Algoritma Decrease and Conquer

(Bagian 1)

Bahan Kuliah IF2211 Strategi Algoritma

Oleh: Rinaldi Munir

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika ITB
2024

Definisi *Decrease and Conquer*

- *Decrease and conquer*: metode perancangan algoritma dengan mereduksi persoalan menjadi dua upa-persoalan (*sub-problem*) yang lebih kecil, tetapi selanjutnya hanya memproses satu upa-persoalan saja.

- Berbeda dengan *divide and conquer* yang memproses *semua* upa-persoalan dan menggabungkan semua solusi setiap sub-persoalan.

- Di dalam literatur lama, semua algoritma yang membagi persoalan menjadi dua upa-persoalan yang lebih kecil dimasukkan ke kategori *divide and conquer*. Meskipun demikian, tidak kedua upa-persoalan hasil pembagian diselesaikan. Jika hanya satu upa-persoalan yang diselesaikan, maka tidak tepat dimasukkan sebagai algoritma *divide and conquer*. Mereka dikategorikan sebagai *decrease and conquer*

- Algoritma *decrease and conquer* terdiri dari dua tahapan:
  1. ***Decrease***: mereduksi persoalan menjadi beberapa persoalan yang lebih kecil (biasanya dua upa-persoalan).

  2. ***Conquer***: memproses satu upa-persoalan secara rekursif.

- Tidak ada tahap *combine* dalam *decrease and conquer*, karena hanya satu upa-persoalan yang diselesaikan.

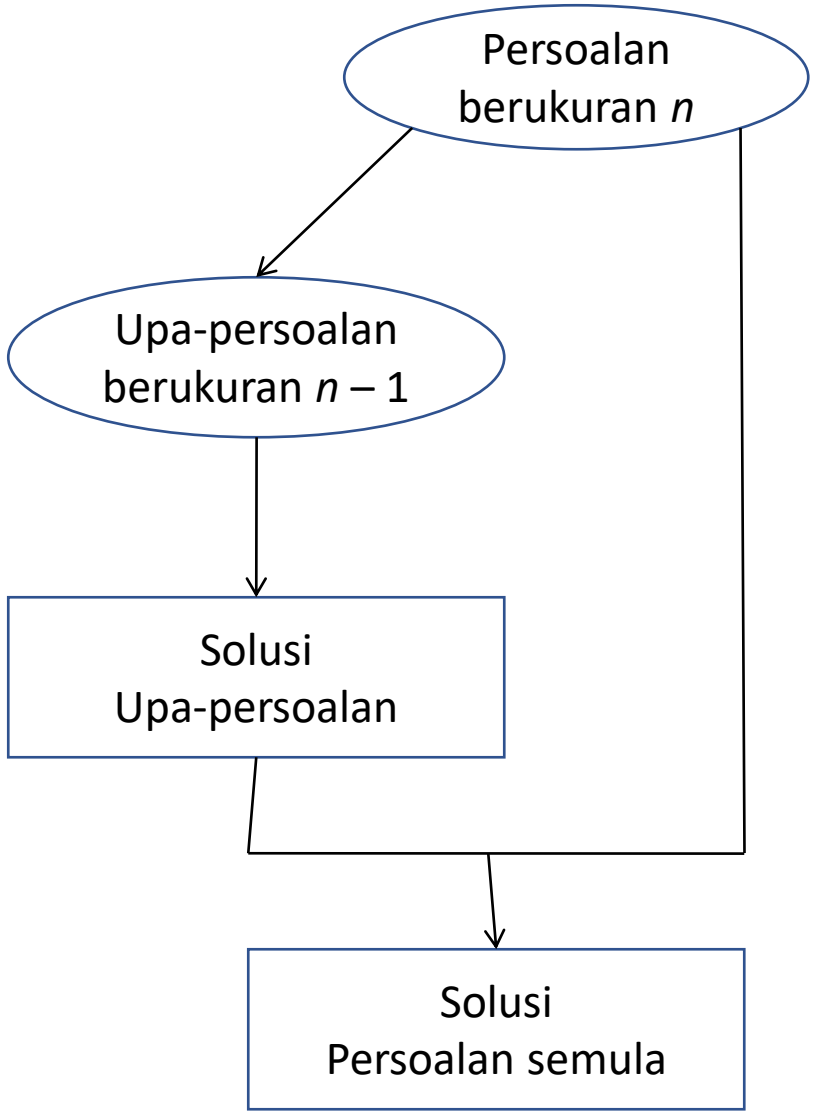
Tiga varian *decrease and conquer*:

1. ***Decrease by a constant***: ukuran instans persoalan direduksi sebesar konstanta yang sama setiap iterasi algoritma. Biasanya konstanta = 1.

