7 - Decrease and Conquer

[KOMS120403]

Desain dan Analisis Algoritma (2022/2023)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 8 (March 2023)

Daftar isi

Prinsip dasar Decrease-and-Conquer (1)

- Algoritma ini mirip dengan DnC, namun algoritma ini mempartisi masalah menjadi beberapa submasalah dengan ukuran yang lebih kecil, kita menggunakan beberapa teknik untuk mereduksi masalah kita menjadi satu masalah yang lebih kecil dari aslinya.
- Beberapa penulis menganggap bahwa nama "Divide-and-Conquer" hanya digunakan hanya ketika setiap masalah dapat menghasilkan **dua atau lebih submasalah**. Nama Reduce-and-Conquer telah diusulkan sebagai gantinya untuk kelas subproblem tunggal. Dalam literatur lama, keduanya disebut sebagai " Divide-and-Conquer".

Prinsip dasar Decrease-and-Conquer (2)

Pendekatan ini didasarkan pada mengeksploitasi hubungan antara solusi untuk contoh masalah tertentu dan solusi untuk contoh yang lebih kecil.

Implementasi:

- Top-down approach: Itu selalu mengarah pada implementasi masalah secara rekursif.
- Bottom-up approach: Ini biasanya diimplementasikan dengan cara iteratif, dimulai dengan solusi dari masalah terkecil.

Prinsip dasar Decrease-and-Conquer (3)

Algoritma Powering

Pendekatan 1.

$$X^n = \begin{cases} 1 & \text{for } n = 0 \\ X^{n-1} \cdot X & \text{for } n > 0 \end{cases}$$

Pendekatan 2.

$$X^n = \underbrace{X \cdot X \cdot X \cdot \cdots \cdot X}_{n \text{ times}}$$

Pendekatan manakah yang termasuk Top-down/Bottom-up?

Varian Decrease-and-Conquer

Tiga variasi utama metode Decrease-and-Conquer

1. Decrease by a constant: ukuran instance dikurangi dengan konstanta yang sama pada setiap iterasi algoritma. Biasanya, konstanta ini sama dengan satu.
2. Decrease by a constant factor: mengurangi contoh masalah dengan faktor konstan yang sama pada setiap iterasi algoritma. Pada kebanyakan penerapan, faktor konstanta-nya adalah 2.
3. Decrease by a variable size: pola pengurangan ukuran bervariasi dari satu iterasi algoritma ke iterasi lainnya.

1. Decrease by a constant (1)

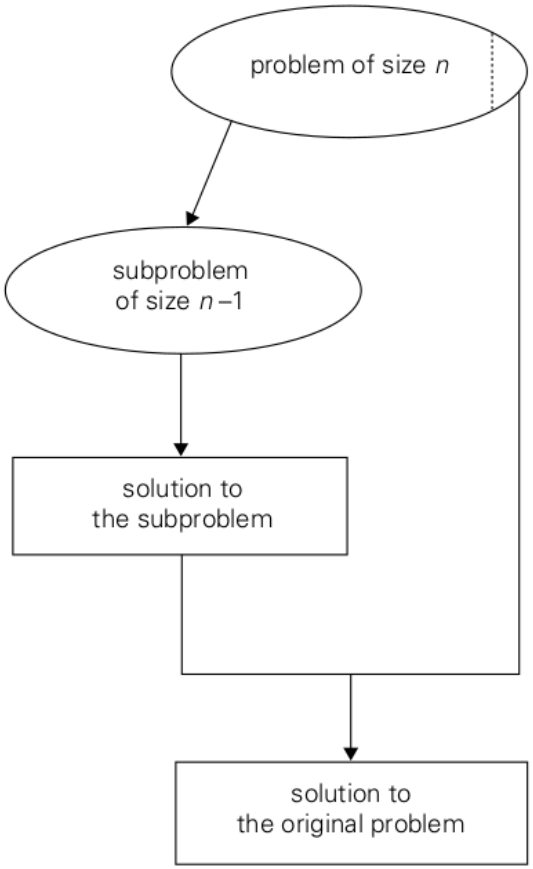


Figure: Decrease-(by one)-and-conquer technique.

