Sorting(1/2)

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

Bubble Sort
Selection Sort
Insertion Sort

Objectives & Agenda

- **Objectives:**
  - Undrestand the principles of sorting.
  - Understand the basic algorithms of sorting.

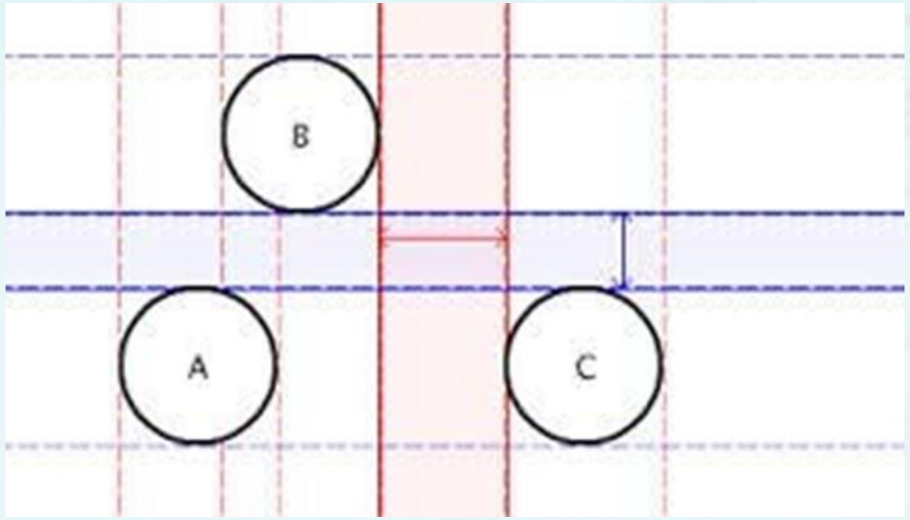
- **Agenda**
  - Motivation
  - Bubble Sort algorithm
  - Selection Sort algorithm
  - Insertion Sort algorithm
  - Time complexity – Big O

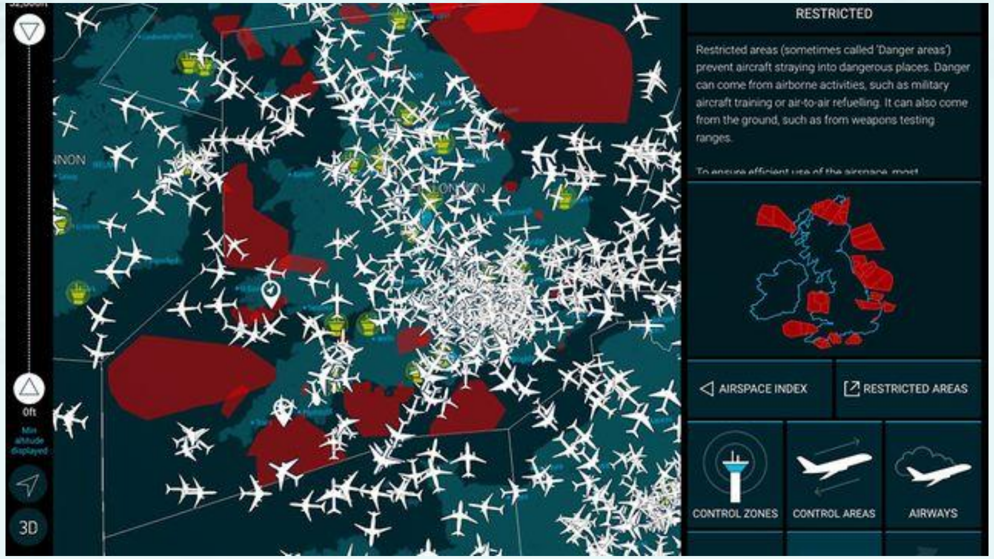
Sorting: One of the Most Common Activities on Computers

- **Example 1:**
  - Alphabetically sorted names:
    - e.g., names in phone book, street names in map, file names in a folder
  - Advantages:
    - Can use efficient search algorithms:
    - Binary search finds item in $O(\log n)$ time
- **Example 2:**
  - Sorted numbers:
    - e.g., house prices, student IDs, grades, rankings
  - Advantages:
    - Can use efficient search algorithms (see example 1)
    - Easy to find position or range of values in sorted list, e.g., minimum value, median value, quartile values, all students with A grades, all houses within a certain price range etc.

