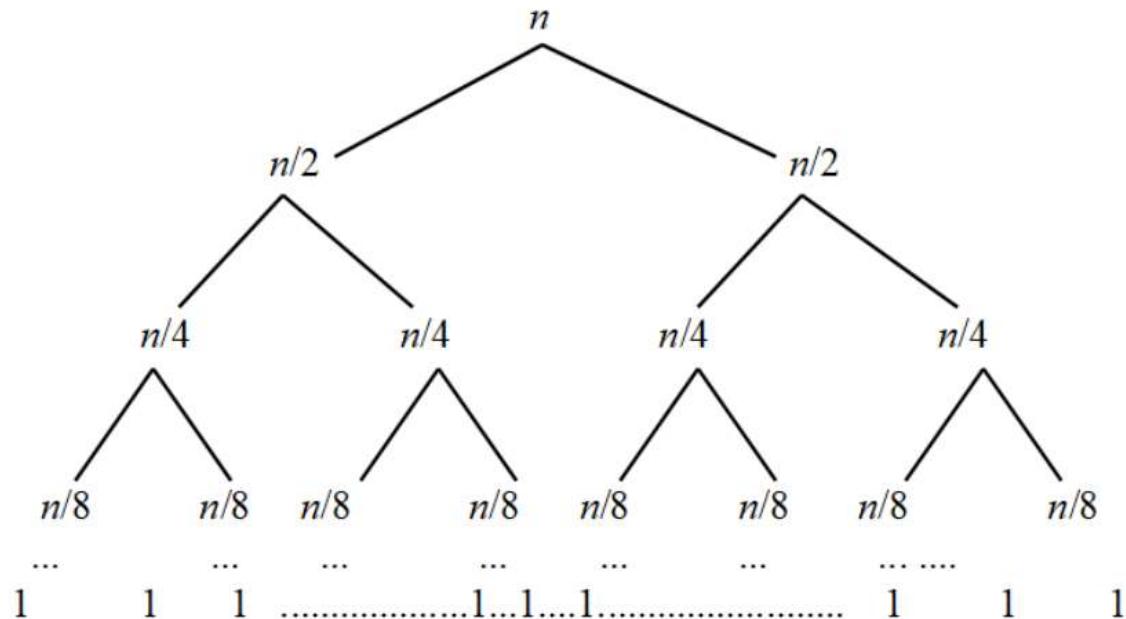
LANJUTAN

- ▶ Cara pemilihan pivot (khusus pada Quicksort versi 1):
 - ▶ 1. Pivot = elemen pertama/element terakhir/element 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.



LANJUTAN

- 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*:

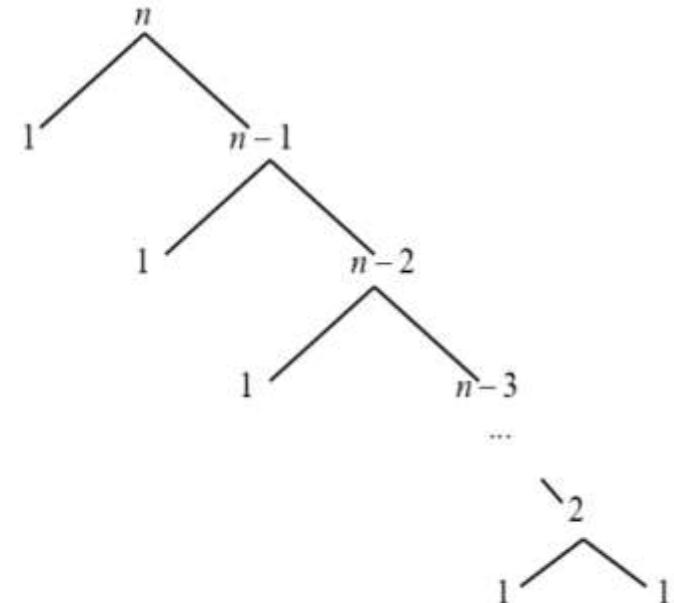
$$T(n) = 2T(n/2) + cn = na + cn^2 \log n = O(n^2 \log n).$$

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

LANJUTAN

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.



LANJUTAN

- 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*.

LANJUTAN

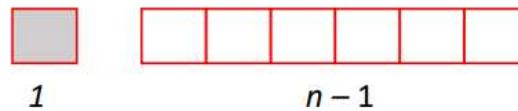
3. Kasus rata-rata (*average case*)

- 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:

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

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.



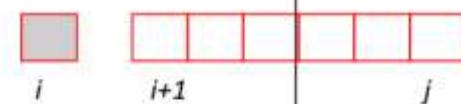
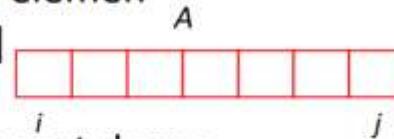
- *Selection Sort* dapat dipandang sebagai kasus khusus dari *Quick Sort* dengan hasil pembagian terdiri dari 1 elemen dan $n - 1$ elemen, namun proses partisinya dilakukan dengan cara berbeda.

LANJUTAN

- 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
  Partisi3(A, i, j) { Ukuran(A) > 1 }
  { 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*

