3.2 - Algoritma *Brute Force* (bagian 2)

[KOMS120403]

Desain dan Analisis Algoritma (2022/2023)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 4 (February 2022)

Daftar isi

- Q and A (oral quiz)
- Selection sort
- Bubble sort
- Insertion sort
- Pembuktian kebenaran algoritma menggunakan invariansi loop (*loop invariant*)

Tugas:

Jawablah QnA berikut untuk mengecek pemahaman Anda!

Q n A

1. Dapatkah Anda jelaskan kelemahan algoritma *brute force*?

Q n A

1. Dapatkah Anda jelaskan kelemahan algoritma *brute force*?
2. Dengan kelemahan tersebut, lalu mengapa algoritma *brute force* masih digunakan?

Q n A

1. Dapatkah Anda jelaskan kelemahan algoritma *brute force*?
2. Dengan kelemahan tersebut, lalu mengapa algoritma *brute force* masih digunakan?
3. Jelaskan deskripsi masalah penugasan (*assignment problem*)!

Q n A

1. Dapatkah Anda jelaskan kelemahan algoritma *brute force*?
2. Dengan kelemahan tersebut, lalu mengapa algoritma *brute force* masih digunakan?
3. Jelaskan deskripsi masalah penugasan (*assignment problem*)!
4. Jelaskan deskripsi masalah partisi (*partition problem*)!

Q n A

1. Dapatkah Anda jelaskan kelemahan algoritma *brute force*?
2. Dengan kelemahan tersebut, lalu mengapa algoritma *brute force* masih digunakan?
3. Jelaskan deskripsi masalah penugasan (*assignment problem*)!
4. Jelaskan deskripsi masalah partisi (*partition problem*)!
5. Jelaskan deskripsi masalah *magic square*!

Q n A

1. Dapatkah Anda jelaskan kelemahan algoritma *brute force*?
2. Dengan kelemahan tersebut, lalu mengapa algoritma *brute force* masih digunakan?
3. Jelaskan deskripsi masalah penugasan (*assignment problem*)!
4. Jelaskan deskripsi masalah partisi (*partition problem*)!
5. Jelaskan deskripsi masalah *magic square*!
6. Bagaimana teknik heuristik dapat membantu dalam meningkatkan efisiensi teknik brute-force untuk memecahkan masalah *magic square*?

Bagian 1. Selection Sort

Selection Sort (1): Algoritma

Permasalahan: Diberikan array dengan n elemen yang dapat diurutkan. Urutkan array dan tampilkan array yang diurutkan dalam urutan yang *tak-turun (non-descending)*.

1. Temukan elemen terbesar x dalam interval $[0..n-1]$
2. Tukar x dengan elemen ke-$(n-1)$
3. Kurangi n dengan 1 dan ulangi Langkah 1

Selection Sort (2): Contoh

29	10	14	37	13
29	10	14	13	37
13	10	14	29	37
13	10	14	29	37
10	13	14	29	37

37 is the largest, swap it with the last element, i.e. **13**.

How to find the largest?

X	Unsorted item
X	Largest item for the current iteration
X	Sorted item

Selection Sort (3): Pseudocode

Algorithm 1 Selection sort

```
1:  procedure SelectionSort(A[0..n − 1]: ordorable array)
2:      for i = n − 1 downto 1 do
3:          maxIdx = i                          ▷ We'll find the correct elmt for position i
4:          for j = 0 to i − 1 do
5:              if a[j] >= a[maxIdx] then
6:                  maxIdx = j                  ▷ Iteratively choose a larger elmt for position i
7:              end if
8:          end for
9:          swap(a[i], a[maxIdx])               ▷ the correct elmt at position i is found at index maxIdx
10:     end for
11: end procedure
```

Selection Sort (4): Algoritma versi kedua (*by minimum*)

Coba Anda bandingkan algoritma sebelumnya dengan algoritma Selection Sort berikut. Analisis perbedaannya!

