Last Lecture

- Recursion
  - Towers of Hanoi
    - Provide a *hard* problem with a *clean* solution!
  - Problems with Recursive algorithms
    - Space complexity
    - Function call overhead
    - Stack overflow
    - Redundant computation
    - Difficult in understanding and debugging
    - Non-optimal time complexity
  - Recursive algorithms vs iterative algorithms
- Sorting
  - What is sorting?
  - Ordering relation
  - Inversion

A Question

Given the following statements, how is X related to Y?

1. Y and Z are children of D who is a wife of X.
2. R's sister X is married to Ys father.

- A ① alone is sufficient while ② alone is not sufficient
- B ② alone is sufficient while ① alone is not sufficient
- C Either ① or ② is sufficient
- D Neither ① nor ② is sufficient
- E Both ① and ② are sufficient

CPT108 Data Structures and Algorithms

Lecture 9

Sorting

Selection Sort, Insertion Sort, and Bubble Sort

Outline

What is Sorting?

- A process of rearranging data elements based on some relationship between them

Why we need this?

- Enable efficient data storage, search, and retrieval, e.g.,
  - the complexity of searching a particular element (in an array or list) is reduced

What is Sorting? (cont.)

An **ordering relation** for any elements in a set has the following properties (Knuth, 1988):

Definition (Law of Trichotomy (Cormen et al., 2022))

For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$

Definition (Law of Transitivity (Cormen et al., 2022))

For a binary relation R on a set X is *transitive* if, for $x, y, z \in X$, $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.

$$\begin{tikzcd} a \arrow[r, "R"] \arrow[rr, bend left, "R"] & b \arrow[r, "R"] & c \end{tikzcd}$$

An ordering relation with the properties above is know as a *total order*.

What is Sorting? (cont.)

Example of ordering relation

Comparing Strings:

Let $\mathtt{a}$ =“pine”, $\mathtt{b}$ =“apple”, $\mathtt{c}$ =“pineapple”, and $\mathtt{d}$ =“mango”.

We can:

- Compare the strings based on their lengths
  - Law of Trichotomy:
    $len(\mathtt{a}) < len(\mathtt{b})$, $len(\mathtt{c}) > len(\mathtt{d})$, $len(\mathtt{a}) = len(\mathtt{d})$, etc.
  - Law of Transitivity:
    $\because len(\mathtt{a}) < len(\mathtt{b})$ and $len(\mathtt{b}) < len(\mathtt{c})$,
    $\Rightarrow \quad len(\mathtt{a}) < len(\mathtt{c})$
- Compare the strings based on the alphabetical order
  - Law of Trichotomy:
    $alph(\mathtt{b}) < alph(\mathtt{a})$, $alph(\mathtt{b}) < alph(\mathtt{c})$, $alph(\mathtt{d}) > alph(\mathtt{c})$, etc.
  - Law of Transitivity:
    $\because alph(\mathtt{b}) < alph(\mathtt{a})$ and $alph(\mathtt{a}) < alph(\mathtt{c})$,
    $\Rightarrow \quad alph(\mathtt{b}) < alph(\mathtt{c})$

What is Sorting? (cont.)

A formal definition (Cormen et al., 2022):

Input: A sequence of n numbers $\langle a_1, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example

Given an input sequence $\langle 57, 12, 27, 69, 87, 31, 43 \rangle$, a correct sorting algorithm return as output the sequence $\langle 12, 27, 31, 43, 57, 69, 87 \rangle$.

Example

Based on the definition above, when comparing two strings based on their lengths in the previous slide, we can have two valid sorts:

- “pine”, “apple”, “mango” “pineapple”
- “pine”, “mango”, “apple” “pineapple”

What is Sorting? (cont.)

- An *inversion* is a pair of elements that are out of order w.r.t. the ordering relation.
- For example, given the following sequence of numbers:

61 17 42 29 98 75 6

There are 11 inversions out of 21 max!

61-17	61-42	61-29	61-6
17-6	42-29	42-6	29-6
98-75	98-6	75-6	

Therefore, another way to state sorting is:

> Given a sequence of elements with Z inversions, our goal is to perform a sequence of operations that reduces inversions to 0.

What is Sorting? (cont.)

Implementation note

Ordering relation are typically realized in the form of `compareTo` or `compare` methods.

We have to consider four cases in the implementation.