Sorting: One of the Most Common Activities on Computers

- **Example 3:**
  - Sort objects in space.
    - e.g., Objects in a street, Objects in space
  - Advantages:
    - Can use efficient search algorithms, e.g., for collision detection





Sorting: Need comparisons and swaps

- In order to sort items, we will need to **compare** items and **swap** them if they are out of order.
- Number of **comparisons** and the number of **swaps** are the costly operations in the sorting process, and these affect the efficiency of a sorting algorithm.

Sorting: Considerations

- An **internal sort** requires that the collection of data fit entirely in the computer's main memory.
- An **external sort:** the collection of data will not fit in the computer's main memory all at once but must reside in secondary storage.
- For very large collections of data it is costly to create a new structure (list) and fill it with the sorted elements so we will look at sorting **in place.**
- **One pass** is defined as one trip through the data structure (or part of the structure) comparing and, if necessary, swapping elements along the way. (In these examples the data structure is a list of ints.)

- In these discussions we sort from smallest (on the left of the list) to largest (on the right of the list).

Bubble Sort(거품 정렬)

- IDEA:
  - Given is a list L of n value {L[0], … , L[n-1]}
  - Divide list into **unsorted (left)** and **sorted part (right** – initially empty):
    Unsorted: {L[0], … , L[n-1]} Sorted: {}
  - In each pass **compare adjacent elements** and swap elements not in correct order
    - largest element is **“bubbled”** to the right of the unsorted part
  - **Reduce size of unsorted part by one** and increase size of sorted part by one.
    After i-th pass:
    Unsorted: {L[0], … , L[n-1-i]} Sorted: {L[n-i],…,L[n-1]}
  - Repeat until unsorted part has a size of 1 – then all elements are sorted

Bubble Sort(거품 정렬)

- Given is a list L of n value `{L[0], … , L[n-1]}`
  - Divide list into **unsorted (left)** and **sorted part (right** - initially empty):
    `Unsorted: {L[0], … , L[n-1]}` **`Sorted: {}`**
  - In each pass compare adjacent elements and swap elements not in correct order
    → largest element is “bubbled” to the right of the unsorted part
  - Reduce size of unsorted part by one and increase size of sorted part by one.
    After i-th pass:
    `Unsorted: {L[0], … , L[n-1-i]}` **`Sorted: {L[n-i],…,L[n-1]}`**
  - Repeat until unsorted part has a size of 1, then all elements are sorted

29	10	14	37	13	`List to sort`
10	14	29	13	37	PASS 1 (4 Comp, 3 Swap)
10	14	13	29	37	PASS 2 (3 Comp, 1 Swap)
10	13	14	29	37	PASS 3 (2 Comp, 1 Swap)
10	13	14	29	37	PASS 4 (1 Comp, 0 Swap)

Bubble Sort(거품 정렬) Example

- It compares every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

1st Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Swap since 5 > 1
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**),

2nd Pass:

(1 4 2 5 8) –> (1 4 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 4 5 8) –> (1 2 4 5 8)
(1 2 4 5 8) –> (1 2 4 5 8)

3rd Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Bubble Sort(거품 정렬) Exercise

- It compares every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

54	26	93	17	77	31	44	55	20	List to sort
26	54	17	77	31	44	55	20	93	PASS 1 (8 Comp, 7 Swap)
27	17	54	31	44	55	20	77		PASS 2 (7 Comp, 5 Swap)
									PASS 3 (Comp, Swap)
									PASS 4 (Comp, Swap)
									PASS 5 (Comp, Swap)
									PASS 6 (Comp, Swap)
17	20	26							PASS 7 (2 Comp, 1 Swap)
17	20								PASS 8 (1 Comp, 0 Swap)