1. Decrease by a constant (2)

Contoh. **Powering:** X^n

$$X^n = \begin{cases} 1 & \text{for } n = 0 \\ X^{n-1} \cdot X & \text{for } n > 0 \end{cases}$$

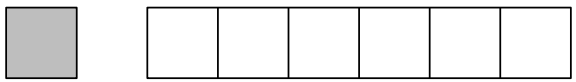
Dalam hal ini, X^n dihitung dengan terlebih dahulu mengurangi eksponen dengan 1 (yaitu dengan menghitung X^{n-1}).

1. Decrease by a constant (3)

Contoh lain:

- Selection sort
  Ini adalah algoritma *hard split/easy join* dengan membagi array menjadi dua sub-array: sub-array pertama hanya berisi *one element*, dan sub-array lainnya berisi $n - 1$ elemen.

- Insertion sort
  Ini adalah algoritma *easy split/hard join* dengan membagi array menjadi dua sub-array: sub-array pertama hanya berisi *one element*, dan sub-array lainnya berisi $n - 1$ elemen .



2. Decrease by a constant factor (1)

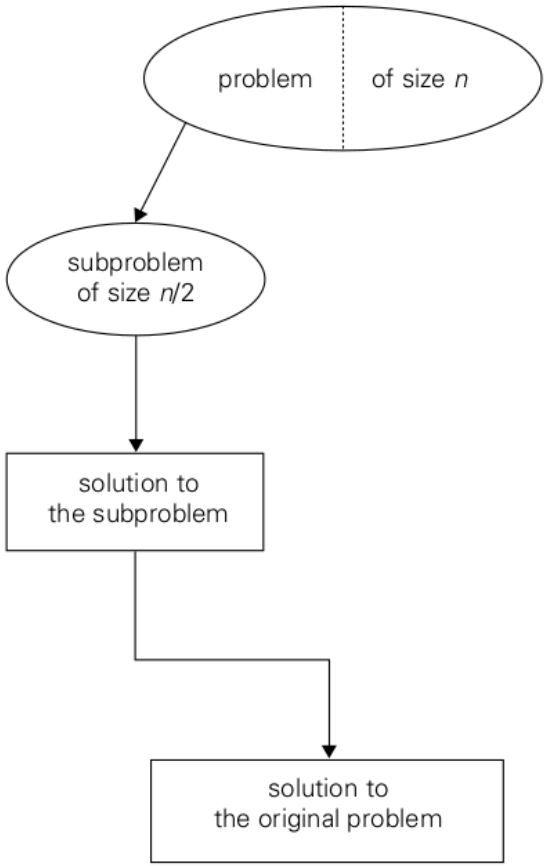


Figure: Teknik “decrease-(by half)-and-conquer”.

2. Decrease by a constant factor (2)

Contoh 1: **Powering:** X^n

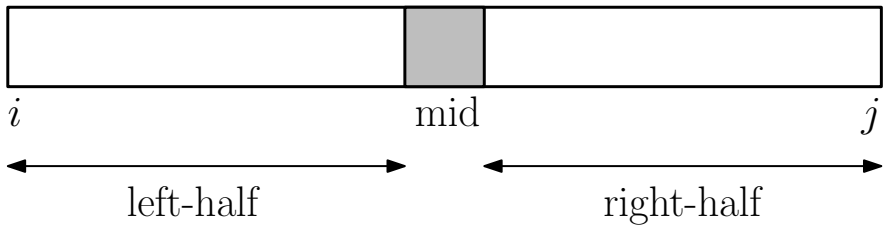
$$X^n = \begin{cases} 1 & \text{for } n = 0 \\ X^{n/2} \cdot X^{n/2} & \text{for } n > 0 \end{cases}$$

Dalam hal ini, X^n dihitung dengan terlebih dahulu mengurangi eksponen menjadi setengahnya (yaitu menghitung $X^{n/2}$).

2. Decrease by a constant factor (3)

Contoh 2: Binary search (*telah dibahas pada Week 05*)

Diberikan array A yang terurut (dalam urutan menaik), dan nilai X. Kita ingin memeriksa apakah X termuat dalam A, dan menemukan posisi dari X di A.



- Jika nilai tengah array $\neq X$, maka kita melakukan pencarian biner X hanya di bagian kiri atau bagian kanan.
- Jadi ukuran instance berkurang setengahnya.

2. Decrease by a constant factor (4)

Contoh 3: Fake-Coin Problem

Problem

Di antara n koin yang tampak identik, satu koin palsu. Dengan menggunakan timbangan setimbang, kita dapat membandingkan dua set koin. Artinya, dengan memiringkan ke kiri, ke kanan, atau seimbang, timbangan akan menunjukkan apakah set tersebut memiliki berat yang sama atau set mana yang lebih berat dari yang lain tetapi tidak seberapa banyak.