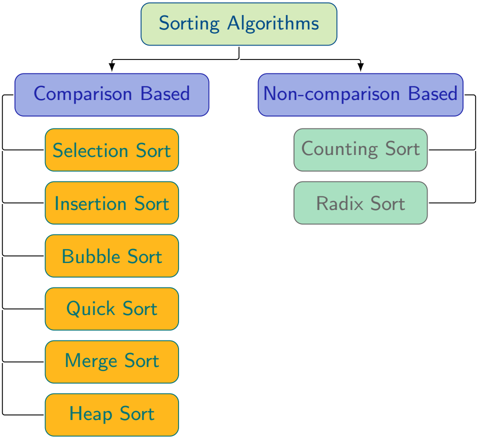
```java
import java.util.Comparator;

public class LengthComparator
    implements Comparator<String> {

  @Override
  public int compare(String s1, String s2)
        {
    if (null == s1) {
      if (null == s2) return 0;
      return -1;
    } else {
      if (null == s2) return 1;
      return s1.length() - s2.length();
    }
  }

}
```

	s_1	s_2
case 1	null	null
case 2	null	non-null
case 3	non-null	null
case 4	non-null	non-null

What is Sorting? (cont.)

Categories of sorting algorithms

- Broadly classified into two categories:



Outline

Selection sort

Pseudocode

For each elements in the array
  Set the first unsorted element as the `minimum`
  Find the `smallest` element in the unsorted portion of the array
  If `smallest` < `minimum`
    swap(`minimum`, `smallest`)

Algorithm

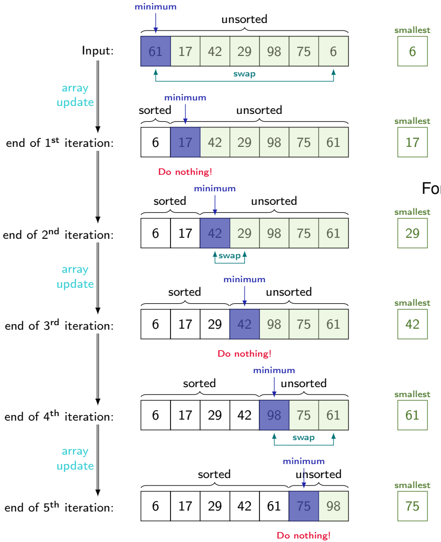
$\textsc{Selection-Sort}(A, n)$

// *A*: Array of items
// *n*: size of the array

for $i = 0$ **to** $n - 1$
  $smallest = \textsc{Find-Smallest}(A, i, n-1)$
  if $A[i] > A[smallest]$
    $\textsc{Swap}(i, smallest)$

Selection sort

An example



<u>Pseudocode</u>

For each elements in the array
  Set the first unsorted element as the `minimum`
  Find the `smallest` element in the unsorted portion of the array
  If `smallest` < `minimum`
    swap(`minimum`, `smallest`)

Sorted Array

6	17	29	42	61	75	98

Selection sort

Exercise

Input:

21	21	48	16	51	60	73	24

Pseudocode

For each elements in the array
  Set the first unsorted element as the `minimum`
  Find the `smallest` element in the unsorted portion of the array
  If `smallest` < `minimum`
    swap(`minimum`, `smallest`)

Selection sort

Complexity

repeat
N-times

For each elements in the array
    Set the first unsorted element as the `minimum`
    Find the `smallest` element in the unsorted portion of the array
    If `smallest` < `minimum`
        swap(`minimum`, `smallest`)

$O(N)$

$\Rightarrow$ *Overall time complexity* $= O(N^2)$

Selection sort

Summary (Geeksforgeeks.org, 2024c)

Advantages

- Simple and easy to implement
- Works well with *small* dataset
- It is an in-place algorithm, as it does not require extra space

Disadvantages

- Complexity of $O(N^2)$ in the worst-case
- Does not work well on *large* dataset
- Does not preserve relative order of elements with equal value, which means it is not stable

Outline

Insertion sort

- Inspired from the way in which we sort playing cards.

Insertion sort

Pseudocode

Mark first element as sorted
For each element `key` in the unsorted portion of the array
    Compare `key` with the element to its left
    If the current element is smaller
        swap them
    else
        continue with next element

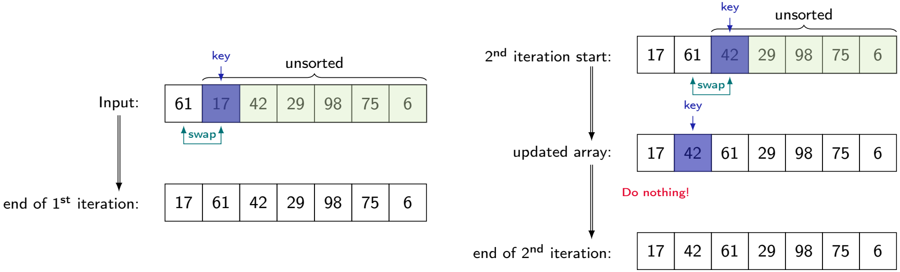
Algorithm

INSERTION-SORT(A, n)

for $i = 1$ **to** $n - 1$
    $key = A[i]$
    // Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.
    $j = i - 1$
    while $j > 0$ and $A[j] > key$
        $A[j + 1] = A[j]$
        $j = j - 1$
    $A[j + 1] = key$