ALGORITHM *SelectionSort*$(A[0..n-1])$

//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

Figure: Algoritma Selection sort pada buku Anany Levitin

Selection Sort (5): Analisis kompleksitas

Number of executions

Algorithm 1 Selection sort

```
1:  procedure SelectionSort(A[1..n])
2:      for i = n − 1 downto 1 do                    ← n-1
3:          maxIdx = i
4:          for j = 0 to i − 1 do                    ← n-1
5:              if a[j] >= a[maxIdx] then            ← (n-1) + (n-2) + ... + 1
6:                  maxIdx = j                           = n(n-1) / 2
7:              end if
8:          end for
9:          swap(a[i], a[maxIdx])                    ← n-1
10:     end for
11: end procedure
```

Kompleksitas waktu: $\mathcal{O}(n^2)$

Bagian 2. Bubble Sort

Bubble Sort (1): Algoritma

Idea: Diberikan sebuah array dengan n elemen

1. Bandingkan sepasang elemen yang berdekatan
2. Tukar jika elemen rusak
3. Ulangi sampai akhir array
   - Elemen terbesar akan berada di posisi terakhir
4. Kurangi n dengan 1 dan lanjutkan ke Langkah 1

Bubble Sort (2): Contoh

(a) Pass 1

29	10	14	37	13
10	29	14	37	13
10	14	29	37	13
10	14	29	37	13
10	14	29	13	37

At the end of **Pass 1**, the largest item **37** is at the last position

(b) Pass 2

10	14	29	13	37
10	14	29	13	37
10	14	29	13	37
10	14	13	29	37

At the end of **Pass 2**, the second-largest item **29** is at the second last position

Bubble Sort (3)

Apakah algoritma berikut juga mendefinisikan algoritma bubble-sort? Jelaskan!

ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

source: book of Levitin

Bubble Sort (4): Pseudocode

Algorithm 2 Bubble sort

```
1: procedure BubbleSort(A[0..n − 1])
2:     for i = n − 1 downto 1 do
3:         for j = 1 to i do
4:             if a[j − 1] > a[j] then          ▷ Compare adjacent pairs of elements
5:                 swap(a[j], a[j − 1])         ▷ Swap if the elements are not in correct order
6:             end if
7:         end for
8:     end for
9: end procedure
```

Bubble Sort (5): Analisis kompleksitas

- Satu iterasi dari loop dalam ("if condition" dan "swap") membutuhkan waktu yang dibatasi oleh konstanta c.
- Untuk dua loop bersarang:
  - Outer loop: memuat n iterasi
  - Inner loop:
    - ketika $i = 0$, terdapat $(n-1)$ iterasi
    - ketika $i = 1$, terdapat $(n-2)$ iterasi
    - $\cdots$
    - ketika $i = n-1$, terdapat 0 iterasi
- Jumlah total iterasi: $0 + 1 + \cdots + (n-1) = \frac{n(n-1)}{2}$.
- Total waktu eksekusi: $c \cdot \frac{n(n-1)}{2} = \mathcal{O}(n^2)$

Bubble Sort (6): Algoritma (*version 2*)

Algorithm 3 Bubble sort version 2

```
1:  procedure BubbleSort2(A[1..n])
2:      for i = n − 1 downto 1 do
3:          sorted = True
4:          for j = 1 to i do
5:              if a[j − 1] > a[j] then
6:                  swap(a[j], a[j − 1])
7:                  sorted = False
8:              end if
9:          end for
10:         if (sorted) then
11:             return
12:         end if
13:     end for
14: end procedure
```

Bubble Sort (7): Analisis kompleksitas (*version 2*)

- Kasus terburuk (*worst-case*)
  - Terjadi jika input sudah dalam urutan menurun (*descending*)
  - Kompleksitas waktu: $\mathcal{O}(n^2)$
- Kasus terbaik (*best-case*)
  - Terjadi jika input sudah dalam urutan naik (*ascending*)
  - Algoritma kembali setelah iterasi luar tunggal
  - Kompleksitas waktu: $\mathcal{O}(n)$