2. ***Decrease by a constant factor***: ukuran instans persoalan direduksi sebesar faktor konstanta yang sama setiap iterasi algoritma. Biasanya faktor konstanta = 2.

3. ***Decrease by a variable size***: ukuran instans persoalan direduksi bervariasi pada setiap iterasi algoritma.

Decrease by a Constant



Contoh persoalan:

1. Perpangkatan a^n
2. Selection sort
3. Insertion sort

1. Persoalan perpangkatan a^n

Dengan metode *decrease and conquer*:

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

Contoh: $25^{10} = 25^9 \times 25$

Kompleksitas waktu (berdasarkan jumlah operasi kali):

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

Bila diselesaikan:

$$T(n) = T(n-1) + 1 = \ldots. = O(n)$$

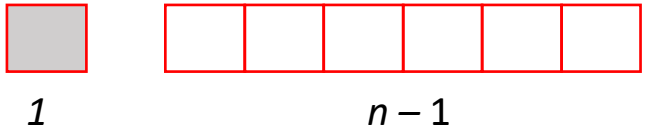
sama seperti algoritma *brute-force*.

```
function exp(a : real; n : integer) → real
{ memberikan hasil perpangkatan a^n }

Deklarasi
    k : integer

Algoritma:
    if n = 0 then
        return 1
    else
        return exp(a, n – 1) * a
    endif
```

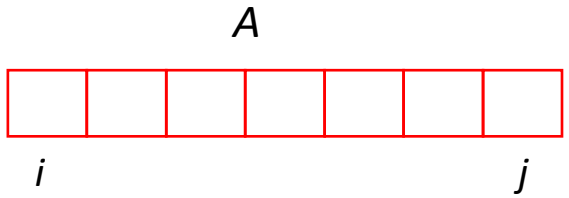
2. *Insertion Sort*

- *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.
- Algoritma ini dikategorikan sebagai *decrease and conquer* daripada sebagai *divide and conquer*

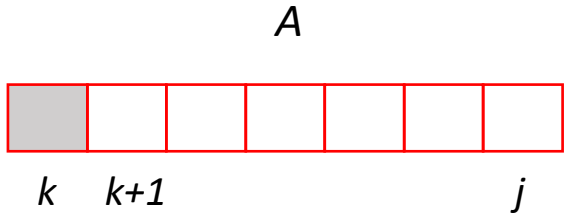


procedure *InsertionSort*(**input/output** *A* : *LarikInteger*, **input** *i*, *j* : **integer**)
{ Mengurutkan larik A[i..j] dengan algoritma Insertion Sort.
Masukan: Larik A[i..j] yang sudah terdefinisi elemen-elemennya
Luaran: Larik A[i..j] yang terurut
}