Bubble Sort(거품 정렬) Big-O

- For a list with n elements:
  - The number of comparisons?
    - pass 1 pass 2 pass 3 ... last pass
      n-1 n-2 n-3 … 1

      $1 + 2 + \ldots + (n\text{-}3) + (n\text{-}2) + (n\text{-}1) = \frac{1}{2}(n^2 - n)$

- Big O of the bubble sort is $O(n^2)$
  - The number of data increases 10 times, then it takes a 100 times longer.
  - On average, the number of swaps is half the number of comparisons.

Bubble Sort(거품 정렬) Summary

- Sorting is a necessary tool in computing.
- The bubble sort algorithm is simple, but slow.
  - It performs lots of **comparisons** $O(n^2)$ and many **swaps** in each pass additionally.

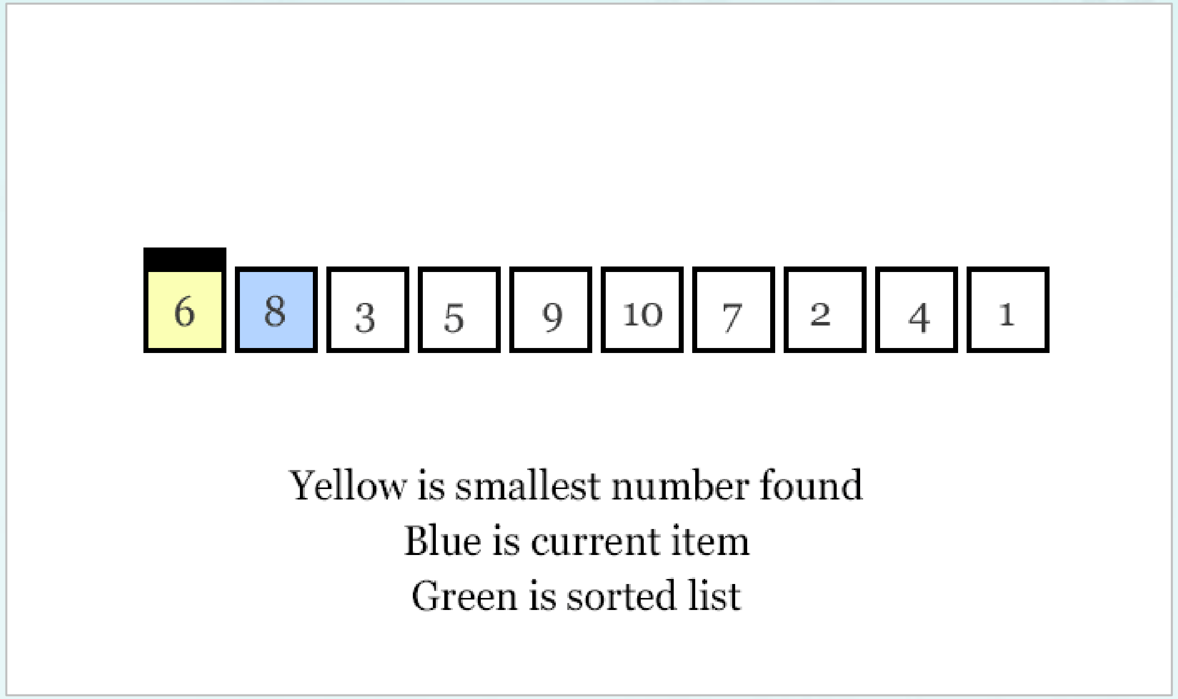
Selection Sort(선택 정렬)