LANJUTAN

```
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:

$idxmin \leftarrow i$

for $k \leftarrow i+1$ **to** j **do**

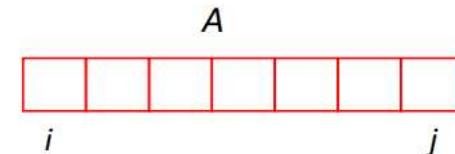
if $A[k] < A[idxmin]$ **then**

$idxmin \leftarrow k$

endif

endfor

$swap(A[i], A[idxmin])$ { pertukarkan $A[i]$ dengan $A[idxmin]$ }



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

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!

LANJUTAN

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 menghasilkan 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*, $O(n^2)$)

Teorema Master

- Teorema Master dapat digunakan untuk menentukan notasi asimptotik kompleksitas waktu yang berbentuk relasi rekurens dengan mudah tanpa harus menyelesaiakannya secara iteratif.
- Misalkan $T(n)$ adalah fungsi monoton naik 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 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 $a = b^d$)

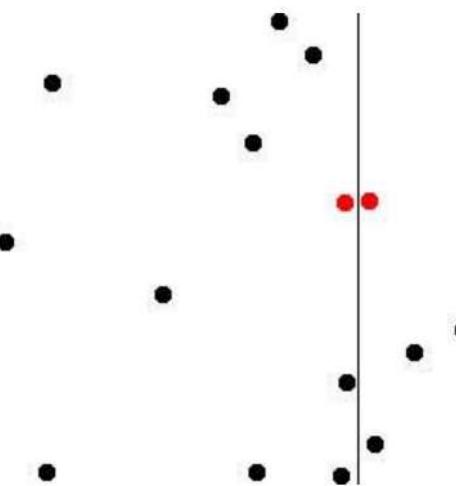
$$\begin{cases} O(n^d) & \text{jika } a < b^d \\ O(n^d \log n) & \text{jika } a = b^d \\ O(n^{\log_b a}) & \text{jika } a > b^d \end{cases}$$

sehingga

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

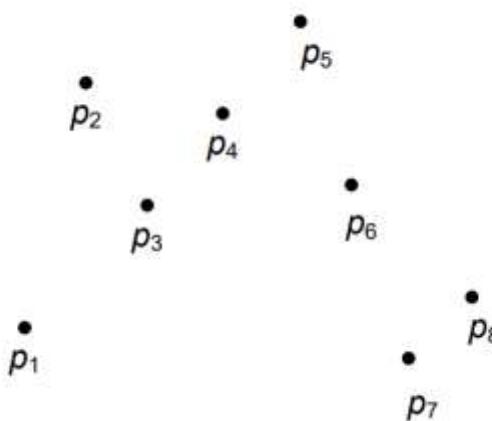
Mencari Pasangan Titik Terdekat (Closest Pair)

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.



LANJUTAN

$n = 8$



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

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

LANJUTAN

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 $O(n^2)$.

LANJUTAN

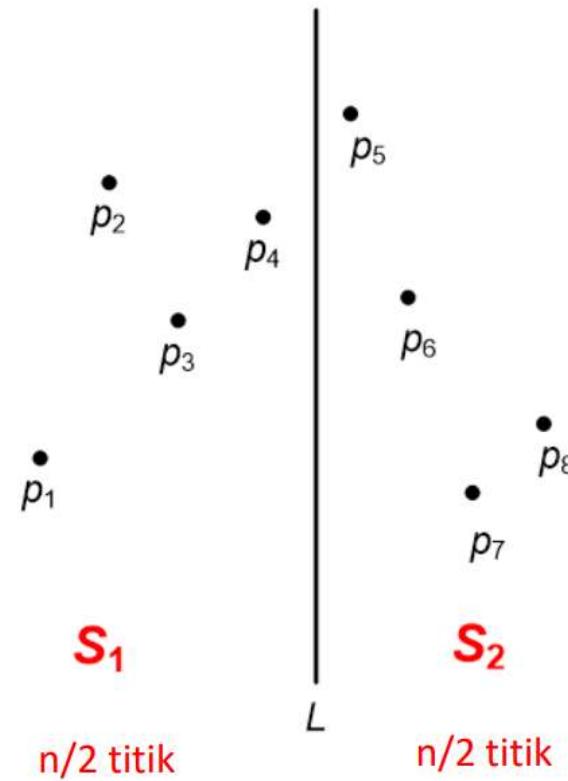
Penyelesaian secara *Divide and Conquer*

- 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.

LANJUTAN

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)).

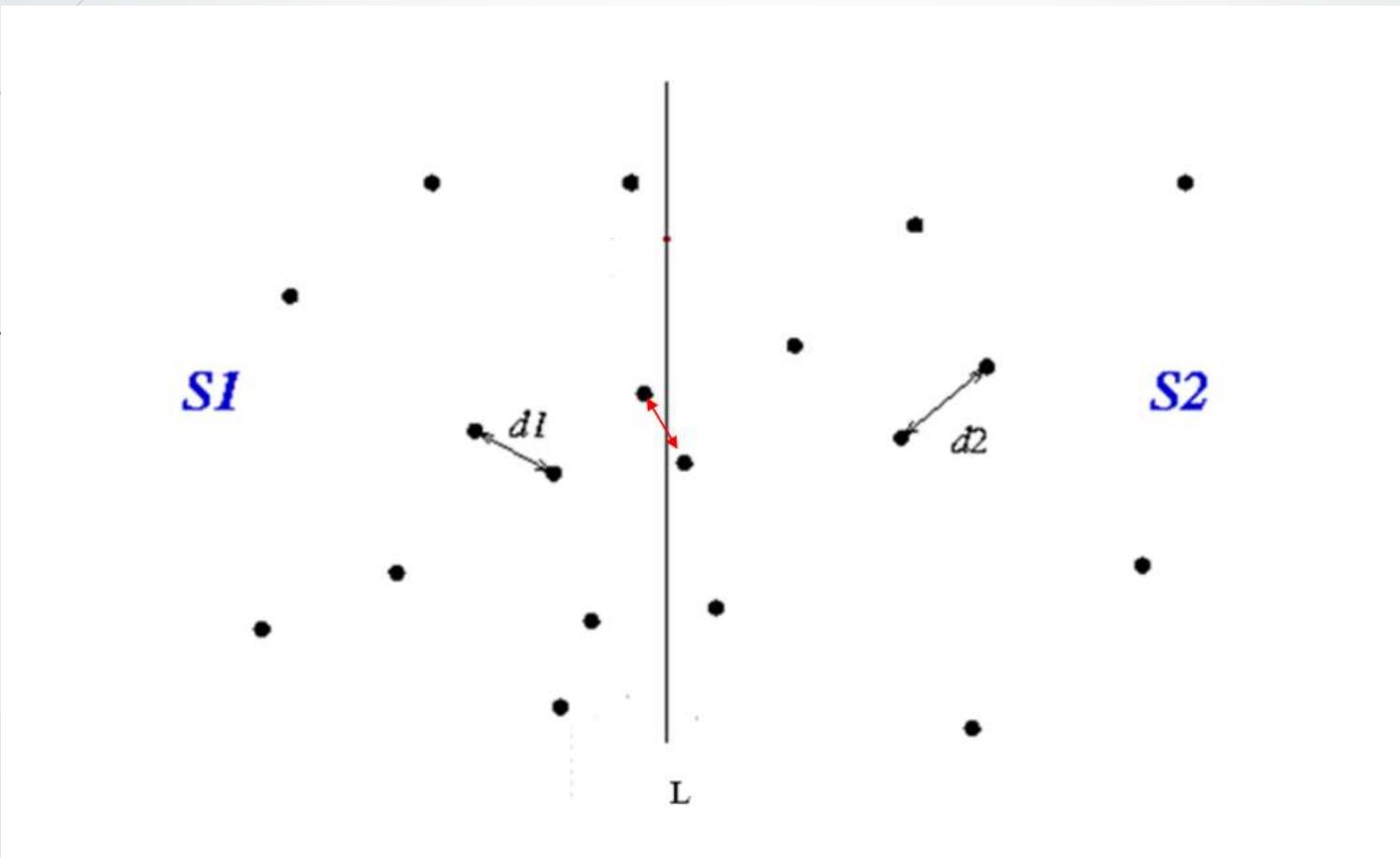


LANJUTAN

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.

LANJUTAN



LANJUTAN

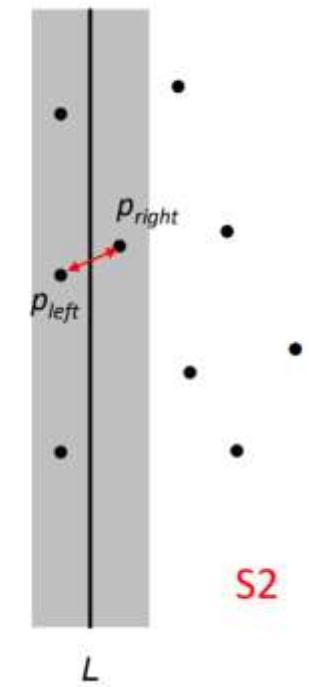
```
procedure FindClosestPair(input P : SetOfPoint, n : integer, output d : real)
{ Mencari jarak terdekat sepasang titik di dalam himpunan P
  Masukan: P = { $p_1, p_2, \dots, p_n$ }, titik-titik di dalam P sudah terurut menaik berdasarkan absisnya (x)
  Luaran: d adalah jarak sepasang titik terdekat }

Deklarasi:
d1, d2 : 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  $\leftarrow$  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 jarak( $p_{left}, p_{right}$ ) < d. Jika ada,
  maka set d dengan jarak terkecil tersebut.
  {*****}
endif
```

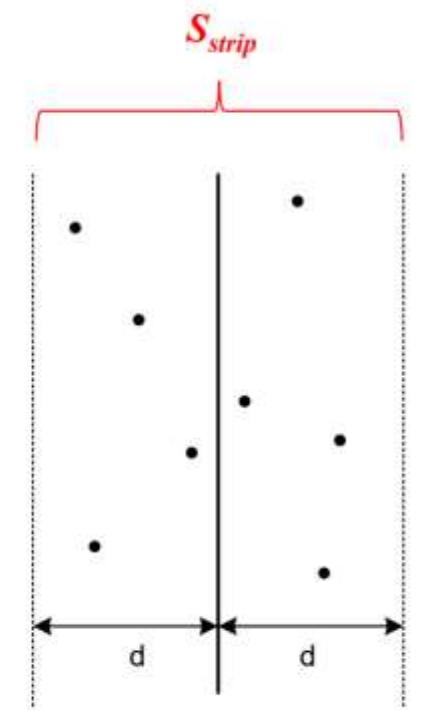
LANJUTAN

- 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?

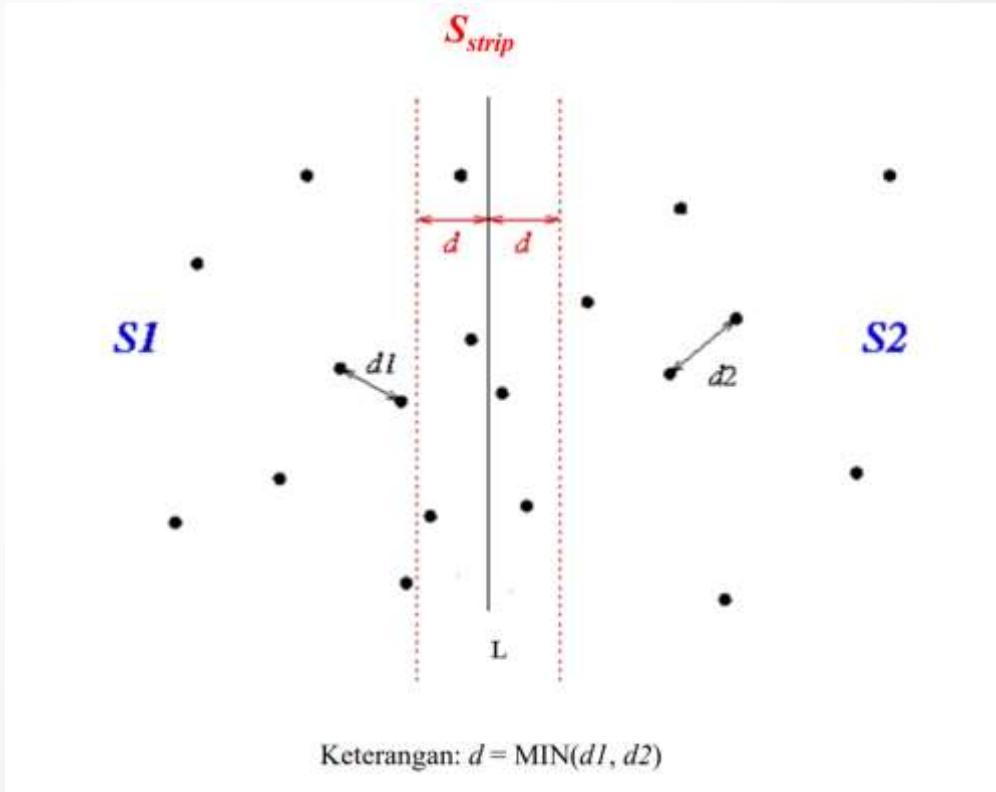


LANJUTAN

- 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.



LANJUTAN



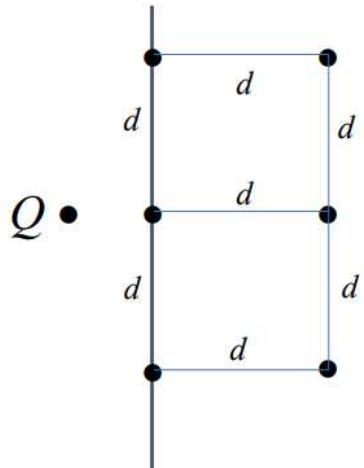
LANJUTAN

- 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←1 to s do
    for j←i+1 to s do
        if (ABS( $q_i.x - q_j.x$ ) > d or ABS( $q_i.y - q_j.y$ ) > d) then
            { tidak diproses }
        else
            d3 ← 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 ← d3
            endif
        endif
    endfor
endfor
```

LANJUTAN

- 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 .