Tugas: *Rancanglah algoritma yang efisien untuk mendeteksi koin palsu.*

Versi permasalahan yang lebih sederhana: *asumsikan bahwa koin palsu diketahui lebih ringan daripada yang asli*

2. Decrease by a constant factor (5)

Strategi:

1. Bagi koin n menjadi dua tumpukan masing-masing $\lfloor n/2 \rfloor$ koin.
2. Jika tumpukan beratnya sama, koin yang disisihkan pasti palsu.
3. Jika tidak, kita dapat melanjutkan dengan cara yang sama dengan tumpukan yang lebih ringan, yang harus menjadi koin palsu.

Kompleksitas waktu:

$W(n)$: banyaknya perbandingan yang dibutuhkan oleh algoritma ini dalam kasus terburuk.

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

$W(n) = \mathcal{O}(\log n)$.

3. Decrease by a variable size (1)

Pola pengurangan ukuran bervariasi dari satu iterasi algoritma ke iterasi lainnya.

Contoh: Algoritma Euclid untuk menghitung pembagi persekutuan terbesar. Algoritma didasarkan pada properti:

$$fpb(m, n) = fpb(n,\ m \bmod n)$$

Meskipun nilai argumen kedua selalu lebih kecil di sisi kanan daripada di sisi kiri, tidak terjadi pengurangan baik dengan konstanta maupun dengan faktor konstanta.

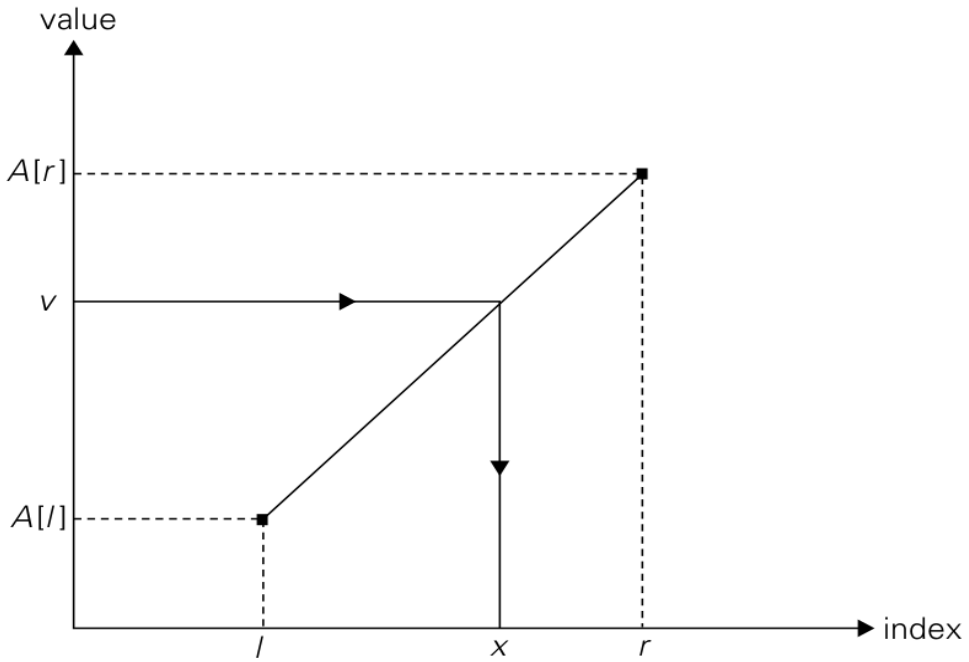
3. Decrease by a variable size (2): Interpolation search

Interpolation search

Masalah: Diberikan array A diurutkan dengan cara menaik, dan kunci pencarian v. Periksa apakah v ada di A, dan temukan posisi v di A.

- Pencarian interpolasi memperhitungkan nilai kunci pencarian untuk menemukan elemen array yang akan dibandingkan dengan kunci pencarian.
- Algoritma meniru cara kita mencari nama di buku telepon.
  - Misalnya jika kita sedang mencari seseorang bernama "Brown", kita membuka buku tidak di tengah tetapi sangat dekat dengan halaman depan. Demikian pula halnya jika kita mencari nama "Smith", kita cenderung membuka buku dekat dengan halaman akhir.
- Algoritma mengasumsikan bahwa nilai array meningkat secara linear, yaitu, sepanjang garis lurus melalui titik $(\ell, A[\ell])$ dan $(r, A[r])$. (Keakuratan asumsi ini dapat mempengaruhi efisiensi algoritma, tetapi tidak mempengaruhi kebenarannya.)