- Given is a list L of n value **`{L[0], … , L[n-1]}`**
  - Divide list into unsorted (left) and sorted part (right - initially empty):
    `Unsorted: {L[0], … , L[n-1]}` **`Sorted: {}`**
  - In each pass find the **largest and place it to the right of the unsorted part** using a single swap.
    (Alternative method: Find the smallest and place it to the left of the unsorted part.)
  - Reduce size of unsorted part by one and increase size of sorted part by one.
    After i-th pass:
    `Unsorted: {L[0], … , L[n-1-i]}` **`Sorted: {L[n-i],…,L[n-1]}`**
  - Repeat until unsorted part has a size of 1, then all elements are sorted

29	10	14	37	13	`List to sort`
29	10	14	13	37	PASS 1 (4 Comp, 1 Swap)
13	10	14	29	37	PASS 2 (3 Comp, 1 Swap)
13	10	14	29	37	PASS 3 (2 Comp, 0 Swap)
10	13	14	29	37	PASS 4 (1 Comp, 1 Swap)

Selection Sort(선택 정렬) Example

- **Alternatively**, find the smallest and place it to the left of the unsorted part.



https://en.wikipedia.org/wiki/Selection_sort

Selection Sort(선택 정렬)

- Each pass we need to swap two elements of the list **once**.
  For example, at the end of the first pass we want to swap the element at position 3 with the element at position 8.

11	34	26	90	37	58	10	47	36

```
List to sort
```

- After the first pass:

11	34	26	36	37	58	10	47	90

`maxslot` `fillslot`

- After the second pass:

11	34	26	36	37	47	10	58	90

`maxslot` `fillslot`

Selection Sort(선택 정렬) "unstable"?

- **Why is a selection sort algorithm "unstable"?**
  - It picks the minimum and swaps it with the element at current position.
  - Suppose the array is:
    5 2 9 5 4 3 1 6
  - Let's distinguish the two 5's as 5(a) and 5(b) .
    5(a) 2 9 5(b) 4 3 1 6

Selection Sort(선택 정렬) "unstable"?

- **Why is a selection sort algorithm "unstable"?**
  - It picks the minimum and swaps it with the element at current position.
  - Suppose the array is:
    5 2 9 5 4 3 1 6
  - Let's distinguish the two 5's as 5(a) and 5(b) .
    5(a) 2 9 5(b) 4 3 1 6
  - After the first iteration, will be swapped with the element in 1st position:
    So the array becomes:
    1 2 9 5(b) 4 3 5(a) 6

5 2 9 5 4 3 1 6
1 2 9 5 4 3 5 6
1 2 9 5 4 3 5 6
1 2 3 5 4 9 5 6
1 2 3 4 5 9 5 6
1 2 3 4 5 9 5 6
1 2 3 4 5 5 9 6
1 2 3 4 5 5 6 9

Selection Sort(선택 정렬) "unstable"?

- **Why is a selection sort algorithm "unstable"?**
  - It picks the minimum and swaps it with the element at current position.
  - Suppose the array is:
    5 2 9 5 4 3 1 6
  - Let's distinguish the two 5's as 5(a) and 5(b) .
    5(a) 2 9 5(b) 4 3 1 6
  - After the first iteration, will be swapped with the element in 1st position:
    So the array becomes:
    1 2 9 5(b) 4 3 5(a) 6
  - Now, we clearly see that **5(a)** and **5(b)** are swapped in the sorted array.
    Therefore, this algorithm is **unstable.**

Selection Sort(선택 정렬) Exercise

54	**26**	**93**	**17**	**77**	**31**	**44**	**55**	**20**	List to sort
54	26	20	17	77	31	44	55	93	PASS 1 (8 Comp, 1 Swap)
									PASS 2 (Comp, Swap)
									PASS 3 (Comp, Swap)
									PASS 4 (Comp, Swap)
									PASS 5 (Comp, Swap)
									PASS 6 (Comp, Swap)
									PASS 7 (Comp, Swap)
17	**20**	**26**	**31**	**44**	**54**	**55**	**77**	**93**	PASS 8 (Comp, Swap)