LANJUTAN

Kompleksitas Algoritma *Closest Pair*

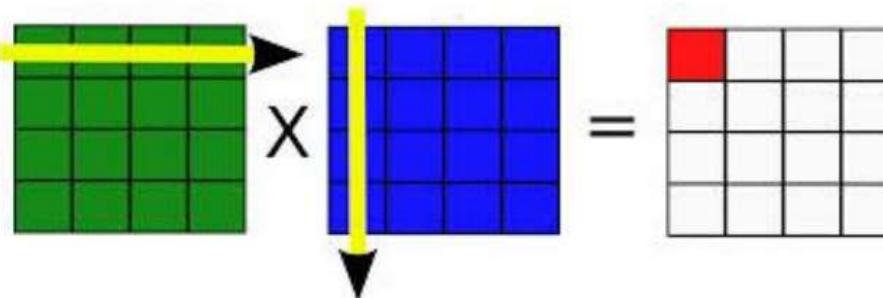
- Pengurutan titik-titik dalam absis x dan ordinat y dilakukan sebelum menerapkan algoritma *Divide and Conquer*.
- Pemrosesan titik-titik di dalam S_{strip} memerlukan waktu $t(n) = cn = O(n)$.
- Kompleksitas algoritma *closest pair*:

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

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

Perkalian Matriks

- Misalkan A dan B dua buah matrik berukuran $n \times n$.
- Perkalian matriks: $C = A \times B$



Elemen-elemen hasilnya: $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$

LANJUTAN

Penyelesaian secara *Brute Force*

Algoritma *brute force*: kalikan setiap vektor baris i dari matriks A dengan setiap vektor kolom j dari matriks B .

$$\text{Elemen-elemen hasilnya: } c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

```
function KaliMatriks(A, B : Matriks, n : integer) → Matriks
{ Mengalikan matriks A dan B yang berukuran  $n \times n$ , menghasilkan matriks C yang juga berukuran  $n \times n$  }

Deklarasi
i, j, k : integer

Algoritma:
for i←1 to n do
    for j←1 to n do
        C[i, j]←0 {inisialisasi penjumlahan}
        for k ← 1 to n do
            C[i, j]←C[i, j] + A[i, k]*B[k, j]
        endfor
    endfor
endfor
return C
```

Kompleksitas waktu algoritma: $O(n^3)$

LANJUTAN

Penyelesaian dengan *Divide and Conquer*

Matriks A dan B dibagi menjadi 4 buah matriks bujur sangkar. Masing-masing matriks bujur sangkar berukuran $n/2 \times n/2$:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

A B C

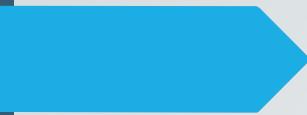
Elemen-elemen matriks C adalah:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

$$A = \begin{bmatrix} A_{11} & & A_{12} \\ & A_{21} & \\ & & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & & B_{12} \\ & B_{21} & \\ & & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & & C_{12} \\ & C_{21} & \\ & & C_{22} \end{bmatrix}$$



Contoh 11. Misalkan matriks A adalah sebagai berikut:

$$A = \begin{bmatrix} 3 & 4 & 8 & 16 \\ 21 & 5 & 12 & 10 \\ 5 & 1 & 2 & 3 \\ 45 & 9 & 0 & -1 \end{bmatrix}$$

Matriks A dibagi menjadi 4 upa-matriks 2 x 2:

$$A_{11} = \begin{bmatrix} 3 & 4 \\ 21 & 5 \end{bmatrix} \quad A_{12} = \begin{bmatrix} 8 & 16 \\ 12 & 10 \end{bmatrix} \quad A_{21} = \begin{bmatrix} 5 & 1 \\ 45 & 9 \end{bmatrix} \quad A_{22} = \begin{bmatrix} 2 & 3 \\ 0 & -1 \end{bmatrix}$$

LANJUTAN

Pseudo-code perkalian matriks dengan algoritma divide and conquer:

```
function KaliMatriks2(A, B : Matriks, n : integer) → Matriks
{ Memberikan hasil kali matriks A dan B yang berukuran n × n. }
Deklarasi
i, j, k : integer
A11, A12, A21, A22, B11, B12, B21, B22, C11, C12, C21, C22 : Matriks

Algoritma:
if n = 1 then { matriks berukuran 1 x 1 atau sebagai scalar }
    return A * B { perkalian dua buah scalar biasa }
else
    Bagi A menjadi A11, A12, A21, dan A22 yang masing-masing berukuran n/2 x n/2
    Bagi B menjadi B11, B12, B21, dan B22 yang masing-masing berukuran n/2 x n/2
    C11 ← KaliMatriks2(A11, B11, n/2) + KaliMatriks2(A12, B21, n/2)
    C12 ← KaliMatriks2(A11, B12, n/2) + KaliMatriks2(A12, B22, n/2)
    C21 ← KaliMatriks2(A21, B11, n/2) + KaliMatriks2(A22, B21, n/2)
    C22 ← KaliMatriks2(A21, B12, n/2) + KaliMatriks2(A22, B22, n/2)
    return C { C adalah gabungan C11, C12, C21, C22 }
endif
```

LANJUTAN

Pseudo-code untuk operasi penjumlahan (+):

$$\begin{aligned}C11 &= A11 \cdot B11 + A12 \cdot B21 \\C12 &= A11 \cdot B12 + A12 \cdot B22 \\C21 &= A21 \cdot B11 + A22 \cdot B21 \\C22 &= A21 \cdot B12 + A22 \cdot B22\end{aligned}$$

```
function Tambah(A, B : Matriks, n : integer) → Matriks
{ Memberikan hasil penjumlahkan dua buah matriks, A dan B, yang berukuran n × n }
```

Deklarasi
i, j, k : integer

Algoritma:

```
for i ← 1 to n do
    for j ← 1 to n do
        C[i, j] ← A[i, j] + B[i, j]
    endfor
endfor
```

```
return C
```

Kompleksitas algoritmanya: $O(n^2)$

LANJUTAN

- Kompleksitas waktu perkalian matriks dengan *divide and conquer*:

$$T(n) = \begin{cases} a & , n=1 \\ 8T(n/2) + cn^2 & , n>1 \end{cases}$$

$$\begin{aligned} C11 &= A11 \cdot B11 + A12 \cdot B21 \\ C12 &= A11 \cdot B12 + A12 \cdot B22 \\ C21 &= A21 \cdot B11 + A22 \cdot B21 \\ C22 &= A21 \cdot B12 + A22 \cdot B22 \end{aligned}$$

Menurut Teorema Master, $T(n) = aT(n/b) + cn^d$, diperoleh $a = 8$, $b = 2$, $d = 2$, dan memenuhi $a > b^d$ (yaitu $8 > 2^2$) maka relasi rekurens

$$T(n) = 8T(n/2) + cn^2$$

memenuhi case 3 (jika $a > b^d$)

sehingga

$$T(n) = O(n^{2\log 8}) = O(n^3)$$

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

- Hasil ini tidak memberi perbaikan kompleksitas dibandingkan dengan algoritma *brute force*.
- Dapatkah kita membuat algoritma perkalian matriks yang lebih baik?

Algoritma Perkalian Matriks Strassen

- Ditemukan oleh Volker Strassen, seorang matematikawan Jerman
- Ideanya adalah mengurangi jumlah operasi kali. Operasi kali lebih ‘mahal’ ongkos komputasinya dibandingkan dengan operasi penjumlahan.
- Empat persamaan perkalian upamatriks (*sub-matrix*) terdahulu:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

terdapat 8 operasi perkalian (\cdot) dan 4 operasi penjumlahan ($+$) upamatriks.

- Strassen memanipulasi empat persamaan di atas sedemikian rupa sehingga jumlah operasi kali berkurang menjadi 7 (dengan konsekuensi operasi penjumlahan menjadi bertambah).