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
```



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

- Algoritma di atas dapat dianggap sebagai versi rekursif algoritma *Insertion Sort* yang umumnya dikenal dalam versi iterative.

- Prosedur *Merge* di dalam *Insertion sort* sama seperti *merge* di dalam algoritma Merge sort

- Selain menggunakan prosedur *Merge*, kita dapat mengganti *Merge* dengan prosedur penyisipan sebuah elemen pada larik yang terurut (seperti pada algoritma *Insertion Sort* versi iterative yang anda kenal).

Contoh 1: 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:

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

$= cn + \{ c\,(n-1) + T(n-2) \}$

$= cn + c(n-1) + \{ c(n-2) + T(n-3) \}$

$= cn + c(n-1) + c(n-2) + \{c(n-3) + T(n-4) \}$

$= \ldots$

$= cn + c(n-1) + c(n-2) + c(n-3) + \ldots + c2 + T(1)$

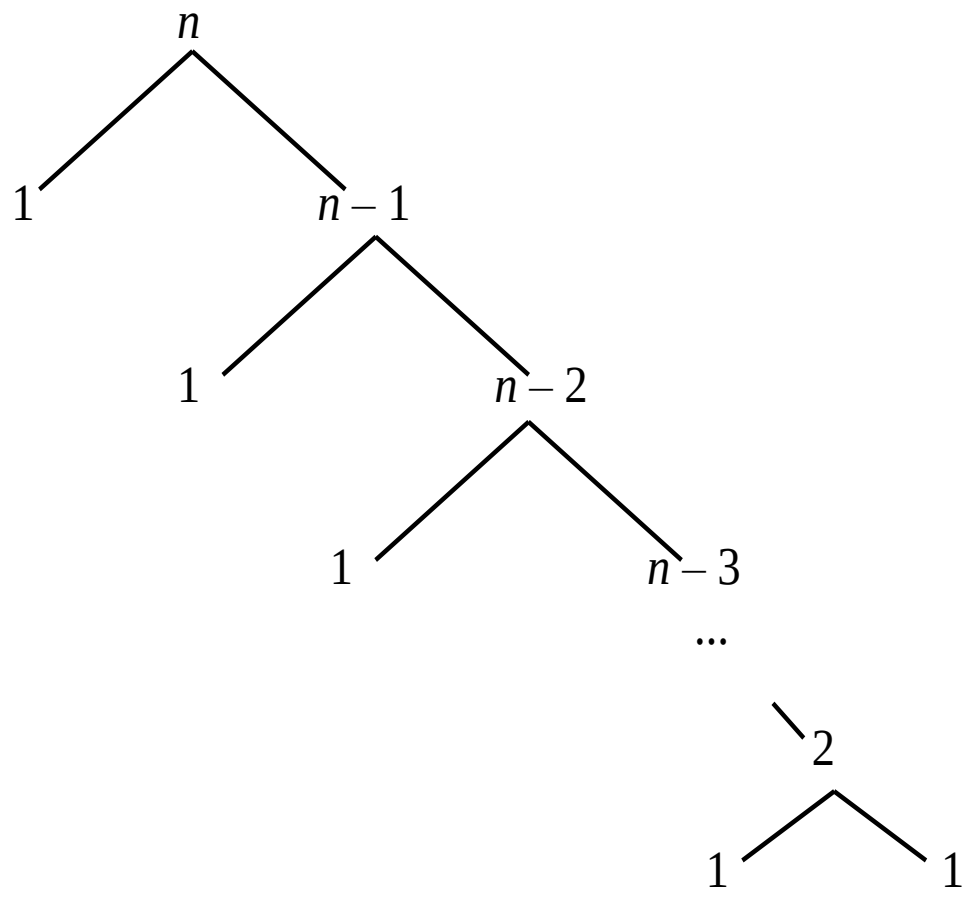
$= c\{ n + (n-1) + (n-2) + (n-3) + \ldots + 2 \} + a$

$= c\{ (n-1)(n+2)/2 \} + a$

$= cn^2/2 + cn/2 + (a - c)$

$= O(n^2) \rightarrow$ sama seperti kompleksitas algoritma *Insertion Sort* versi iteratif

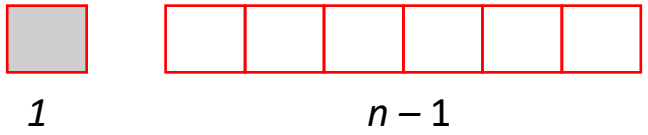
Pohon pembagian larik:



3. Selection Sort

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

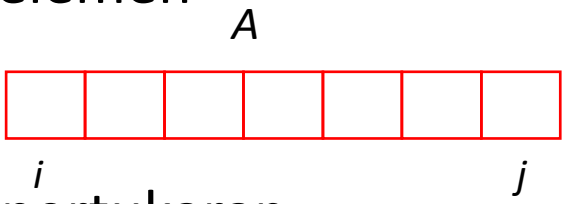
- yaitu, upalarik pertama hanya satu elemen, sedangkan upalarik kedua berukuran $n - 1$ elemen.



- *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.

- Algoritma ini dikategorikan sebagai *decrease and conquer* daripada sebagai *divide and conquer*

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

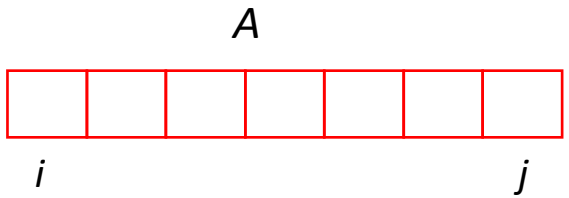


- lalu elemen minimum ditempatkan pada posisi *A*[*i*] dengan cara pertukaran.