Total PASS 8 (36 Comp, 5 Swap)

Selection Sort(선택 정렬) Big O

- For a list with n elements:
  - The number of comparisons?
  - pass 1   pass 2   pass 3   …   last pass
    $n-1$ $n-2$ $n-3$ ⋯   1

    $1 + 2 + \cdots + (n-3) + (n-2) + (n-1) = \frac{1}{2}(n^2 - n)$

- Big O of the selection sort is $O(n^2)$
  - The number of data increases 10 times, then it takes a 100 times longer.
  - However, it swaps less than Bubble Sort. It swaps just once each pass.

Selection Sort(선택 정렬) Big O

- What if the data is already sorted?
  - Swaps?
  - Comparisons?

29	10	14	37	13	List to sort
29	10	14	13	37	PASS 1 (4 Comp, 1 Swap)
13	10	14	29	37	PASS 2 (3 Comp, 1 Swap)
13	10	14	29	37	PASS 3 (2 Comp, 0 Swap)
10	13	14	29	37	PASS 4 (1 Comp, 1 Swap)

5	10	14	32	35	List to sort
5	10	14	32	35	PASS 1 (Comp, Swap)
5	10	14	32	35	PASS 2 (Comp, Swap)
5	10	14	32	35	PASS 3 (Comp, Swap)
5	10	14	32	35	PASS 4 (Comp, Swap)

Selection Sort(선택 정렬) Big O

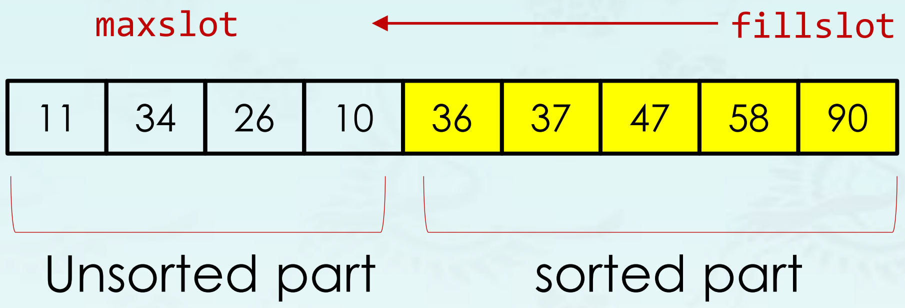
- What if the data is in reverse order?
  - Swaps?
  - Comparisons?

29	10	14	37	13	List to sort
10	14	29	13	37	PASS 1 (4 Comp, 1 Swap)
10	14	13	29	37	PASS 2 (3 Comp, 1 Swap)
10	13	14	29	37	PASS 3 (2 Comp, 0 Swap)
10	13	14	29	37	PASS 4 (1 Comp, 1 Swap)

35	32	14	10	5	List to sort
5	32	14	10	35	PASS 1 (Comp, Swap)
5	10	14	32	35	PASS 2 (Comp, Swap)
5	10	14	32	35	PASS 3 (Comp, Swap)
5	10	14	32	35	PASS 4 (Comp, Swap)

Selection Sort summary

- Divide array into unsorted (left) and sorted part (right, initially empty)
- Find largest value in unsorted part and place at end - after each pass sorted part increases by one and unsorted part reduces by one.



- It is also $O(n^2)$ algorithm, but it involves fewer swaps compared to Bubble Sort.

Bubble Sort vs Selection Sort

- Bubble and Selection Sort use the same number of comparisons.
- Bubble Sort does $O(n)$ swaps per pass on average, but Selection Sort only 1 swap per pass.
- Selection Sort typically executes faster than bubble sort.

- **How can we do better?**
  IDEA: Reduce number of comparisons by inserting into sorted array.