Bagian 3. Insertion Sort

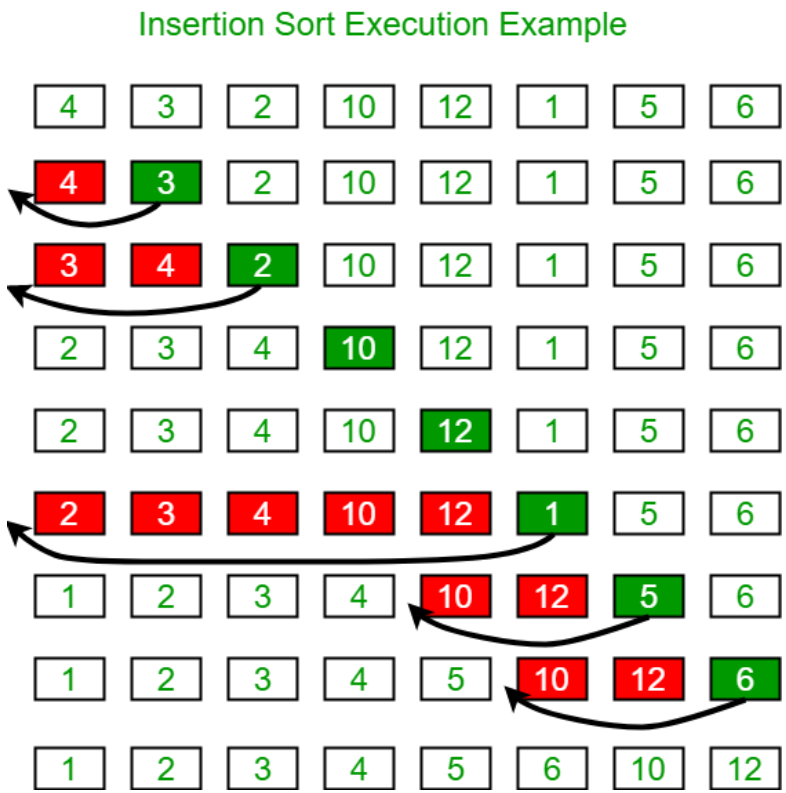
Insertion sort (1): Algoritma

Untuk mengurutkan array $A[0..n-1]$ dengan ukuran n dalam urutan menaik:

1. Ulangi langkah berikut mulai dari $A[0]$ sampai dengan $A[n-1]$ sepanjang array.
2. Bandingkan elemen saat ini (dinamai sebagai 'kunci') dengan pendahulunya (elemen sebelumnya).
3. Jika elemen kunci kurang dari pendahulunya, bandingkan dengan elemen pendahulu sebelumnya. Pindahkan elemen yang lebih besar satu posisi ke atas untuk memberi ruang bagi elemen yang ditukar.

Insertion sort (2): Contoh

Berikut adalah ilustrasi contoh penerapan algoritma Selection Sort. Pelajari dengan seksama, dan pahami setiap langkahnya.



source: `https://www.geeksforgeeks.org/insertion-sort`

Insertion sort (3): Pseudocode

Algorithm 4 Insertion sort

```
1:  procedure InsertionSort(A[0..n − 1]: ordorable array)
2:      i ← 1
3:      while i < n do                ▷ We'll find the correct position for A[1], A[2], . . . , A[n − 1]
4:          key ← A[i]                ▷ 'key' is the current element that will be inserted
5:          j ← i − 1                 ▷ Using index j, we'll find the correct position for A[i]
6:          while j ≥ 0 and A[j] > key do
7:              A[j + 1] ← A[j]       ▷ A[j] is shifted one position to the right (to index j + 1)
8:              j ← j − 1             ▷ Decrement j (until the correct position is found)
9:          end while
10:         A[j + 1] ← key            ▷ Once found, 'key' is inserted in the correct position (at idx j + 1)
11:         i ← i + 1                 ▷ Increment i to work on the next 'key'
12:     end while
13: end procedure
```

Insertion sort (4): Kompleksitas waktu

Kompleksitas: $\mathcal{O}(n^2)$ (karena ada dua loop *while* bersarang, masing-masing dengan kompleksitas $\mathcal{O}(n)$)