```
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 }
      Partisi2(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 *Partisi2*(**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.
}



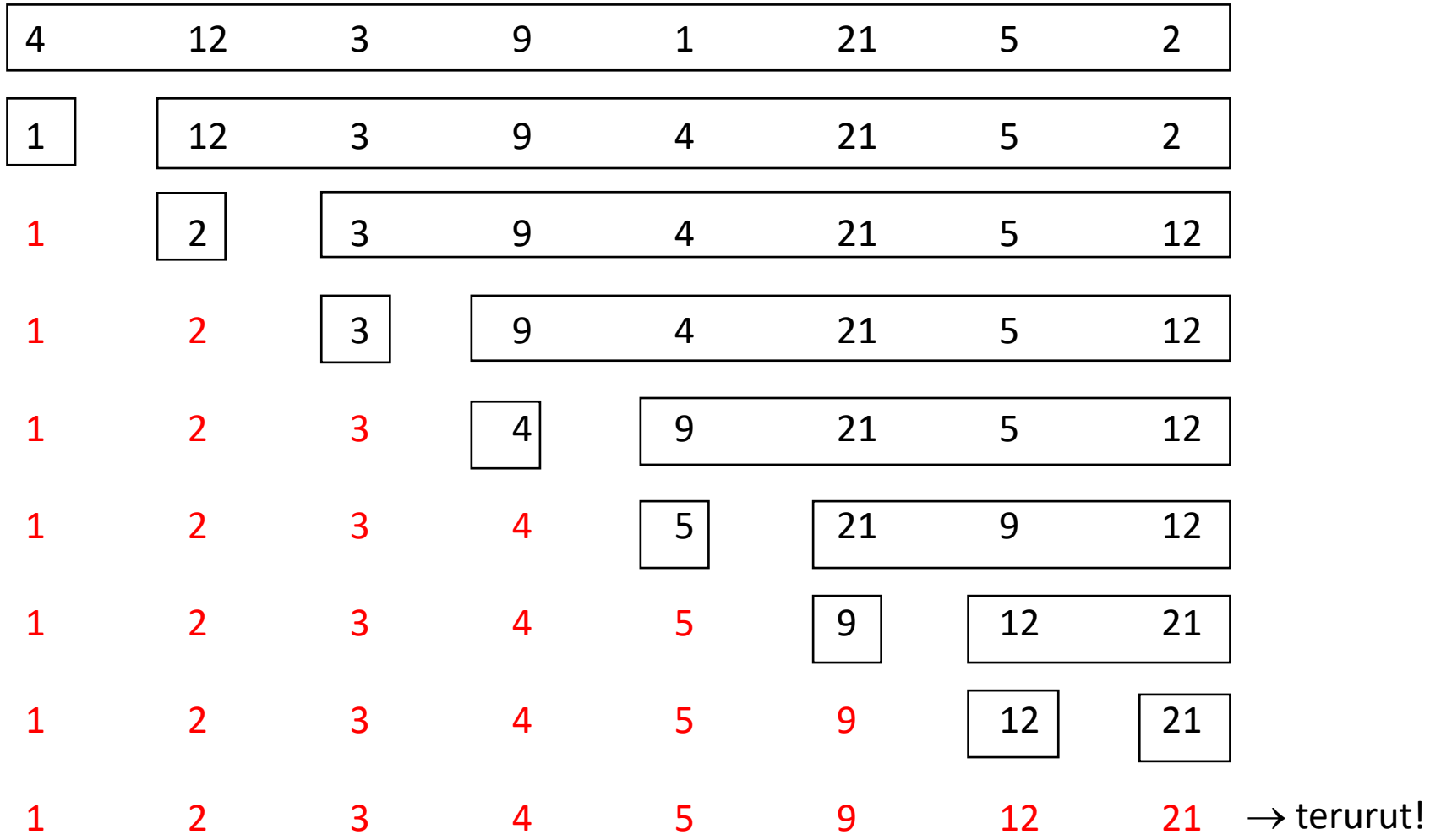
Deklarasi
idxmin, *k* : **integer**