Insertion Sort(삽입 정렬)

- Given is a list L of n value {L[0], ⋯ , L[n−1]}
  - Divide list into sorted (left - initially only one element) and unsorted part (right):
    `Sorted: {L[0]}                Unsorted: {L[1], … , L[n-1]}`
  - In each pass, **take left most element from unsorted part** and place it into correct position of sorted part.
  - Reduce size of unsorted part by one and increase size of sorted part by one.
    After i-th pass:
    `Sorted: {L[0],…,L[i]}     Unsorted: {L[i+1], … , L[n-1-i]}`
  - Repeat until unsorted part is an empty list - then all elements are sorted.

Unsorted

29	10	14	13	18	List to sort
10	29	14	13	18	PASS 1 (1 Comp, 1 Shift)
10	14	29	13	18	PASS 2 (2 Comp, 1 Shift)
10	13	14	29	18	PASS 3 (3 Comp, 2 Shift)
10	13	14	18	29	PASS 4 (2 Comp, 1 Shift)

Insertion Sort(삽입 정렬)

- Given is a list L of n value {L[0], ⋯ , L[n-1]}
  - Divide list into sorted (left - initially only one element) and sorted part (right):
    `Sorted: {L[0]}` **`Unsorted: {L[1], … , L[n-1]}`**
  - In each pass, take left most element from unsorted part and place it into correct position of sorted part.
  - Reduce size of unsorted part by one and increase size of sorted part by one.
    After i-th pass:
    `Sorted: {L[0],…,L[i]}` **`Unsorted: {L[i+1], … , L[n-1-i]}`**
  - Repeat until unsorted part is an empty list - then all elements are sorted.

Unsorted

29	10	14	13	18	List to sort
10	29	14	13	18	PASS 1 (1 Comp, 1 Shift)
10	14	29	13	18	PASS 2 (2 Comp, 1 Shift)
10	13	14	29	18	PASS 3 (3 Comp, 2 Shift)
10	13	14	18	29	PASS 4 (2 Comp, 1 Shift)

29	10	14	13	18	pick 10 & inserted
10	29	14	13	18	
10	14	29	13	18	pick 14 & inserted
10	13	14	29	18	pick 13 & inserted
10	13	14	18	29	pick 18 & inserted

Insertion Sort(삽입 정렬)

6	28	34	35	37	90	**10**	27	36

- For example, to insert 10 into the sorted part of the list we need to store 10 into a temporary variable and move all the elements which are bigger than 10 up one position, then insert 10 into the empty slot.

temp = 10

6	28	34	35	37	90	_	27	36

Shift 5 list values

6	_	28	34	35	37	90	27	36

6	**10**	28	34	35	37	90	27	36

Insertion Sort(삽입 정렬) Example

- It works the way we sort playing cards in our hands. It builds the final sorted array one item at a time.
- "Stable" does not change the relative order of elements with equal keys.
- "In-place" only requires a constant amount O(1) of additional memory space.
- "Online" can sort a list as it receives it.

The partial sorted list (black) initially contains only the first element in the list.
With each iteration one element (red) is removed from the "not yet checked for order" input data and inserted in-place into the sorted list.

https://en.wikipedia.org/wiki/Insertion_sort

Insertion Sort(삽입 정렬) Exercise

	54	26	93	17	77	31	44	55	20	List to sort
pick 26 & inserted	26	54	93	17	77					PASS 1 (1 Comp, 1 Shift)
pick 93 & inserted	26	54	93	17	77					PASS 2 (1 Comp, 0 Shift)
pick 17 & inserted	17	26	54	93	77					PASS 3 (3 Comp, 3 Shift)
pick 77 & inserted										PASS 4 (Comp, Shift)
pick 31 & inserted										PASS 5 (Comp, Shift)
pick 44 & inserted										PASS 6 (Comp, Shift)
pick 55 & inserted	17	26	31	44	54	55	77	93	20	PASS 7 (3 Comp, 2 Shift)
pick 20 & inserted	17	20	26	31	44	54	55	77	93	PASS 8 (8 Comp, 7 Shift)

Total PASS 8 (26 Comp, 20 Shift)

Insertion Sort(삽입 정렬) Big-O

- For a list with n elements:
  - The number of comparisons in the WORST CASE?
  - pass 1 pass 2 pass 3 ... last pass
    1 2 3 ... n-3 n-2 n-1

    $1 + 2 + \ldots + (n-3) + (n-2) + (n-1) = ½(n^2 – n)$

- The time complexity of the insertion sort is O(n^2)
  - The number of data increases 10 times, then it takes a 100 times longer.