LANJUTAN

- Caranya adalah melakukan perhitungan intermediate sebagai berikut:

$$M1 = (A12 - A22)(B21 + B22)$$

$$M2 = (A11 + A22)(B11 + B22)$$

$$M3 = (A11 - A21)(B11 + B12)$$

$$M4 = (A11 + A12)B22$$

$$M5 = A11 (B12 - B22)$$

$$M6 = A22 (B21 - B11)$$

$$M7 = (A21 + A22)B11$$

maka,

$$C11 = M1 + M2 - M4 + M6$$

$$C12 = M4 + M5$$

$$C21 = M6 + M7$$

$$C22 = M2 - M3 + M5 - M7$$

Terdapat 7 operasi x dan 18 operasi +

LANJUTAN

```
function KaliMatriksStrassen(A, B : Matriks, n : integer) → Matriks
{ Memberikan hasil kali matriks A dan B yang berukuran n × n. }
Deklarasi
i, j, k : integer
A11, A12, A21, A22, B11, B12, B21, B22, C11, C12, C21, C22, M1, M2, M3, M4, M5, M6, M7 : Matriks

Algoritma:
if n = 1 then { matriks berukuran 1 x 1 atau sebagai scalar }
    return A * B { perkalian dua buah scalar biasa }
else
    Bagi A menjadi A11, A12, A21, dan A22 yang masing-masing berukuran n/2 x n/2
    Bagi B menjadi B11, B12, B21, dan B22 yang masing-masing berukuran n/2 x n/2
    M1 ← KaliMatriksStrassen(A12 – A22, B21 + B22, n/2)
    M2 ← KaliMatriksStrassen (A11 + A22, B11 + B22 , n/2)
    M3 ← KaliMatriksStrassen (A11 – A21, B11 + B12 , n/2)
    M4 ← KaliMatriksStrassen (A11 + A12, B22 , n/2)
    M5 ← KaliMatriksStrassen (A11, B12 – B22 , n/2)
    M6 ← KaliMatriksStrassen (A22, B21 – B11 , n/2)
    M7 ← KaliMatriksStrassen (A21 + A22, B11 , n/2)
    C11 ← M1 + M2 – M4 + M6
    C12 ← M4 + M5
    C21 ← M6 + M7
    C22 ← M2 – M3 + M5 – M7
    return C { C adalah gabungan C11, C12, C21, C22 }
endif
```

LANJUTAN

- Kompleksitas algoritmanya menjadi:

$$T(n) = \begin{cases} a & , n=1 \\ 7T(n/2) + cn^2 & , n>1 \end{cases}$$

Bila diselesaikan dengan Teorema Master, $T(n) = aT(n/b) + cn^d$, diperoleh $a = 7$, $b = 2$, $d = 2$, dan memenuhi $a > b^d$ (yaitu $7 > 2^2$) maka relasi rekurens

$$T(n) = 7T(n/2) + cn^2$$

memenuhi case 3 (jika $a > b^d$) sehingga

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

$$T(n) = O(n^{2\log 7}) = O(n^{2.81})$$

- Lebih baik dari perkalian matriks secara *divide and conquer* sebelumnya yang $O(n^3)$

Perkalian Bilangan Bulat yang Besar

- Bilangan bulat yang besar (*big number*) adalah bilangan bulat dengan panjang n angka atau n bit.

Contoh: 564389018149014329871520, 1000011011010100100110010101, ...

- Bahasa-bahasa pemrograman memiliki keterbatasan dalam merepresentasikan bilangan bulat yang besar.
- Dalam Bahasa C misalnya, tipe bilangan bulat hanya `char` (8 bit), `int`, (16 bit) dan `long` (32 bit).
- Untuk bilangan bulat yang lebih dari 32 bit, kita harus membuat tipe sendiri dan mendefinisikan primitif operasi-operasi aritmetika di dalamnya (+, -, *, /, dll)

LANJUTAN

- Di sini hanya akan dibahas bagaimana algoritma melakukan operasi perkalian bilangan bulat yang besar.

Contoh: $1765420875208345186 \times 75471199736308361736432$

- **Persoalan:** Misalkan bilangan bulat X dan Y yang panjangnya n angka (atau n bit):

$$X = x_1x_2x_3 \dots x_n$$

$$Y = y_1y_2y_3 \dots y_n$$

Hitunglah hasil kali X dengan Y .

LANJUTAN

Contoh 12. Misalkan,

$$X = 1234 \quad (n = 4)$$

$$Y = 5678 \quad (n = 4)$$

Cara klasik mengalikan X dan Y (*brute force*):

$$\begin{array}{r} X \times Y = 1234 \\ \underline{5678 \times} \\ 9872 \\ 8368 \\ 7404 \\ \hline 6170 \quad + \\ 7006652 \quad (7 \text{ angka}) \end{array}$$

LANJUTAN

Pseudo-code algoritma perkalian bilangan besar secara brute force:

```
function Kali1(X, Y : LongInteger, n : integer) → LongInteger
{ Memberikan hasil perkalian X dan Y, masing-masing panjangnya n digit dengan algoritma brute force. }

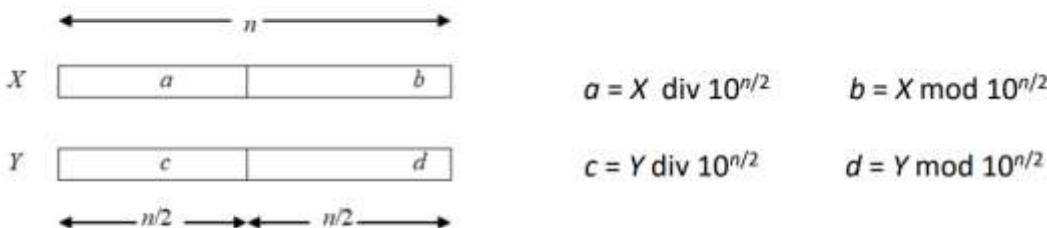
Deklarasi
temp, AngkaSatuan, AngkaPuluhan : integer

Algoritma:
for setiap angka  $y_i$  dari  $y_n, y_{n-1}, \dots, y_1$  do
    AngkaPuluhan ← 0
    for setiap angka  $x_j$  dari  $x_n, x_{n-1}, \dots, x_1$  do
        temp ←  $x_j * y_i$ 
        temp ← temp + AngkaPuluhan
        AngkaSatuan ← temp mod 10
        AngkaPuluhan ← temp div 10
        write(AngkaSatuan)
    endfor
endfor
Z ← Jumlahkan semua hasil perkalian dari atas ke bawah
return Z
```

Kompleksitas algoritma: $O(n^2)$

LANJUTAN

Penyelesaian dengan algoritma *divide and conquer*



X dan Y dapat dinyatakan dalam a, b, c , dan d sebagai

$$X = a \cdot 10^{n/2} + b \quad \text{dan} \quad Y = c \cdot 10^{n/2} + d$$

Perkalian X dengan Y dinyatakan sebagai

$$\begin{aligned} X \cdot Y &= (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d) \\ &= ac \cdot 10^n + ad \cdot 10^{n/2} + bc \cdot 10^{n/2} + bd \\ &= ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd \end{aligned}$$



Contoh 13: Misalkan $n = 6$ dan $X = 346769$ dan $Y = 279431$, maka

$$X = 346769 \rightarrow a = 346, b = 769 \rightarrow X = 346 \cdot 10^3 + 769$$

$$Y = 279431 \rightarrow c = 279, d = 431 \rightarrow Y = 279 \cdot 10^3 + 431$$

Perkalian X dengan Y dinyatakan sebagai

$$X \cdot Y = (346 \cdot 10^3 + 769) \cdot (279 \cdot 10^3 + 431)$$

$$= (346)(279) \cdot 10^6 + (346)(431) \cdot 10^3 + (769)(279) \cdot 10^3 + (769)(431)$$

$$= (346)(279) \cdot 10^6 + ((346)(431) + (769)(279)) \cdot 10^3 + (769)(431)$$

↑
perkalian
bilangan besar

↑
perkalian
bilangan besar

↑
perkalian
bilangan besar

↑
perkalian
bilangan besar

LANJUTAN

Pseudo-code algoritma perkalian bilangan besar dengan algoritma divide and conquer:

```
function Kali2(X, Y : LongInteger, n : integer) → LongInteger
{ Memberikan hasil perkalian X dan Y, masing-masing panjangnya n digit dengan algoritma divide and conquer. }

Deklarasi
a, b, c, d : LongInteger
s : integer

Algoritma:
if n = 1 then
    return X*Y { perkalian skalar biasa }
else
    s ← n div 2 { bagidua pada posisi s }
    a ← X div  $10^s$ 
    b ← X mod  $10^s$ 
    c ← Y div  $10^s$ 
    d ← Y mod  $10^s$ 
    return Kali2(a, c, s)* $10^{2s}$  + Kali2(b, c, s)* $10^s$  + Kali2(a, d, s)* $10^s$  + Kali2(b, d, s)
endif
```

LANJUTAN

$$Kali2(a, c, s)^*10^{2s} + Kali2(b, c, s)^*10^s + Kali2(a, d, s)^*10^s + Kali2(b, d, s)$$

- Kompleksitas algoritma *Kali2*:

$$T(n) = \begin{cases} a & , n=1 \\ 4T(n/2) + cn & , n>1 \end{cases}$$

Catatan: Menghitung 10^s dan 10^{2s} tidak perlu dilakukan dengan memangkatkan 10 sebanyak s kali atau $2s$ kali, tetapi cukup dilakukan dengan menambahkan 0 sebanyak s kali atau $2s$ kali

- Dengan menggunakan Teorema Master (tunjukkan!), diperoleh:

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

- Ternyata, perkalian dengan algoritma *Divide and Conquer* seperti di atas belum memperbaiki kompleksitas waktu algoritmanya, sama seperti perkalian secara *brute force*.
- Adakah algoritma perkalian yang lebih baik?