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] }*

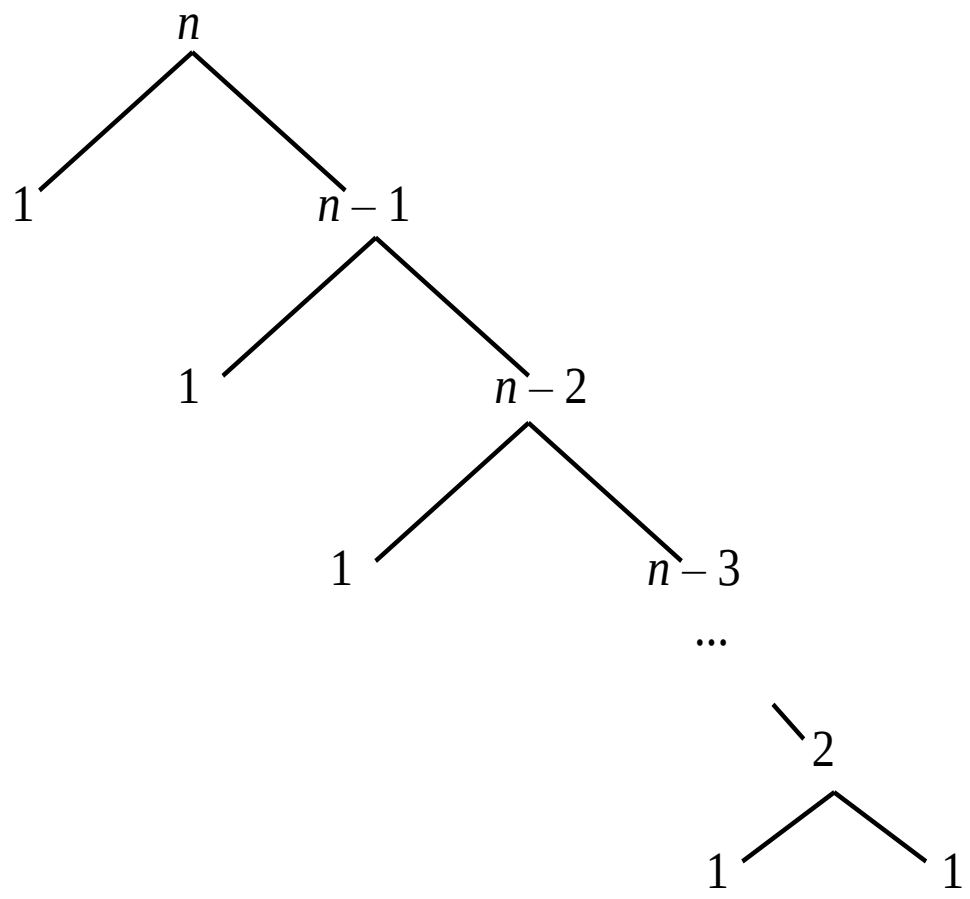
Contoh 2. Misalkan tabel A berisi elemen-elemen berikut:

4	12	3	9	1	21	5	2

Langkah-langkah pengurutan dengan *Selection Sort*:



Pohon pembagian larik:



Kompleksitas waktu algoritma *Selection Sort*:

