03 - Brute Force Algorithm (part 2)

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 28 Feb - 4 March 2022

Table of contents

Q n A

1. Why is brute-force still used?

Q n A

1. Why is brute-force still used?
2. Explain the assignment problem!

Q n A

1. Why is brute-force still used?
2. Explain the assignment problem!
3. Explain the partition problem!

Q n A

1. Why is brute-force still used?
2. Explain the assignment problem!
3. Explain the partition problem!
4. Explain the magic square problem!

Q n A

1. Why is brute-force still used?
2. Explain the assignment problem!
3. Explain the partition problem!
4. Explain the magic square problem!
5. How heuristic technique can help in improving the efficiency of brute-force technique to solve the magic square problem?

Selection Sort

Selection Sort (1): Algorithm

Problem: Given an array of n ordorable elements.
Sort the array and output the sorted array in non-decreasing order.

1. Find the largest item x in the range of $[0..n-1]$
2. Swap x with the $(n-1)^{\text{th}}$ item
3. Reduce n by 1 and go to Step 1

Selection Sort (2)

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?

Unsorted item

Largest item for the current iteration

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): Algorithm (*by minimum*)

What is the difference with the following selection-sort algorithm?

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: Selection sort algorithm in the book of Anany Levitin

Selection Sort (5): Complexity analysis

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
```

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

Bubble Sort

Bubble Sort (1): Algorithm

Idea: Given an array of n items

1. Compare pair of adjacent items
2. Swap if the items are out of order
3. Repeat until the end of array
   - The largest item will be at the last position
4. Reduce n by 1 and go to Step 1

Bubble Sort (2): Example

(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)

Does the following algorithm also define bubble-sort algorithm? Explain!

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): Complexity analysis

- one iteration of the inner loop (test and swap) requires time bounded by a constant c
- Two nested loops
  - Outer loop: exactly n iterations
  - Inner loop:
    - When $i = 0$, $(n-1)$ iterations
    - When $i = 1$, $(n-2)$ iterations
    - $\cdots$
    - When $i = n-1$, 0 iterations
- Total number of iterations: $0 + 1 + \cdots + (n-1) = \frac{n(n-1)}{2}$
- Total execution time: $c \cdot \frac{n(n-1)}{2} = \mathcal{O}(n^2)$

Bubble Sort (6): Algorithm (*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): Complexity analysis (*version 2*)

- Worst-case
  - Input is in descending order
  - Running time: $\mathcal{O}(n^2)$
- Best-case
  - Input is already in ascending order
  - The algorithm return after a single outer iteration
  - Running time: $\mathcal{O}(n)$

Insertion Sort

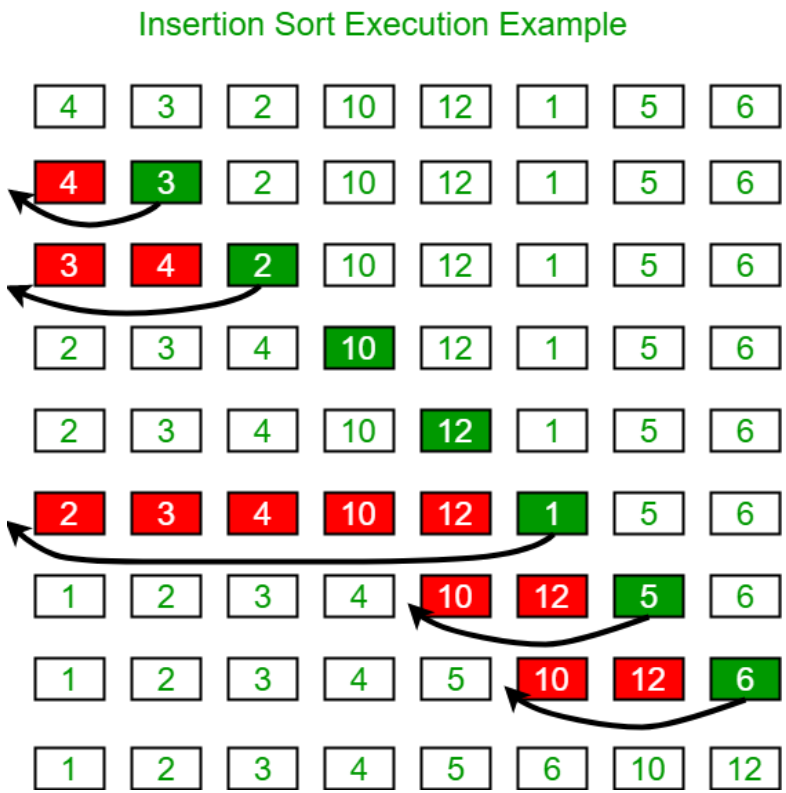
Insertion sort (1): Algorithm

To sort an array $A[0..n-1]$ of size n in ascending order:

1. Iterate from $A[0]$ to $A[n-1]$ over the array.
2. Compare the current element (named as 'key') to its predecessor.
3. If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Insertion sort (2): Example

Remark. We skipped the discussion of this part during the lecture. Please read it carefully yourself!



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): Time complexity

Complexity: $\mathcal{O}(n^2)$ (because there are two nested while loops, each of $\mathcal{O}(n)$-complexity)

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
```