Perbaikan: Algoritma Perkalian Karatsuba

- Ditemukan Anatolii Alexeevitch Karatsuba, seorang matematikawan Rusia pada tahun 1962
- Ideanya sama seperti pada perkalian matriks Strassen, yaitu dengan mengurangi jumlah operasi kali.
- Persamaan perkalian X dan Y terdahulu:
$$X \cdot Y = ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd$$
terdapat 4 operasi perkalian dan 3 operasi penjumlahan bilangan besar.
- Karatsuba memanipulasi persamaan di atas sedemikian rupa sehingga jumlah operasi kali berkurang menjadi 3 (dengan konsekuensi operasi penjumlahan menjadi bertambah).

LANJUTAN

Misalkan

$$r = (a + b)(c + d) = ac + (ad + bc) + bd$$

maka,

$$(ad + bc) = r - ac - bd = (a + b)(c + d) - ac - bd$$

Dengan demikian, perkalian X dan Y dimanipulasi menjadi

$$\begin{aligned} X \cdot Y &= ac \cdot 10^n + (ad + bc) \cdot 10^{n/2} + bd \\ &= \underbrace{ac}_{p} \cdot 10^n + \underbrace{\{(a + b)(c + d) - ac - bd\}}_{r} \cdot 10^{n/2} + \underbrace{bd}_{q} \end{aligned}$$

Sekarang hanya terdapat tiga operasi kali, yaitu p , q , dan r

LANJUTAN

Pseudo-code algoritma perkalian bilangan besar dengan algoritma Karatsuba:

```
function Kali3(X, Y: LongInteger, n : integer) → LongInteger
  / Memberikan hasil perkalian X dan Y, masing-masing panjangnya n digit dengan algoritma Karatsuba. /

Deklarasi
  a, b, c, d, p, q, r : LongInteger
  s : integer

Algoritma:
  if n = 1 then
    return X * Y  { perkalian skalar biasa }
  else
    s ← n div 2    { bagidua pada posisi s }
    a ← X div 10s
    b ← X mod 10s
    c ← Y div 10s
    d ← Y mod 10s
    p ← Kali3(a, c, s)
    q ← Kali3(b, d, s)
    r ← Kali3(a + b, c + d, s)
    return p * 102s + (r - p - q) * 10s + q
  endif
```

LANJUTAN

- Kompleksitas algoritmanya:

$T(n) = \text{tiga buah perkalian dua integer yang berukuran } n/2 + \text{penjumlahan integer berukuran } n/2$

$$T(n) = \begin{cases} a & , n=1 \\ 3T(n/2) + cn & , n>1 \end{cases}$$

- Bila diselesaikan dengan Teorema Master, $T(n) = aT(n/b) + cn^d$, diperoleh $a = 3$, $b = 2$, $d = 1$, dan memenuhi $a > b^d$ (yaitu $3 > 2^1$) maka relasi rekurens $T(n) = 3T(n/2) + cn$ memenuhi case 3 (jika $a > b^d$), sehingga

$$T(n) = O(n^{2\log 3}) = O(n^{1.59})$$

- Ini lebih baik daripada kompleksitas waktu algoritma perkalian sebelumnya ($O(n^2)$).

Perkalian Polinom

- **Persoalan:**

Diberikan dua buah polinom derajat n

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

$$B(x) = b_0 + b_1x + b_2x^2 + \dots + b_nx^n$$

Hitunglah perkalian $A(x)B(x)$

Contoh 14:

$$A(x) = 1 + 2x + 3x^2$$

$$B(x) = 3 + 2x + 2x^2$$

$$A(x)B(x) = (1 + 2x + 3x^2)(3 + 2x + 2x^2) = 3 + 8x + 15x^2 + 10x^3 + 6x^4$$

LANJUTAN

Penyelesaian secara *brute force*

- Misalkan:

$$A(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n = \sum_{i=0}^n a_i x^i$$

$$B(x) = b_0 + b_1x + b_2x^2 + \dots + b_nx^n = \sum_{i=0}^n b_i x^i$$

- Maka algoritma *brute force* adalah mengalikan kedua polinom secara langsung:

$$C(x) = A(x)B(x) = \sum_{k=0}^{2n} c_k x^k , \text{ di sini } c_k = \sum_{0 \leq i,j \leq n, i+j=k} a_i b_j , 0 \leq k \leq 2n$$

- Dengan menggunakan formula perkalian di atas, ada dua buah kalang (*loop*), kalang pertama dari $k = 0$ sampai $2n$, kalang kedua dari $i = 0$ sampai k .
- Dapat dengan mudah ditunjukkan bahwa operasi perkalian adalah $O(n^2)$ dan operasi penjumlahan $O(n^2)$. Kompleksitas algoritma seluruhnya adalah $O(n^2)$.

LANJUTAN

Penyelesaian dengan algoritma *divide and conquer*

- Bagidua $A(x)$ menjadi $A_0(x)$ dan $A_1(x)$, masing-masing $n/2$ suku

$$A_0(x) = a_0 + a_1x + a_2x^2 + \dots + a_{\lfloor n/2 \rfloor - 1}x^{\lfloor n/2 \rfloor - 1}$$

$$A_1(x) = a_{\lfloor n/2 \rfloor} + a_{\lfloor n/2 \rfloor + 1}x + a_{\lfloor n/2 \rfloor + 2}x^2 + \dots + a_nx^{n - \lfloor n/2 \rfloor}$$

maka

$$A(x) = A_0(x) + A_1(x)x^{\lfloor n/2 \rfloor}$$

- Dengan cara yang sama untuk $B(x)$ menjadi $B_0(x)$ dan $B_1(x)$ sedemikian sehingga

$$B(x) = B_0(x) + B_1(x)x^{\lfloor n/2 \rfloor}$$

- Maka, perkalian $A(x)$ dan $B(x)$ ditulis menjadi

$$A(x)B(x) = A_0(x)B_0(x) + \{A_0(x)B_1(x) + A_1(x)B_0(x)\}x^{\lfloor n/2 \rfloor} + A_1(x)B_1(x)x^{2\lfloor n/2 \rfloor}$$

→ Terdapat empat buah perkalian upa-polinom

LANJUTAN

Contoh 15: $A(x) = 2 + 5x + 3x^2 + x^3 - x^4$ dan $B(x) = 1 + 2x + 2x^2 + 3x^3 + 6x^4$

Bagidua $A(x)$ menjadi:

$$A_0(x) = 2 + 5x \text{ dan } A_1(x) = 3 + x - x^2 \text{ sehingga } A(x) = A_0(x) + A_1(x)x^2$$

Bagidua $B(x)$ menjadi:

$$B_0(x) = 1 + 2x \text{ dan } B_1(x) = 2 + 3x + 6x^2 \text{ sehingga } B(x) = B_0(x) + B_1(x)x^2$$

Maka

$$\begin{aligned} A(x)B(x) &= A_0(x)B_0(x) + (A_0(x)B_1(x) + A_1(x)B_0(x))x^2 + A_1(x)B_1(x)x^4 \\ &= (2 + 5x)(1 + 2x) + \{ (2 + 5x)(2 + 3x + 6x^2) + (3 + x - x^2)(1 + 2x) \} x^2 + \\ &\quad (3 + x - x^2)(2 + 3x + 6x^2)x^4 \\ &= 2 + 9x + 10x^2 + (4 + 16x + 27x^2 + 30x^3 + 3 + 7x + x^2 - 2x^3)x^2 + \\ &\quad 6 + 11x + 19x^2 + 3x^3 - 6x^4 \\ &= 2 + 9x + 17x^2 + 23x^3 + 34x^4 + 39x^5 + 19x^6 + 3x^7 - 6x^8 \end{aligned}$$

LANJUTAN

$$A(x)B(x) = A_0(x)B_0(x) + (A_0(x)B_1(x) + A_1(x)B_0(x)) x^{\lfloor n/2 \rfloor} + A_1(x)B_1(x) x^{2\lfloor n/2 \rfloor}$$

```
function KaliPolinom(A, B : Polinom, n : integer) → Polinom
{ Memberikan hasil perkalian polinom A(x) dan B(x), masing-masing berderajat n dengan algoritma divide and conquer. }

Deklarasi
A0, A1, B0, B1 : Polinom
s : integer

Algoritma:
if n = 0 then
    return A * B { perkalian skalar biasa }
else
    s ← n div 2 { bagidua suku-suku polinom pada posisi s }
    A0 ← a0 + a1x + a2x2 + ... + as-1xs-1
    A1 ← as + as+1x + as+2x2 + ... + anxn-s
    B0 ← b0 + b1x + b2x2 + ... + bs-1xs-1
    B1 ← bs + bs+1x + bs+2x2 + ... + bnxn-s
    return KaliPolinom(A0, B0, s) + ( KaliPolinom(A0, B1, s) + KaliPolinom(A1, B0, s) ) * xs +
           KaliPolinom(A1, B1, s) * x2s
endif
```

LANJUTAN

Kompleksitas algoritmanya:

- Tinjau lagi $A(x)B(x) = A_0(x)B_0(x) + (A_0(x)B_1(x) + A_1(x)B_0(x))x^{\lfloor n/2 \rfloor} + A_1(x)B_1(x)x^{2\lfloor n/2 \rfloor}$
- Penjumlahan dua polinom berukuran $n/2$ (operator $+$ berwarna merah) kompleksitasnya $O(n)$ atau cn
- Kompleksitas algoritmanya: $T(n) = \begin{cases} a & , n = 0 \\ 4T(n/2) + cn & , n > 0 \end{cases}$
- Dengan menggunakan Teorema Master, $a = 4$, $b = 2$, $d = 1$, dan memenuhi $a > b^d$ (yaitu $4 > 2^1$) maka relasi rekurens $T(n) = 4T(n/2) + cn$ memenuhi case 3, sehingga
$$T(n) = O(n^{2\log 4}) = O(n^2)$$
- Hasil ini tidak memberi perbaikan dibandingkan dengan algoritma *brute force*.
- Dapatkah kita membuat algoritma perkalian polinom yang lebih baik?