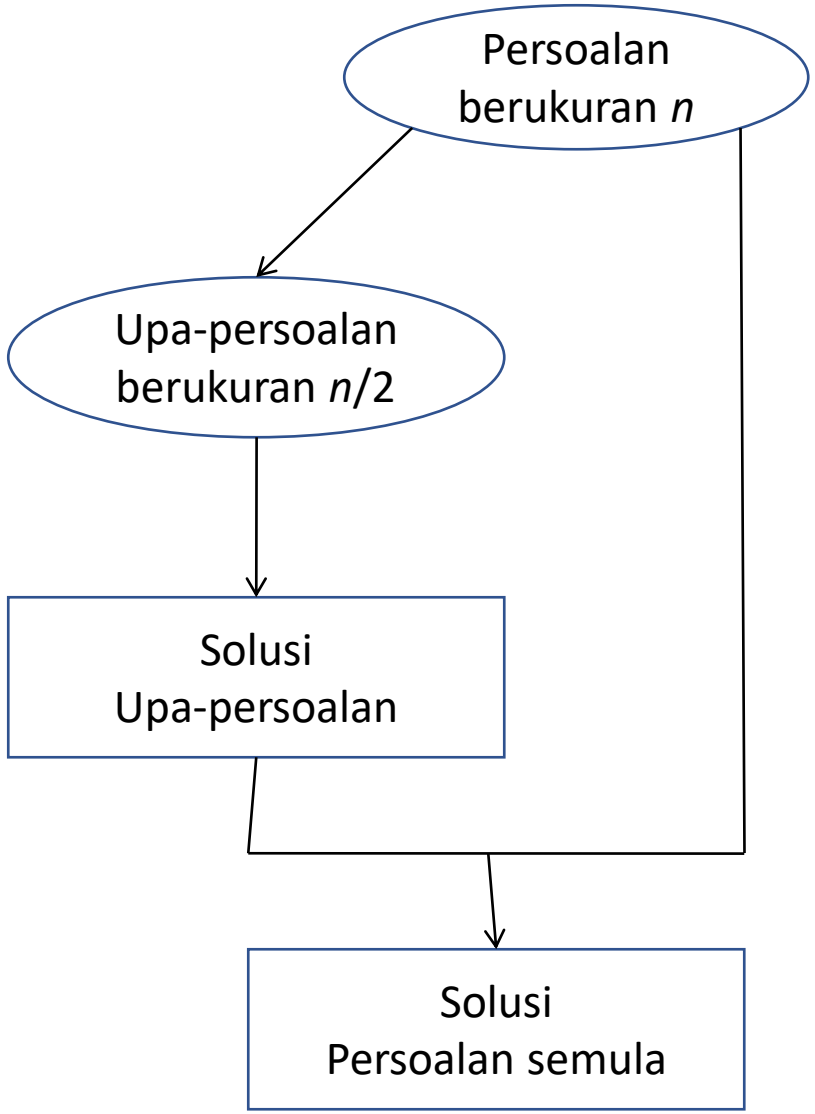
$$T(n) = \begin{cases} a & ,n = 1 \\ T(n-1) + cn & ,n > 1 \end{cases}$$

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*, $O(n^2)$)

Decrease by a Constant Factor



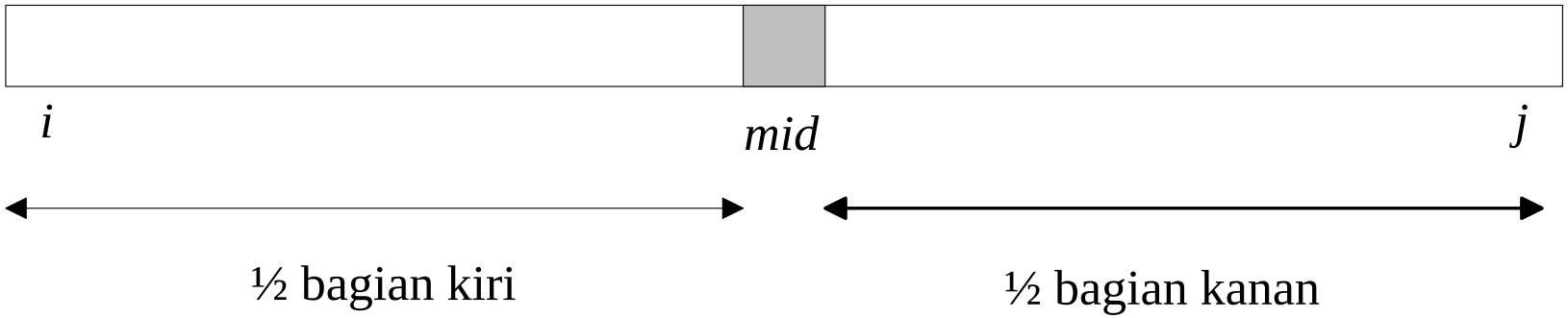
Contoh persoalan:

1. Binary search
2. Mencari koin palsu

4. *Binary search*

- Kondisi awal: larik *A* sudah terurut menaik

  K adalah nilai yang dicari di dalam larik

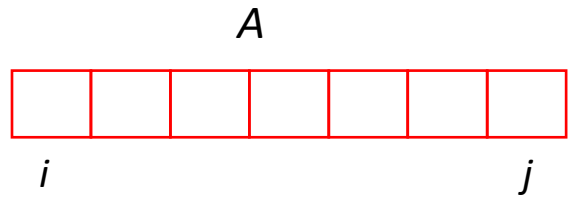


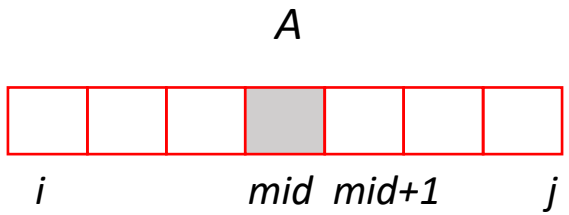
- Jika elemen tengah (*mid*) ≠ *k*, maka pencarian dilakukan hanya pada setengah bagian larik (kiri atau kanan)

- Ukuran persoalan selalu berkurang sebesar setengah ukuran semula. Hanya setengah bagain yang diproses, setengah bagian lagi tidak.