Proving correctness: loop invariant

Proving correctness through loop invariant

- Recall the definition correct algorithm: *one returns the correct solution for every valid instance of a problem*
- A standard way to prove correctness: loop invariance property

The loop invariance property:

- It is a **key property** of the data manipulated by the main loop of an algorithm
  - The property must be defined and can help us understand why the algorithm is correct
  - We must show that the property holds in the initial case, is maintained each iteration, and that when the loop terminates the property yields correctness
- Determining this property in general can be difficult. But in many algorithms, the property is often the key defining feature of the algorithm.

From loop invariance to correctness

Three main characteristics of loop invariant

1. Initialization: The loop invariance must be true prior to the first iteration of the loop.

2. Maintenance: If the property holds prior to an iteration of the loop, it must still hold after the iteration is complete.

3. Termination: When the loop terminates, the invariant provides a useful property that helps demonstrate that the algorithm is correct.

- To prove correctness, we must prove the above about the loop invariance property
- *Characteristics (1) and (2) are similar to induction.* When they hold, the loop invariant is true prior to every iteration of the loop
- *Characteristic (3) is where it differs from induction* and is the most important piece. We are not showing that the loop invariant holds *ad infinitum*, but rather that it results in a correct answer after a finite number of steps

Correctness of selection sort: Loop invariant (1)

Example of loop invariant (*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
```

Loop invariant. At the start of each iteration of the outermost loop:

- The sublist $A[0..i]$ contains the remaining $i + 1$ elements. Our goal is to put the correct element at position i.
- The sublist $A[i + 1..n - 1]$ consists of the $n - 1 - i$ largest elements of A in the correct order.

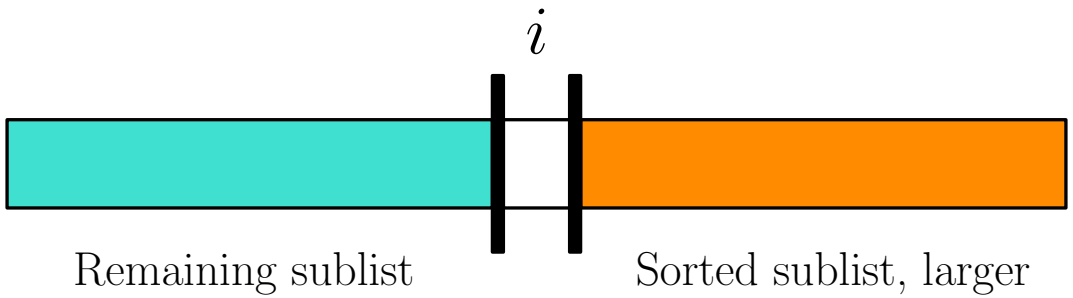
Correctness of selection sort: Loop invariant (2)

Essentially, the loop invariant says that at each step, the data set can be divided into two parts:

- The part from i to the left on which the algorithm is still working
- The part to the right of i is a sorted sublist from elements in 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.



Correctness of selection sort: Loop invariant (3)

What did we prove?

- Recall that the sorting problem is to take a list of unordered items and order them by value.
- We showed that the SelectionSort algorithm will, at each step, preserve the property that items to the right of the main index are in sorted order and not less than any item to the left.
- If this property holds at initialization, is maintained each step, and terminates properly, SelectionSort must be a correct implementation of an algorithm for solving the sorting problem.

Correctness of BubbleSort and InsertionSort?

Exercises:

- Does the "loop-invariant" proof still work for the selection-sort-algorithm given in the book of Levitin?
- Prove the correctness of BubbleSort and Insertion Sort!
- Does the loop-invariance method work? Define the loop invariant for each algorithm!