LANJUTAN

Perbaikan:

- Ideya sama seperti pada metode perkalian Karatsuba, yaitu dengan mengurangi jumlah operasi kali.

- Persamaan perkalian $A(x)$ dan $B(x)$ sebelumnya:

$$A(x)B(x) = A_0(x)B_0(x) + (A_0(x)B_1(x) + A_1(x)B_0(x))x^{\lfloor n/2 \rfloor} + A_1(x)B_1(x)x^{2\lfloor n/2 \rfloor}$$

terdapat 4 operasi perkalian dan 3 operasi penjumlahan upa-polinom derajat $n/2$.

- Dengan memanipulasi persamaan di atas sedemikian rupa kita dapat mengurangi jumlah operasi kali berkurang menjadi 3 (dengan konsekuensi operasi penjumlahan menjadi bertambah).

LANJUTAN

- Definisikan

$$Y(x) = (A_0(x) + A_1(x)) \times (B_0(x) + B_1(x))$$

$$U(x) = A_0(x)B_0(x)$$

$$Z(x) = A_1(x)B_1(x)$$

- Maka

$$Y(x) - U(x) - Z(x) = A_0(x)B_1(x) + A_1(x)B_0(x)$$

sehingga

$$\begin{aligned} A(x)B(x) &= A_0(x)B_0(x) + (A_0(x)B_1(x) + A_1(x)B_0(x))x^{\lfloor n/2 \rfloor} + A_1(x)B_1(x)x^{2\lfloor n/2 \rfloor} \\ &= U(x) + \{Y(x) - U(x) - Z(x)\}x^{\lfloor n/2 \rfloor} + Z(x)x^{2\lfloor n/2 \rfloor} \end{aligned}$$

Sekarang hanya terdapat tiga operasi perkalian polinom, yaitu $U(x)$, $Y(x)$, dan $Z(x)$

LANJUTAN

$$A(x)B(x) = U(x) + \{ Y(x) - U(x) - Z(x) \} x^{\lfloor n/2 \rfloor} + Z(x) x^{2\lfloor n/2 \rfloor}$$

```
function KaliPolinom2(A, B : Polinom, n : integer) → Polinom
{ Memberikan hasil perkalian polinom A(x) dan B(x), masing-masing berderajat n dengan algoritma divide and conquer. }

Deklarasi
A0, A1, B0, B1, U, Y, Z : Polinom
s : integer

Algoritma:
if n = 0 then
    return A * B { perkalian skalar biasa }
else
    s ← n div 2 { bagidua suku-suku polinom pada posisi s }
    A0 ← a0 + a1x + a2x2 + ... + as-1xs-1
    A1 ← as + as+1x + as+2x2 + ... + anxn-s
    B0 ← b0 + b1x + b2x2 + ... + bs-1xs-1
    B1 ← bs + bs+1x + bs+2x2 + ... + bnxn-s
    Y ← KaliPolinom2(A0 + A1, B0 + B1, s)
    U ← KaliPolinom2(A0, B0, s)
    Z ← KaliPolinom2(A1, B1, s)
    return U + ( Y - U - Z ) * xs + Z * x2s
endif
```

LANJUTAN

Kompleksitas algoritmanya:

$$T(n) = \begin{cases} a & , n = 0 \\ 3T(n/2) + cn & , n > 0 \end{cases}$$

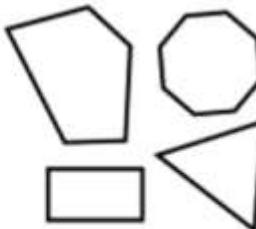
- Dengan menggunakan Teorema Master, $a = 3$, $b = 2$, $d = 1$, dan memenuhi $a > b^d$ (yaitu $3 > 2^1$) maka relasi rekurens $T(n) = 3T(n/2) + cn$ memenuhi case 3, sehingga

$$T(n) = O(n^{2\log_3 3}) = O(n^{1.59})$$

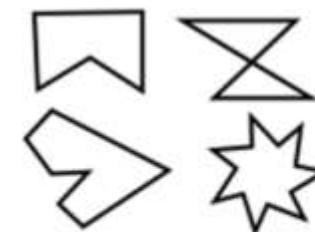
- Hasil ini lebih baik dibandingkan dengan algoritma *divide and conquer* sebelumnya

Convex Hull

- Salah satu hal penting dalam komputasi geometri adalah menentukan *convex hull* dari kumpulan titik.
- Himpunan titik pada bidang planar disebut *convex* jika untuk sembarang dua titik pada bidang tersebut (misal p dan q), seluruh segmen garis yang berakhir di p dan q berada pada himpunan tersebut.
- Contoh gambar 1 adalah poligon yang *convex*, sedangkan gambar 2 menunjukkan contoh yang *non-convex*.



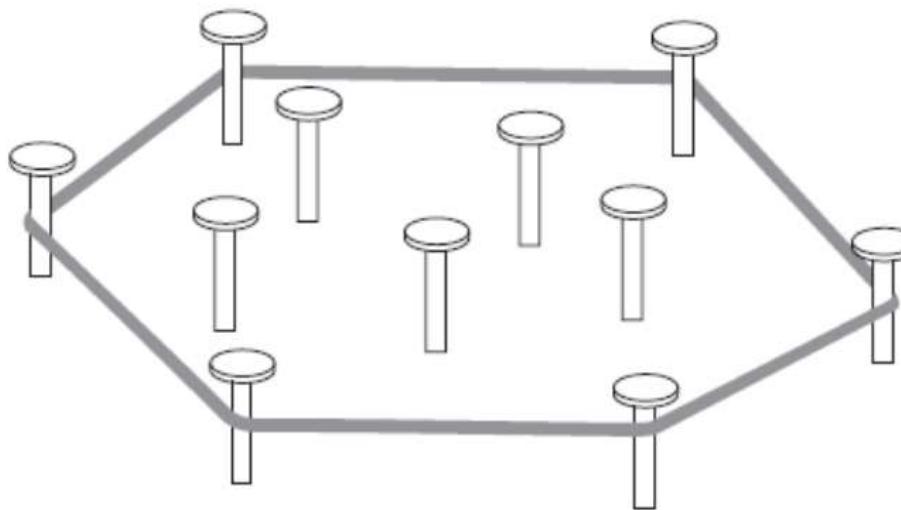
Gambar 1: convex



Gambar 2: non convex

LANJUTAN

- *Convex hull* dari himpunan titik S adalah himpunan *convex* terkecil (*convex polygon*) yang mengandung S

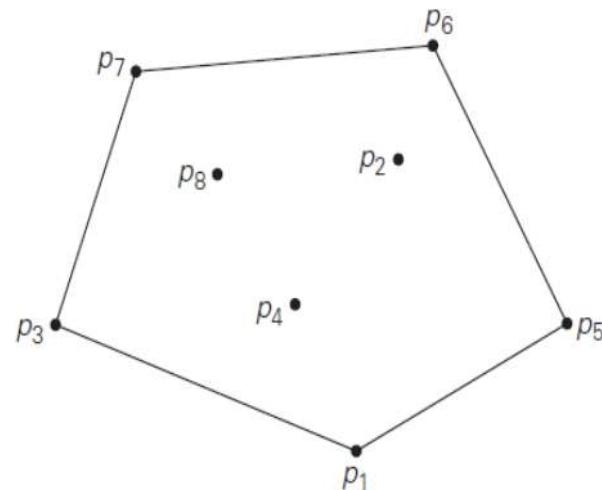


LANJUTAN

- Untuk dua titik, maka *convex hull* berupa garis yang menghubungkan 2 titik tersebut.
- Untuk tiga titik yang terletak pada satu garis, maka *convex hull* adalah sebuah garis yang menghubungkan dua titik terjauh.
- Sedangkan *convex hull* untuk tiga titik yang tidak terletak pada satu garis adalah sebuah segitiga yang menghubungkan ketiga titik tersebut.
- Untuk titik yang lebih banyak dan tidak terletak pada satu garis, maka *convex hull* berupa poligon *convex* dengan sisi berupa garis yang menghubungkan beberapa titik pada S.