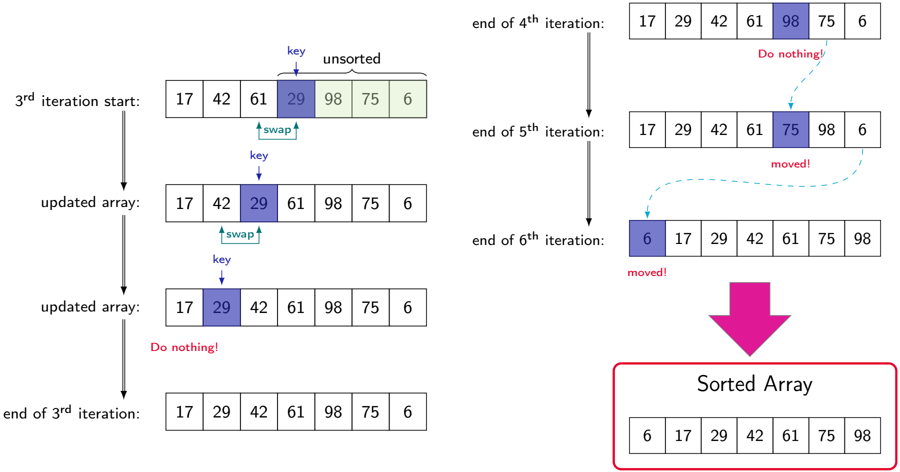
Insertion sort

An example



Insertion sort

An example



Insertion sort

Exercise

Input:

21	21	48	16	51	60	73	24

Pseudocode

Mark first element as sorted
For each element `key` in the unsorted portion of the array
    Compare `key` with the element to its left
    If the current element is smaller
        swap them
    else
        break the loop and continue with next element

Insertion sort

Complexity

$\boxed{\text{Mark first element as sorted}}$ $O(1)$

For each element `key` in the unsorted portion of the array

  Compare `key` with the element to its left

  $\boxed{\begin{array}{l}\text{If the current element is smaller}\\ \quad\text{swap them}\end{array}}$ $O(1)$

  $\boxed{\begin{array}{l}\text{else}\\ \quad\text{break the loop and continue with next element}\end{array}}$ $O(1)$

repeat N-times for each `key` $\Rightarrow O(N)$

repeat N-times

$$\Rightarrow \begin{array}{l} \textit{Worst-case time complexity} = O(N^2) \\ \textit{Best-case time complexity} = O(N) \end{array}$$

Insertion sort

Summary (Geeksforgeeks.org, 2024b)

Characteristics of Insertion Sort

- Simple and easy to implement
- Efficient for *small* dataset
- Adaptive in nature, i.e., it is appropriate for datasets that are already partially sorted
- It is an in-place algorithm, as it does not require extra space
- It preserves the relative order of elements with equal value, which means it is stable
- It is stable meaning that it preserve the relative order of elements with equal value

Outline

Bubble sort

Pseudocode

```
For subarray_size =0 to N-2
    For index=0 to N - 2 - subarray_size
        if the index-th element < the (index+1)-th element
            swap the two elements
```

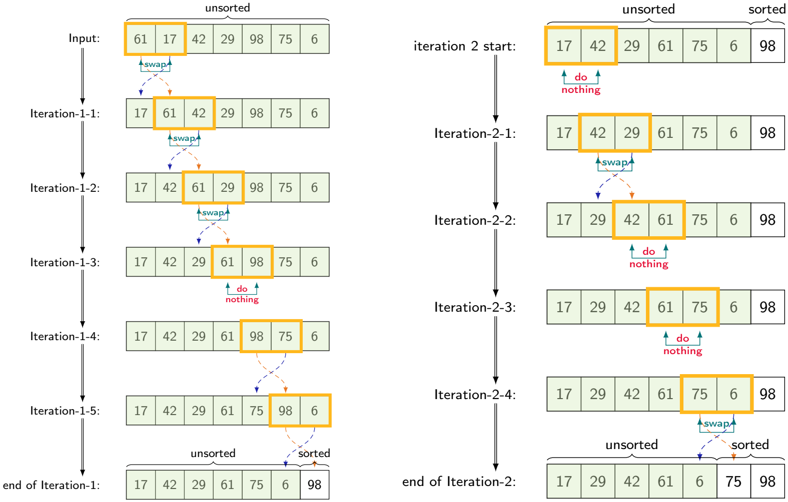
Algorithm

BUBBLE-SORT(A, n)

for $i = 0$ **to** $n - 2$
  for $j = 0$ **to** $(n - 2 - i)$
    if $A[j] > A[j+1]$
      SWAP($A[j], A[j+1]$)