Algoritma *Binary Search* (Kasus 1: Larik sudah terurut menaik)

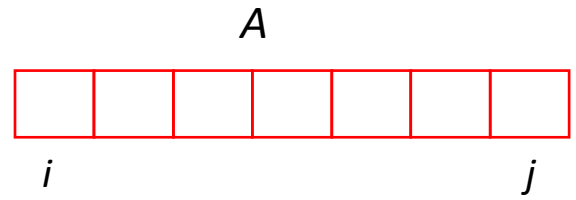
```
procedure binsearch(input A : LarikInteger,  i, j : integer; K : integer; output idx : integer)
{ Mencari elemen bernilai K di dalam larik A[i..j].
  Masukan: larik A sudah terurut menaik, K sudah terdefinisi nilainya
  Luaran: indek lariks sedemikian sehingga A[idx] = K
}
Deklarasi
    mid : integer

Algoritma:
    if i > j then        { ukuran larik sudah 0}
        idx ← –1         { K tidak ditemukan }
    else
        mid ← (i + j)/2
        if A(mid) = K then       { K ditemukan }
            idx ← mid              { indeks elemen larik yang bernilai = K }
        else
            if  A(mid) > K  then
                binsearch(A, i, mid – 1, K, idx)      { cari di upalarik kiri, di dalam larik A[i..mid]}
            else
                binsearch(A, mid + 1, j, K, idx)      { cari di upalarik kanan, di dalam larik A[mid+1..j}
            endif
        endif
    endif
```

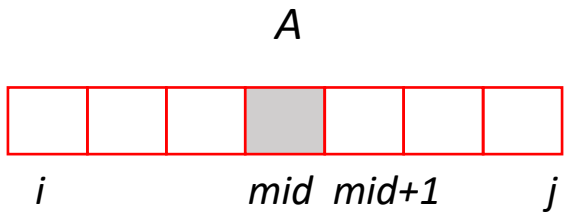




Algoritma *Binary Search* (Kasus 2: Larik sudah terurut menurun)

```
procedure binsearch(input A : LarikInteger,  i, j : integer; K : integer; output idx : integer)
{ Mencari elemen bernilai K di dalam larik A[i..j].
  Masukan: larik A sudah terurut menurun, K sudah terdefinisi nilainya
  Luaran: indek lariks sedemikian sehingga A[idx] = K
}
Deklarasi
    mid : integer

Algoritma:
    if i > j then        { ukuran larik sudah 0}
        idx ← –1         { K tidak ditemukan }
    else
        mid ← (i + j)/2
        if A(mid) = K then       { K ditemukan }
            idx ← mid              { indeks elemen larik yang bernilai = K }
        else
            if  A(mid) < K  then
                binsearch(A, i, mid – 1, K, idx)      { cari di upalarik kiri, di dalam larik A[i..mid]}
            else
                binsearch(A, mid + 1, j, K, idx)      { cari di upalarik kanan, di dalam larik A[mid+1..j}
            endif
        endif
    endif
```





Contoh 3: Misalkan diberikan larik *A* dengan delapan buah elemen yang sudah terurut menurun seperti di bawah ini:

81	76	21	18	16	13	10	7
i =1	2	3	4	5	6	7	8=*j*

Nilai K yang dicari adalah K = 16.