Interpolation search (*lanjutan*)



$$\frac{v - A[\ell]}{A[r] - A[\ell]} = \frac{x - \ell}{r - \ell}$$

$$x = \ell + r - \ell \cdot \frac{v - A[\ell]}{A[r] - A[\ell]}$$

ℓ: the start index of A
r: the end index of A
v: the search key
x: the index of v

Kita tidak menggunakan konstanta $\frac{1}{2}$ seperti dalam Binary Search, tetapi konstanta lain yang lebih akurat "*c*", yang dapat mengarahkan kita lebih dekat ke elemen yang dicari.

Interpolation search (*lanjutan*)

Ingat lagi algoritma Binary Search

Algorithm 1 Binary search algorithm

```
1:  procedure BinSearch(A, i, j, KEY)
2:      if i > j then
3:          return −1                          ▷ Base case is reached but KEY is not found
4:      end if
5:      m = ⌊(i+j)/2⌋                          ▷ Choose the pivot
6:      if KEY = A[m] then
7:          return m                           ▷ KEY is found at index m
8:      else
9:          if KEY < A[m] then                 ▷ The KEY is located on the Left sub-array
10:             return BinSearch(A, i, m − 1, KEY)   ▷ Rec-call left part
11:         else
12:             return BinSearch(A, m + 1, j, KEY)   ▷ Rec-call right part
13:         end if
14:     end if
15: end procedure
```

Interpolation search (*cont.*)

Algoritma Interpolation Search mirip dengan Binary Search, yakni dengan mensubstitusi

$$\text{mid} \leftarrow (i + j) \ \textbf{div} \ 2$$

dengan

$$\text{mid} \leftarrow i + (j - i) * \left(\frac{v - A[i]}{A[j] - A[i]} \right)$$

(*Pada gambar sebelumnya*, $i = \ell, \ \ j = r$)

Interpolation search (*lanjutan*)

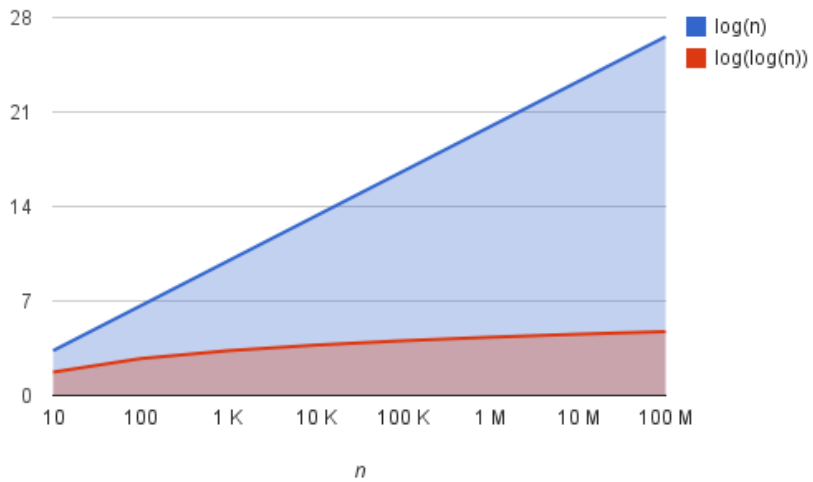
Algorithm 2 Interpolation search algorithm

```
1:  procedure InterpSearch(A, i, j, v)
2:      if i > j then
3:          return −1                          ▷ Base case is reached but v is not found
4:      end if
5:      m = ⌊i + (j − i) * (v − A[i]) / (A[j] − A[i])⌋      ▷ Choose the pivot
6:      if v = A[m] then
7:          return m                           ▷ v is found at index m
8:      else
9:          if v < A[m] then                   ▷ v is located on the left sub-array
10:             return InterpSearch(A, i, m − 1, v)      ▷ Rec-call left part
11:         else
12:             return InterpSearch(A, m + 1, j, v)      ▷ Rec-call right part
13:         end if
14:     end if
15: end procedure
```

Interpolation search (*cont.*)

Kompleksitas waktu InterpSearch

- Best-case: $\mathcal{O}(1)$
- Worst-case: $\mathcal{O}(n)$, untuk sebarang distribusi data.
- Average-case: $\mathcal{O}(\log \log n)$, jika data pada array *berdistribusi seragam (uniformly distributed)*.