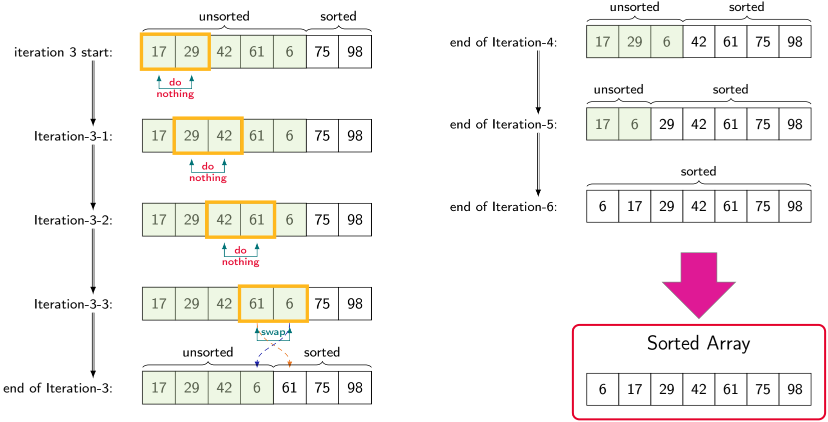
Bubble sort

An example



Bubble sort

An example (cont.)



Bubble sort

Exercise

Input:

21	21	48	16	51	60	73	24

Pseudocode

```
For subarray_size =0 to N-2
    For index=0 to N -2 - subarray_size
        if index-th element < (index+1)-th element
            swap the two elements
```

Bubble sort

Complexity

```
For subarray_size =0 to N-2
    For index=0 to N -2 - subarray_size
        if index-th element < (index+1)-th element
            swap the two elements
```

$O(1)$

repeat $O(N)$-times for each element

repeat $(N-1)$-times

$\Rightarrow$ *Overall time complexity* $= O(N^2)$

Bubble sort

Summary (Geeksforgeeks.org, 2024a)

Advantages

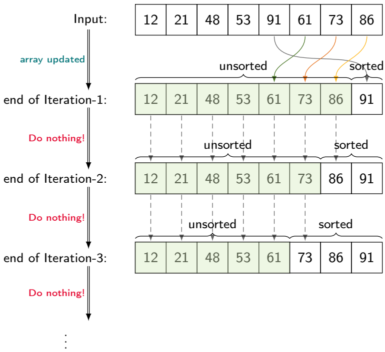
- Easy to understand and implement
- Require no additional space
- It is an in-place algorithm, as it does not require extra space
- It preserves the relative order of elements with equal value, which means it is stable

Disadvantages

- Complexity of $O(N^2)$ which makes it very slow for *large* dataset

Bubble sort: An Improved Version

Consider the scenario as shown below:



Can we improve the performance of the bubble sort?

Bubble sort: An Improved Version (cont.)

```
For subarray_size =0 to N-2
    For index=0 to N -2 - subarray_size
        if index-th element < (index+1)-th element
            swap the two elements
```

add a flag to indicate if the array is already sorted

```
For subarray_size =0 to N-2
    isSorted=True
    For index=0 to subarray_size -2
        if index-th element < (index+1)-th element
            swap the two elements
            isSorted=False
    If (isSorted) break the loop
```

change the flag if the array has been updated, i.e., not sorted

exit the loop if the array is already sorted

Improved Bubble sort

Complexity

```
For subarray_size =0 to N-2
    isSorted=True
    For index=0 to subarray_size -2
        if index-th element < (index+1)-th element
            swap the two elements
            isSorted=False
    If (isSorted) break the loop
```

$O(1)$ if the array is already sorted

repeat $O(N)$-times for each element

no repeat if the array is already sorted

$\Rightarrow$ *Best-case time complexity* $= O(N)$

Outline

Summary

	Worst	Best
Selection sort	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$
Bubble sort	$O(n^2)$	$O(n^2)$
Improved Bubble sort	$O(n^2)$	$O(n)$

Reading

- Chapter 2 & 3, Cormen (2022)

References I

- Cormen, Thomas H. et al. (2022). *Introduction to Algorithms*. 4th. MIT Press.
- Geeksforgeeks.org (2024a). *Bubble Sort — Data Structure and Algorithm Tutorials*. Online: `https://www.geeksforgeeks.org/bubble-sort/`. [last accessed: 20 Mar 2024].
- — (2024b). *Insertion Sort — Data Structure and Algorithm Tutorials*. Online: `https://www.geeksforgeeks.org/insertion-sort/`. [last accessed: 20 Mar 2024].
- — (2024c). *Selection Sort — Data Structure and Algorithm Tutorials*. Online: `https://www.geeksforgeeks.org/selection-sort/`. [last accessed: 20 Mar 2024].
- Knuth, Donald E. (1988). *The Art of Computer Programming*. 2nd. Vol. 3. Reading, Massachusetts: Addison Wesley.