Langkah-Langkah pencarian:

Langkah 1:

$i = 1$ dan $j = 8$

Indeks elemen tengah $mid = (1 + 8)$ div $2 = 4$ (elemen yang diarsir)

81	76	21	18	16	13	10	7
1	2	3	4	5	6	7	8

kiri kanan

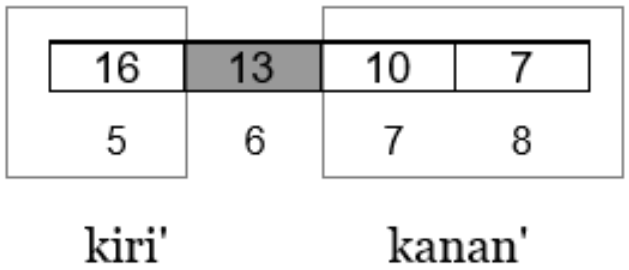
$A[4] \neq 16$

$A[4] < 16$? Tidak, cari pada upalarik kanan

Langkah 2:

$i = 5$ dan $j = 8$

Indeks elemen tengah $mid = (5 + 8)$ div $2 = 6$ (elemen yang diarsir)



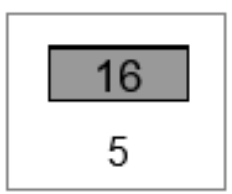
$A[6] \neq 16$

$A[4] < 16$? Ya, cari pada upalarik kiri

Langkah 3:

$i = 5$ dan $j = 5$

Indeks elemen tengah $mid = (5 + 5)$ div $2 = 5$ (elemen yang diarsir)



$A[5] = 16$

Pencarian ditemukan pada idx = 5

- Jumlah operasi perbandingan elemen-elemen larik:

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

- Relasi rekursens tersebut diselesaikan secara iteratif sbb (atau pakai Teorema Master):

T(n) = 1 + T(n/2)

= 1 + (1 + T(n/4)) = 2 + T(n/4)

= 2 + (1 + T(n/8)) = 3 + T(n/8)

= …

= k + T($n/2^k$)

Asumsi: ukuran larik adalah perpangkatan dua, atau n = 2^k → k = ^{2}log n

T(n) = ^{2}log n + T(1) = ^{2}log n + (1 + T(0)) = 1 + ^{2}log n = O(^{2}log n)

5. Mencari koin palsu

Diberikan *n* buah koin yang identik, satu diantaranya palsu. Asumsikan koin yang palsu mempunyai berat yang lebih ringan daripada koin asli. Untuk mencari yang palsu, disediakan sebuah timbangan yang teliti. Carilah koin yang palsu dengan cara penimbangan.

Algoritma *decrease and conquer*:

1. Bagi himpunan koin menjadi dua upa-himpunan (*subset*), masing-masing $\lfloor n/2 \rfloor$ koin. Jika *n* ganjil, maka satu buah koin tidak dimasukkan ke dalam kedua upa-himpunan.
2. Timbang kedua upa-himpunan dengan neraca.
3. Jika beratnya sama, berarti satu koin yang tersisa adalah palsu.
4. Jika beratnya tidak sama, maka ulangi proses untuk upa-himpunan yang beratnya lebih ringan (salah satu koin di dalamnya palsu).

- Ukuran persoalan selalu berkurang dengan faktor setengah dari ukuran semula. Hanya setengah bagian yang diproses, setengah bagian yang lain tidak diproses.

- Jumlah penimbangan yang dilakukan adalah:

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

- Penyelesaian relasi rekurens T(n) mirip seperti *binary search*, atau menggunakan Teorema Master:

$T(n) = 1 + T(\lfloor n/2 \rfloor) = \ldots. = O(^2 \log n)$

6. Persoalan Perkalian Petani Rusia

- Mengalikan dua buah bilangan bulat positif *m* dan *n*

$$n \cdot m = \begin{cases} \frac{n}{2} \cdot 2m & , n \text{ genap} \\ \frac{n-1}{2} \cdot 2m + m & , n \text{ ganjil} \end{cases}$$

- Pada setiap pemanggilan rekursif, nilai *n* berkurang dengan faktor ½ menjadi n/2.

- Kasus trivial untk *n* = 1, maka 1 · *m* = *m*

Contoh: menghitung 50 · 65 dengan metode perkalian petani Rusia

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130 + 1040) = 3250

(a)

n	m	
50	65	
25	130	130
12	260	
6	520	
3	1040	1040
1	2080	2080
		3250

(b)

FIGURE 4.11 Computing 50 · 65 by the Russian peasant method.

Note that all the extra addends shown in parentheses in Figure 4.11a are in the rows that have odd values in the first column. Therefore, we can find the product by simply adding all the elements in the *m* column that have an odd number in the *n* column (Figure 4.11b).

Sumber: Levitin

BERSAMBUNG