Jika $n = 10^9$, $\log(\log(n)) \approx 5$ (dihitung dalam basis 2), $\log(n) \approx 30$.

3. Decrease by a variable size (3): Penghitungan median & Selection Problem

Selection problem adalah masalah menemukan elemen terkecil ke-k dalam daftar angka n. Angka ini disebut ordo statistika ke-k.

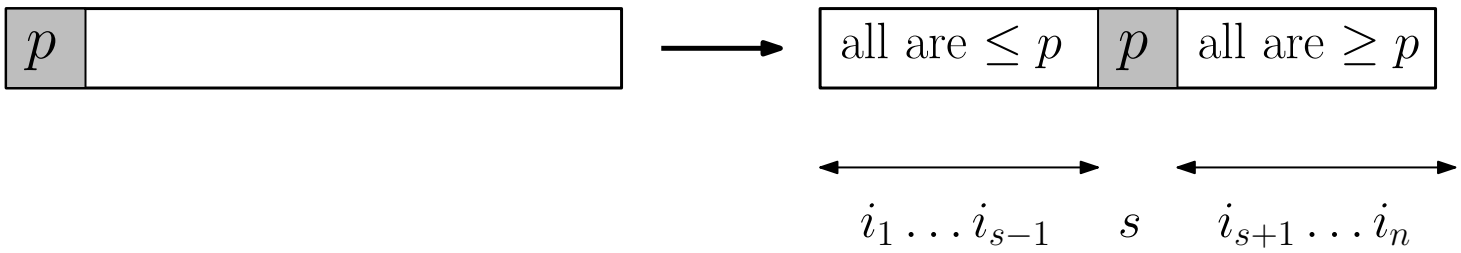
- Jika $k = 1$, maka ini ekuivalen dengan mencari elemen (*minimum*)
- Jika $k = n$, maka ini ekuivalen dengan mencari elemen (*maksimum*)
- Jika $k = \lceil \frac{n}{2} \rceil$, maka ini ekuivalen dengan mencari *median*

Jelas, kita dapat menemukan elemen terkecil ke-k dalam daftar *dengan menyortir daftar terlebih dahulu dan kemudian memilih elemen ke-k dari output algoritma sorting*.

Tapi, bagaimana jika *kita tidak diperbolehkan mengurutkan array?*

Penghitungan median & Selection Problem (*cont.*)

Gunakan ide yang mirip dengan "Partitioning" yang digunakan di Quick Sort. Tapi kita tidak selalu mempartisi array pencarian di tengah.



- Jika $s = \lceil n/2 \rceil$, maka pivot p adalah median
- Jika $s > \lceil n/2 \rceil$, maka mediannya ada di sub-array kiri
- Jika $s < \lceil n/2 \rceil$, maka mediannya ada di sub-array kanan