- Note 1: Best case O(n) ... when does this occur?
- Note 2: The number of shifts is equal or one smaller than the number of comparisons, so same order of magnitude.

Insertion Sort(삽입 정렬) Big-O

- What if the data is **already sorted**?
  - Move elements?
  - Comparisons?

29	10	14	13	18	List to sort
10	29	14	13	18	PASS 1 (1 Comp, 1 Shift)
10	14	29	13	18	PASS 2 (2 Comp, 1 Shift)
10	13	14	29	18	PASS 3 (3 Comp, 2 Shift)
10	13	14	18	29	PASS 4 (2 Comp, 1 Shift)

	5	10	14	32	35	List to sort
pick 10 & inserted	5	10	14	32	35	PASS 1 (Comp, Shift)
pick 14 & inserted	5	10	14	32	35	PASS 2 (Comp, Shift)
pick 32 & inserted	5	10	14	32	35	PASS 3 (Comp, Shift)
pick 35 & inserted	5	10	14	32	35	PASS 4 (Comp, Shift)

Insertion Sort(삽입 정렬) Big-O

- What if the data is in **reverse order**?
  - Move elements?
  - Comparisons?

29	10	14	13	18	List to sort
10	29	14	13	18	PASS 1 (1 Comp, 1 Shift)
10	14	29	13	18	PASS 2 (2 Comp, 1 Shift)
10	13	14	29	18	PASS 3 (3 Comp, 2 Shift)
10	13	14	18	29	PASS 4 (2 Comp, 1 Shift)

	35	32	14	10	5	List to sort
pick 32 & inserted	32	35	14	10	5	PASS 1 (Comp, Shift)
pick 14 & inserted	14	32	35	10	5	PASS 2 (Comp, Shift)
pick 10 & inserted	10	14	32	35	5	PASS 3 (Comp, Shift)
pick 5 & inserted	5	10	14	32	35	PASS 4 (Comp, Shift)

Insertion Sort Summary

- Insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less.
- **Insertion sort is known faster than selection sort** on average.
- For small lists, the insertion sort is appropriate due to its simplicity.
  For **almost sorted** lists, the insertion sort is **a good choice.**
- For large lists, all $O(n^2)$ algorithms, including the insertion sort, are prohibitively inefficient.