Algorithm 5 Insertion sort

```
1:  procedure InsertionSort(A[0..n − 1]: ordorable array)
2:      i ← 1
3:      while i < n do                          ▷ 'while loop' involves (n − 1) iterations
4:          key ← A[i]
5:          j ← i − 1
6:          while j ≥ 0 and A[j] > key do       ▷ In the worst case: ∃ (n − 1) iterations
7:              A[j + 1] ← A[j]
8:              j ← j − 1
9:          end while
10:         A[j + 1] ← key
11:         i ← i + 1
12:     end while
13: end procedure
```

Bagian 4. Pembuktian kebenaran dengan invariansi loop (*loop invariant*)

Membuktikan kebenaran melalui loop invarian

- Ingat kembali definisi kebenaran algoritma:

  memberikan return solusi yang benar untuk setiap instance masalah yang valid

- Cara standar untuk membuktikan kebenaran adalah dengan karakteristik invariansi loop

Karakteristik invariansi loop:

- Ini adalah **sifat kunci** dari data yang dimanipulasi oleh loop utama dari sebuah algoritma
  - Karakteristik harus didefinisikan dan dapat membantu kita memahami mengapa algoritma itu benar
  - Kita harus menunjukkan bahwa karakteristik tersebut berlaku pada kasus awal, dipertahankan pada setiap iterasi, dan ketika perulangan berakhir, dihasilkan kebenaran
- Menentukan karakteristik ini secara umum bisa jadi sulit. Namun dalam banyak algoritma, karakteristik invariansi loop seringkali menjadi kunci yang menentukan fitur dari algoritma.

Keterkaitan invariansi loop dengan kebenaran algoritma (1)

Tiga karakteristik utama dari invariansi loop

1. Inisialisasi: Invariansi loop harus benar sebelum iterasi pertama dari loop.

2. Maintenance: Jika sifat berlaku sebelum iterasi dari loop, maka sifat tersebut harus tetap berlaku setelah iterasi selesai.

3. Termination: Saat loop berakhir, invarian menyediakan sifat berguna yang membantu menunjukkan bahwa algoritma sudah benar.

Keterkaitan invariansi loop dengan kebenaran algoritma (2)

- Untuk membuktikan kebenarannya, kita harus membuktikan hal di atas tentang sifat invarian loop.
- *Karakteristik (1) dan (2) mirip dengan induksi.* Jika invariansi loop berlaku, maka invariansi loop benar **sebelum** eksekusi dari setiap iterasi loop.
- *Karakteristik (3) membedakan invariansi loop dengan induksi* dan merupakan bagian yang paling penting.
  - Kita tidak menunjukkan bahwa invariansi loop harus berlaku pada *ad infinitum* (i.e., untuk $n \to \infty$);
  - melainkan menghasilkan jawaban yang benar setelah sejumlah langkah terbatas (untuk nilai $n \geq n_0$ pada suatu batas $n_0 \in \mathbb{Z}^+$).

Lanjutan bagian 4. Contoh pembuktian kebenaran dengan invariansi loop

1. Kebenaran selection sort dengan invariansi loop (1)

Contoh invariansi loop (*in* SELECTIONSORT)

```
1:  procedure SELECTIONSORT(A[0..n − 1]: ordorable array)
2:      for i = n − 1 downto 1 do
3:          maxIdx = i
4:          for j = 0 to i − 1 do
5:              if a[j] >= a[maxIdx] then
6:                  maxIdx = j
7:              end if
8:          end for
9:          swap(a[i], a[maxIdx])
10:     end for
11: end procedure
```

Invariansi loop. Di awal setiap iterasi dari loop terluar:

- Sublist $A[0..i]$ berisi elemen $i + 1$ yang tersisa. Tujuan kita adalah untuk menempatkan elemen yang benar pada posisi i.
- Sublist $A[i + 1..n - 1]$ terdiri dari $(n - 1 - i)$ elemen terbesar A dalam urutan yang benar (terurut).