LANJUTAN

- Contoh *convex hull* untuk delapan titik dapat dilihat pada gambar 3

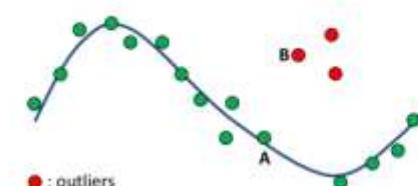


Gambar 3 Convex Hull untuk delapan titik

LANJUTAN

- Pemanfaatan dari *convex hull* ini cukup banyak.
- Pada animasi komputer, pemindahan suatu objek akan lebih mudah dengan memindahkan *convex hull* objek untuk *collision detection*.
- *Convex hull* juga dapat digunakan dalam persoalan optimasi, karena penentuan titik ekstrimnya dapat membatasi kandidat nilai optimal yang diperiksa.
- Pada bidang statistik, *convex hull* juga dapat mendeteksi *outliers* pada kumpulan data.

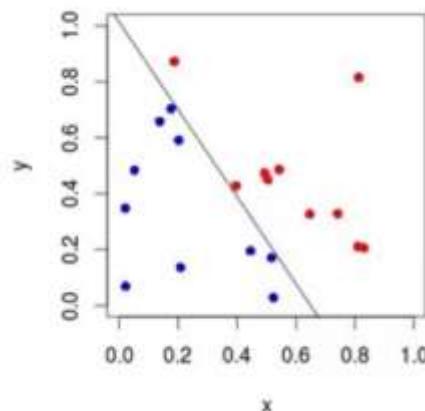
Outliers: titik-titik terluar



LANJUTAN

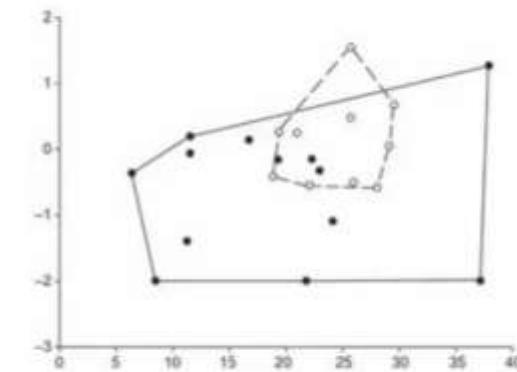
Pemanfaatan Convex Hull: Tes Linearly Separable Dataset

Alternatif 1. Visualisasi data:
analisis manual sulit utk
dimensi tinggi.



<https://www.quora.com/How-can-I-know-whether-my-data-is-linearly-separable>

Alternatif 2. Analisis convex-hull antar kelas tidak overlap
→ linearly separable dataset



LANJUTAN

Convex Hull dengan Divide and Conquer

- Tujuan: menemukan kumpulan titik ‘terluar’ yang membentuk *convex hull*
- Ide dasar: menggunakan algoritma *Quicksort*
- S : himpunan titik sebanyak n , dengan $n > 1$, yaitu titik $p_1(x_1, y_1)$ hingga $p_n(x_n, y_n)$ pada bidang kartesian dua dimensi
- Kumpulan titik diurutkan berdasarkan nilai absis yang menaik, dan jika ada nilai absis yang sama, maka diurutkan dengan nilai ordinat yang menaik
- p_1 dan p_n adalah dua titik ekstrim yang akan membentuk *convex hull* untuk kumpulan titik tersebut.

LANJUTAN

Ilustrasi:

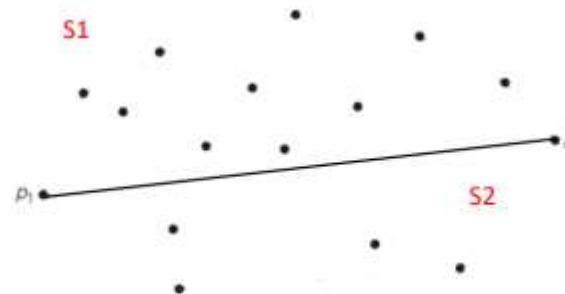


LANJUTAN

Ide Divide and Conquer

1. Garis yang menghubungkan p_1 dan p_n (p_1p_n) membagi S menjadi dua bagian yaitu $S1$ (kumpulan titik di sebelah kiri atau atas garis p_1p_n) dan $S2$ (kumpulan titik di sebelah kanan atau bawah garis p_1p_n).

Ilustrasi:



Untuk memeriksa apakah sebuah titik berada di sebelah kiri (atau atas) suatu garis yang dibentuk dua titik, gunakan penentuan determinan:

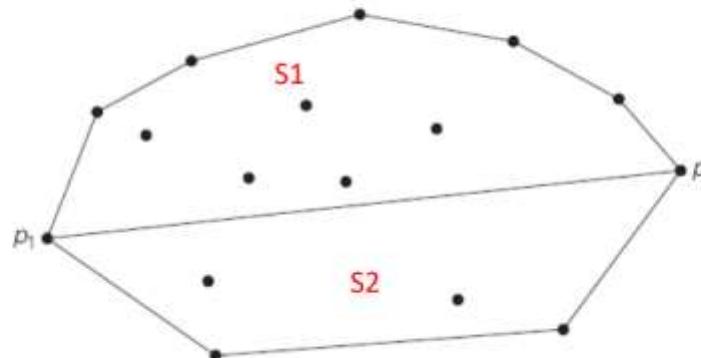
$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1y_2 + x_3y_1 + x_2y_3 - x_3y_2 - x_2y_1 - x_1y_3$$

Titik (x_3,y_3) berada di sebelah kiri dari garis $((x_1,y_1),(x_2,y_2))$ jika hasil determinan positif

LANJUTAN

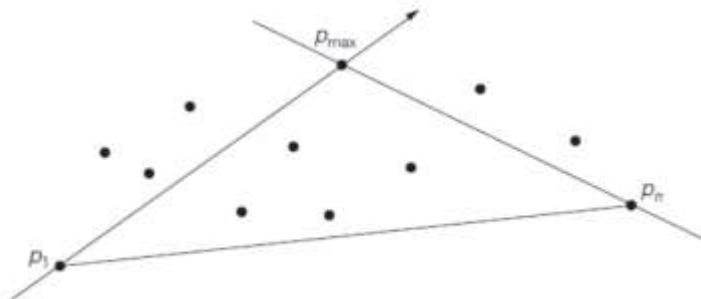
2. Semua titik pada S yang berada pada garis p_1p_n (selain titik p_1 dan p_n) tidak mungkin membentuk *convex hull*, sehingga bisa diabaikan dari pemeriksaan
3. Kumpulan titik pada S_1 bisa membentuk *convex hull* bagian atas, dan kumpulan titik pada S_2 bisa membentuk *convex hull* bagian bawah \rightarrow terapkan D & C

Ilustrasi:



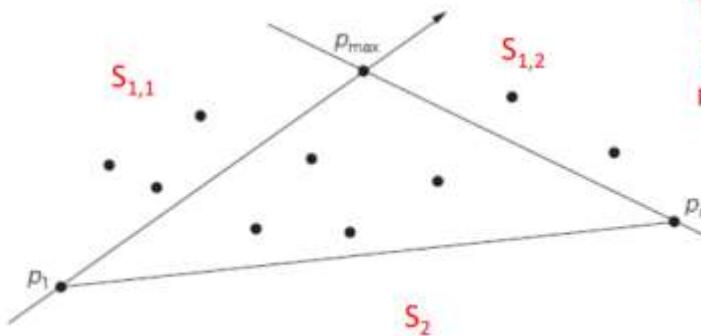
LANJUTAN

4. Untuk sebuah bagian (misal S_1), terdapat dua kemungkinan:
- Jika tidak ada titik lain selain S_1 , maka titik p_1 dan p_n menjadi pembentuk *convex hull* bagian S_1
 - Jika S_1 tidak kosong, pilih sebuah titik yang memiliki jarak terjauh dari garis $p_1 p_n$ (misal p_{\max}). Jika terdapat beberapa titik dengan jarak yang sama, pilih sebuah titik yang memaksimalkan sudut $p_{\max} p_1 p_n$
- Semua titik yang berada di dalam daerah segitiga $p_{\max} p_1 p_n$ diabaikan untuk pemeriksaan lebih lanjut