Running Time Matters

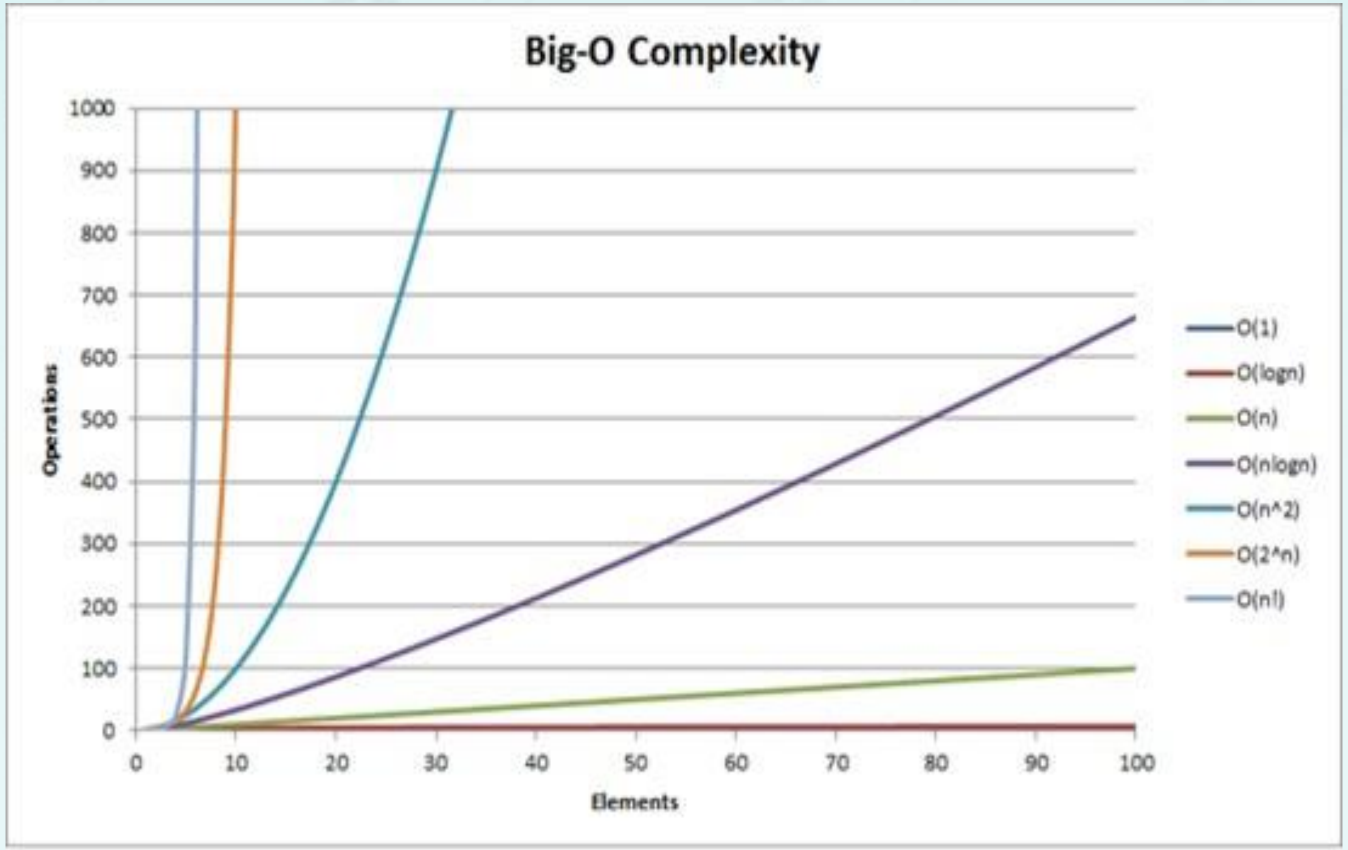
- All sorting algorithms (bubble, selection, insertion sorts) discussed so far had an $O(n^2)$ average and the worst-case complexity
  → In practice for large lists it is **too slow.**

- The **Timsort** algorithm (written in C - not using the Python interpreter) used by Python combines elements from **Merge sort** and **Insertion sort.**
  - Worst case and average case complexity $O(n \log n)$
  - Very fast for almost sorted lists

- All comparison-based sorting algorithms require at least $O(n \log n)$ time in the worst and average case.

Running Time Matters

- The usefulness of an algorithm in practice depends on the data size n and the complexity (Big O) of the algorithm (time and memory).
- In general algorithms with linear, logarithmic or low polynomial running time are acceptable.
  - $O\ (\log n)$
  - $O\ (n)$
  - $O\ (n^k)$ where k is a small constant, (in many cases $k\ < 2$ is ok)
- Algorithms with exponential or high polynomial running time are often of limited use.
  - $O\ (n^k)$ where k is a large constant, say $>\ 3$
  - $O\ (2^n), O\ (n^n)$



C++ For C Coders 7

Data Structures
C++ for C Coders

한동대학교 김영섭교수
idebtor@gmail.com

Bubble Sort
Selection Sort
Insertion Sort