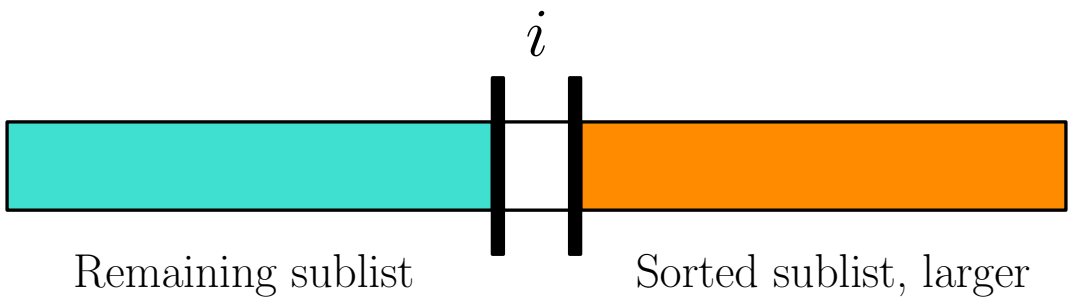
1. Kebenaran algoritma Selection Sort: Invariansi loop (2)

Pada dasarnya, invariansi loop menyatakan bahwa pada setiap langkah, array dapat dibagi menjadi dua bagian:

- Bagian *dari i ke kiri* adalah sublist yang masih dikerjakan oleh algoritma
- Bagian *di sebelah kanan i* adalah sublist yang diurutkan dari elemen di A

Loop invariant. At the start of each iteration of the outermost loop, the sublist $A[i+1..n-1]$ consists of the $n-1-i$ largest elements of A in the correct order. The sublist $A[0..i]$ contains the remaining $i+1$ elements.

Our goal is to put the correct element at position i.



1. Kebenaran selection sort dengan invariansi loop (3)

Apa yang kita buktikan?

- "Selection" berarti mengambil daftar elemen yang tidak diurutkan dan mengurutkannya berdasarkan nilai.
- Kita menunjukkan bahwa algoritma SelectionSort akan, pada setiap langkah, mempertahankan sifat berikut:

  ielemen pada sebelah kanan indeks utama berada dalam urutan terurut dan tidak kurang dari elemen manapun yang terletak di sebelah kiri indeks utama.

- Jika sifat ini berlaku saat inisialisasi, dipertahankan pada setiap langkah, dan diakhiri dengan jawaban yang benar, maka dapat disimpulkan bahwa SelectionSort merupakan algoritma pengurutan yang **benar**, sesuai dengan prinsip invariansi loop.

2. Kebenaran BubbleSort and InsertionSort?

Latihan:

- Apakah bukti "invariansi loop" masih berfungsi untuk algoritma seleksi-urutan yang diberikan dalam buku Anany Levitin?
- Coba Anda buktikan kebenaran BubbleSort dan Insertion Sort! dengan menggunakan teknik invariansi loop!
- Apakah metode "invariansi loop" dapat diaplikasikan dalam hal ini? Jika iya, jelaskan invariansi loop untuk setiap algoritma!

Lampiran 1: Pseudocode Selection Sort

Algorithm 6 Selection sort

```
1:  procedure SelectionSort
2:      Input: array A[0..n − 1]: yang dapat diurutkan
3:      Output: array yang terurut ascending
4:      for i ← 0 to n − 2 do
5:          min ← i
6:          for j ← i + 1 to n − 1 do
7:              if A[j] < A[min] then
8:                  min ← j
9:              end if
10:         end for
11:         swap A[i] dan A[min]
12:     end for
13: end procedure
```

Lampiran 2: Pseudocode Bubble Sort

Algorithm 7 Bubble sort

```
1:  procedure BubbleSort
2:      Input: array A[0..n − 1]: yang dapat diurutkan
3:      Output: array yang terurut ascending
4:      for i ← 0 to n − 2 do
5:          for j ← 0 to n − 2 − i do
6:              if A[j + 1] < A[j] then
7:                  swap A[i] dan A[min]
8:              end if
9:          end for
10:     end for
11: end procedure
```

end of slide...