3.3 - Algoritma *Brute Force* (bagian 3)

[KOMS120403]

Desain dan Analisis Algoritma (2023/2024)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 4 (March 2024)

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*?
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

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

```
procedure SelectionSort(A[1..n])
    for i = n − 1 downto 1 do                      n-1
        maxIdx = i
        for j = 0 to i − 1 do                      n-1
            if a[j] >= a[maxIdx] then              (n-1) + (n-2) + ... + 1
                maxIdx = j                         = n(n-1) / 2
            end if
        end for
        swap(a[i], a[maxIdx])                      n-1
    end for
end procedure
```

Kompleksitas waktu: $\mathcal{O}(n^2)$ *Dapatkah Anda jelaskan mengapa?*

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 tidak dalam urutan yang benar
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 merupakan 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

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

Bubble Sort (5): Analisis kompleksitas

- Satu iterasi dari loop dalam ("if condition" dan "swap") membutuhkan waktu konstan (memiliki batas atas c).
- Untuk dua loop bersarang:
  - *Outer loop* (loop luar): memuat n iterasi
  - *Inner loop* (loop dalam):
    - 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

```
procedure BubbleSort2(A[1..n])
    for i = n − 1 downto 1 do
        sorted = True
        for j = 1 to i do
            if a[j − 1] > a[j] then
                swap(a[j], a[j − 1])
                sorted = False
            end if
        end for
        if (sorted) then
            return
        end if
    end for
end procedure
```

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

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

Bagian 3. Insertion Sort

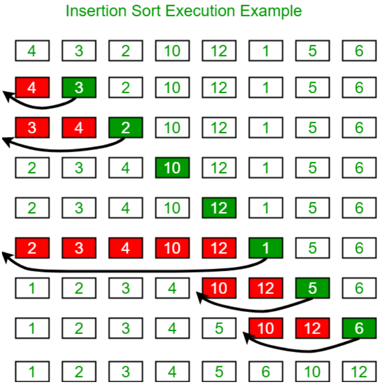
Insertion sort (1): Algoritma

Untuk mengurutkan array $A[0..n-1]$ dengan ukuran n dalam urutan naik (*ascending*):

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

procedure INSERTIONSORT($A[0..n-1]$: ordorable array)
    $i \leftarrow 1$
    while $i < n$ **do** ▷ We'll find the correct position for $A[1], A[2], \ldots, A[n-1]$
        key $\leftarrow A[i]$ ▷ 'key' is the current element that will be inserted
        $j \leftarrow i - 1$ ▷ Using index j, we'll find the correct position for $A[i]$
        while $j \geq 0$ **and** $A[j] >$ key **do**
            $A[j+1] \leftarrow A[j]$ ▷ $A[j]$ is shifted one position to the right (to index $j+1$)
            $j \leftarrow j - 1$ ▷ Decrement j (until the correct position is found)
        end while
        $A[j+1] \leftarrow$ key ▷ Once found, 'key' is inserted in the correct position (at idx $j+1$)
        $i \leftarrow i + 1$ ▷ Increment i to work on the next 'key'
    end while
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

```
procedure InsertionSort(A[0..n − 1]: ordorable array)
    i ← 1
    while i < n do                              ▷ 'while loop' involves (n − 1) iterations
        key ← A[i]
        j ← i − 1
        while j ≥ 0 and A[j] > key do           ▷ In the worst case: ∃ (n − 1) iterations
            A[j + 1] ← A[j]
            j ← j − 1
        end while
        A[j + 1] ← key
        i ← i + 1
    end while
end procedure
```

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

Membuktikan kebenaran melalui loop invarian

- Ingat kembali definisi kebenaran algoritma:

  > *algoritma dikatakan benar apabila algoritma memberikan return/output/solusi yang benar untuk setiap instance yang valid*

- Cara standar untuk membuktikan kebenaran adalah dengan memeriksa karakteristik invariansi loop

Karakteristik invariansi loop:

- Yakni merupakan **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, memberikan output yang benar
- 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)

```
procedure SelectionSort(A[0..n − 1]: ordorable array)
    for i = n − 1 downto 1 do
        maxIdx = i
        for j = 0 to i − 1 do
            if a[j] >= a[maxIdx] then
                maxIdx = j
            end if
        end for
        swap(a[i], a[maxIdx])
    end for
end procedure
```

Invariansi loop. Di awal setiap iterasi dari loop terluar:

- Sublist $A[0..i]$ berisi $(i+1)$ elemen 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).

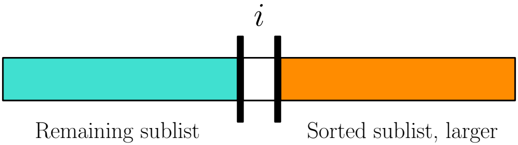
1. Kebenaran algoritma Selection Sort: Invariansi loop (2)

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 sudah 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)

Rangkuman pembuktian dengan invariansi loop pada Selection Sort

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

  > *elemen pada sebelah kanan indeks utama berada dalam urutan terurut dan bernilai lebih dari (atau sama dengan) 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" di atas dapat diterapkan untuk algoritma Selection Sort yang diberikan pada buku Anany Levitin?
- Coba Anda buktikan kebenaran BubbleSort dan Insertion Sort! dengan menggunakan teknik invariansi loop!
- Apakah metode "invariansi loop" dapat diaplikasikan untuk BubbleSort dan Insertion Sort? Jika iya, jelaskan invariansi loop untuk setiap algoritma!

Lampiran 1: Pseudocode Selection Sort

Algorithm 6 Selection sort

```
procedure SelectionSort
    Input: array A[0..n − 1]: yang dapat diurutkan
    Output: array yang terurut ascending
    for i ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[min] then
                min ← j
            end if
        end for
        swap A[i] dan A[min]
    end for
end procedure
```

Lampiran 2: Pseudocode Bubble Sort

Algorithm 7 Bubble sort

```
procedure BubbleSort
    Input: array A[0..n − 1]: yang dapat diurutkan
    Output: array yang terurut ascending
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if A[j + 1] < A[j] then
                swap A[i] dan A[min]
            end if
        end for
    end for
end procedure
```

end of slide...