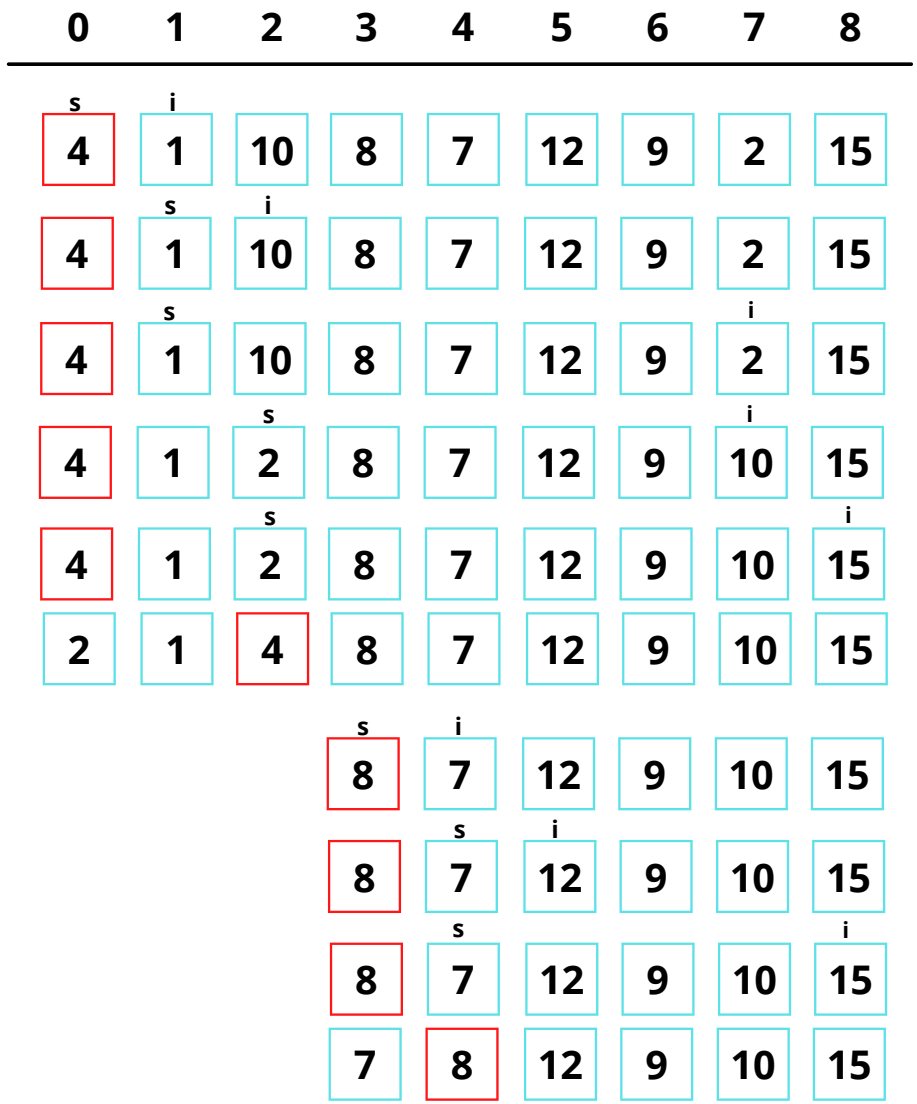
Penghitungan median & Selection Problem (*cont.*)

Example of implementation of QUICKSELECT

Apply the *partition-based* algorithm to find the median

Here k = ⌈9/2⌉ = 5
So we want to find the element in the 5th position

x **pivot**



Penghitungan median & Selection Problem (*cont.*)

Algorithm 3 Lomuto partition

```
1:  procedure LOMUTO(A, ℓ, r)
2:      input: a sub-array A[ℓ..r] of array A[0..n − 1]
3:      output: partition of A[ℓ..r] and the new position of the pivot
4:      p ← A[ℓ]
5:      s ← ℓ
6:      for i ← ℓ + 1 to r do
7:          if A[i] < p then
8:              s ← s + 1
9:              swap(A[s], A[i])
10:         end if
11:     end for
12:     swap(A[ℓ], A[s])
13:     return s
14: end procedure
```

Penghitungan median & Selection Problem (*cont.*)

Algorithm 4 Quick Selection

```
1: procedure QUICKSELECT(A, ℓ, r, k)
2:     input: a sub-array A[ℓ..r] of ordorable array A[0..n − 1], and integer k with
   1 ≤ k ≤ r − ℓ + 1
3:     output: the value of the k-th smallest element in A[ℓ..r]
4:     s ← LOMUTO(A, ℓ, r)
5:     if s = k − 1 then
6:         return A[s]
7:     else
8:         if s > ℓ + k − 1 then
9:             QUICKSELECT(A, ℓ, s − 1, k)
10:        else
11:            QUICKSELECT(A, s + 1, r, k − 1 − s)
12:        end if
13:    end if
14: end procedure
```

Penghitungan median & Selection Problem (*cont.*)

Kompleksitas waktu Quickselect

Best-case

- Partisi array ukuran n oleh LOMUTO selalu membutuhkan $n-1$ perbandingan kunci.
- Jika menghasilkan pemisahan yang menghasilkan solusi tanpa panggilan rekursif, maka:

$$TC_{\text{best}}(n) = n - 1 \in \Theta(n)$$

Worst-case

Tapi, algoritma dapat menghasilkan yang sangat tidak seimbang partisi dari array yang diberikan, dengan satu bagian kosong dan yang lainnya berisi elemen $n-1$.

Dalam hal ini:

$$TC_{\text{worst}}(n) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2} \in \Theta(n^2)$$

end of slide...