LANJUTAN

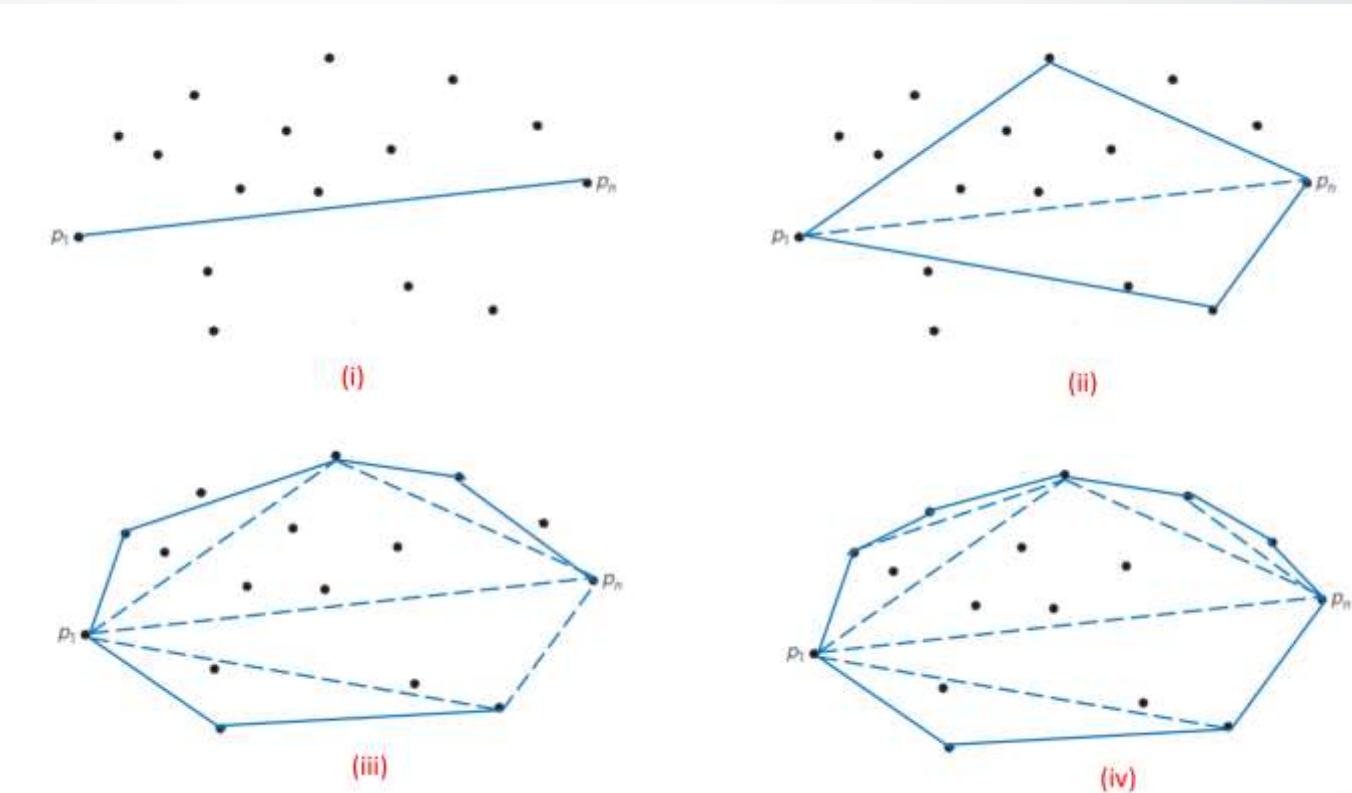
5. Tentukan kumpulan titik yang berada di sebelah kiri garis p_1p_{max} (menjadi bagian $S_{1,1}$), dan di sebelah kanan garis p_1p_{max} (menjadi bagian $S_{1,2}$)



Titik-titik di dalam segitiga
tidak diproses, karena tidak
mungkin membentuk convex hull

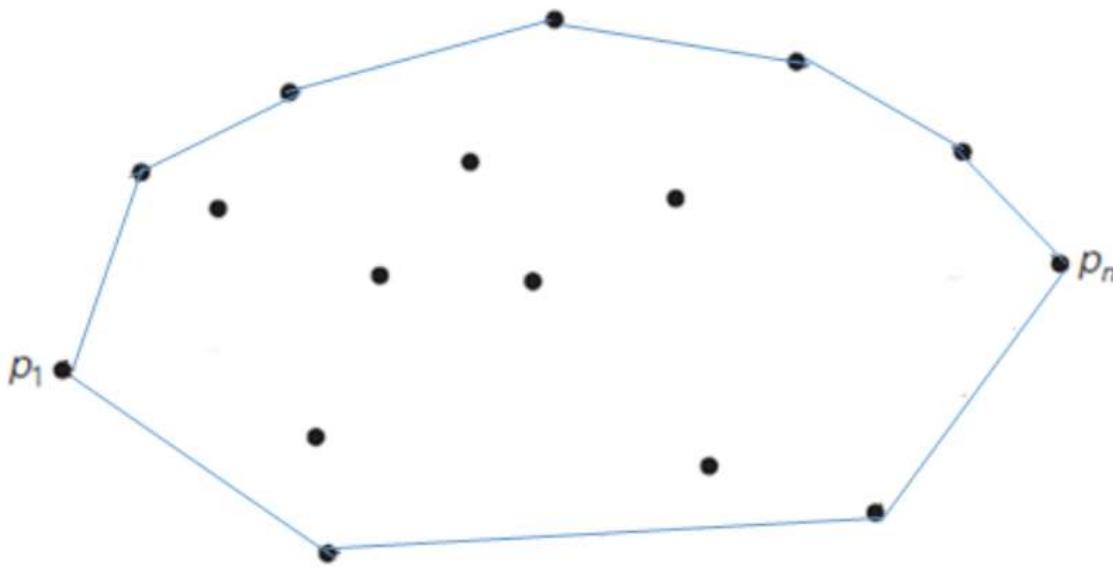
6. Lakukan hal yang sama (butir 4 dan 5) untuk bagian S_2 , hingga bagian 'kiri' dan 'kanan' kosong
7. Kembalikan pasangan titik yang dihasilkan

LANJUTAN



LANJUTAN

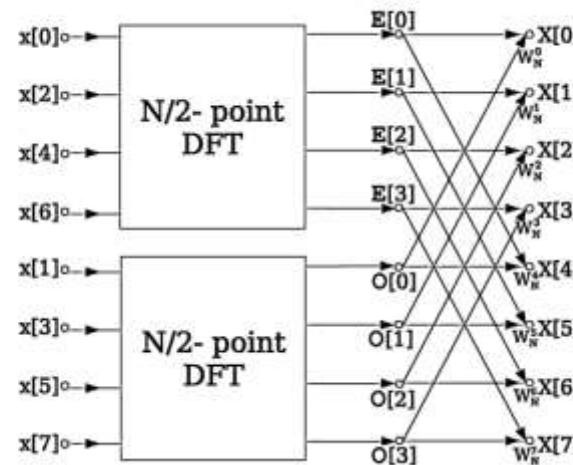
Hasil akhir:



LANJUTAN

Contoh algoritma lain yang menggunakan divide and conquer:

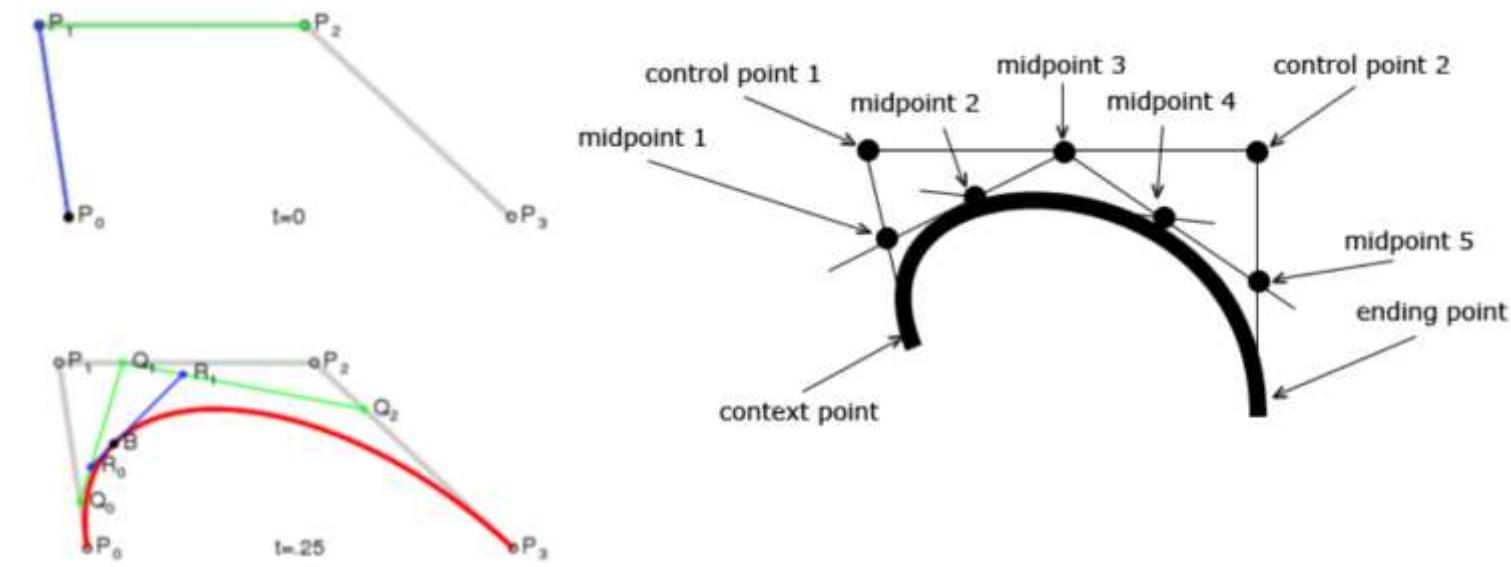
1. FFT - Fast Fourier Transform (Algoritma Cooley-Tukey)

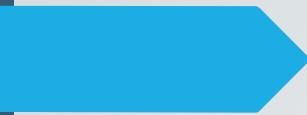


LANJUTAN

2. Kurva Bezier di dalam grafika computer (*computer graphics*)

→ Metode untuk menggambarkan kurva mulus (smooth)





TERIMA KASIH