Chapter Two

Simple Sorting and Searching Algorithms

Why do we study sorting and searching algorithms?

- These algorithms are the most common and useful tasks operated by computer system.
- Computers spend a lot of time for searching and sorting.

1. Simple Searching algorithms

Searching:- is a process of finding an element in a list of items or determining that the item is not in the list.

- To keep things simple, we shall deal with a list of numbers.
- A search method looks for a *key*, arrives by parameter.
- By convention, the method will return the index of the element corresponding to the key or, if unsuccessful, the value -1.

There are two simple searching algorithms:

a) Sequential Search, and

b) Binary Search

a). <u>Sequential Searching (Linear)</u>

- The most natural way of searching an item.
- Easy to understand and implement.

Algorithm:

- In a linear search, we start with top (beginning) of the list, and compare the element at top with the key.
- If we have a match, the search terminates and the index number is returned.
- If not, we go on the next element in the list.
- If we reach the end of the list without finding a match, we return -1.

Implementation: Assume the size of the list is n.

```
int LinearSearch(int list[ ], int key)
  {
    index=-1;
    for(int i=0; i<n; i++)
     {
       if(list[i]==key)
         {
           index=i;
           break;
         }
      }
            return index;
   }
```

Complexity Analysis:

- Big-Oh of sequential searching ➔ How many comparisons are made in the worst case ? n ➔O(n).

b). Binary Searching

- Assume sorted data.
- Use Divide and conquer strategy (approach).

Algorithm:

I. In a binary search, we look for the key in the *middle* of the list. If we get a match, the search is over.

II. If the key is greater than the element in the middle of the list, we make the top (upper) half the list to search.

III. If the key is smaller, we make the bottom (lower) half the list to search.

- Repeat the above steps (I,II and III) until one element remains.
- If this element matches return the index of the element, else return -1 index. (-1 shows that the key is not in the list).

Implementation:

```
 int BinarySearch(int list[ ], int key)
  {
    int found=0,index=0;
    int top=n-1,bottom=0,middle;
do{
      middle=(top + bottom)/2;
    if(key==list[middle])
       found=1;
    else{
       if(key<list[middle])
            top=middle-1;
       else
           bottom=middle+1;
        }
   }
}while(found==0 && top>=bottom);
```

```
if(found==0)
   index=-1;
 else
   index=middle;
return index;
}
```

Complexity Analysis:

Example: Find Big-Oh of Binary search algorithm in the worst case analysis.

➔ O(log n)

2. Simple Sorting Algorithms

<u>**Sorting:**</u> is a process of reordering a list of items in either increasing or decreasing order.

- Ordering a list of items is fundamental problem of computer science.
- Sorting is the most important operation performed by computers.
- Sorting is the first step in more complex algorithms.

Two basic properties of sorting algorithms:

<u>**In-place:**</u> It is possible to sort very large lists without the need to allocate additional working storage.

Simple Sorting Algorithms

Stable: If two elements that are equal, they will remain in the same relative position after sorting is completed.

Two classes of sorting algorithms:

$O(n^2)$:

- Includes the bubble, insertion, and selection sorting algorithms.

$O(n \log n)$:

- Includes the heap, merge, and quick sorting algorithms.

Simple sorting algorithms include:

i. Simple sorting
ii. Bubble Sorting
iii. Selection Sorting
iv. Insertion Sorting

I. Simple sorting

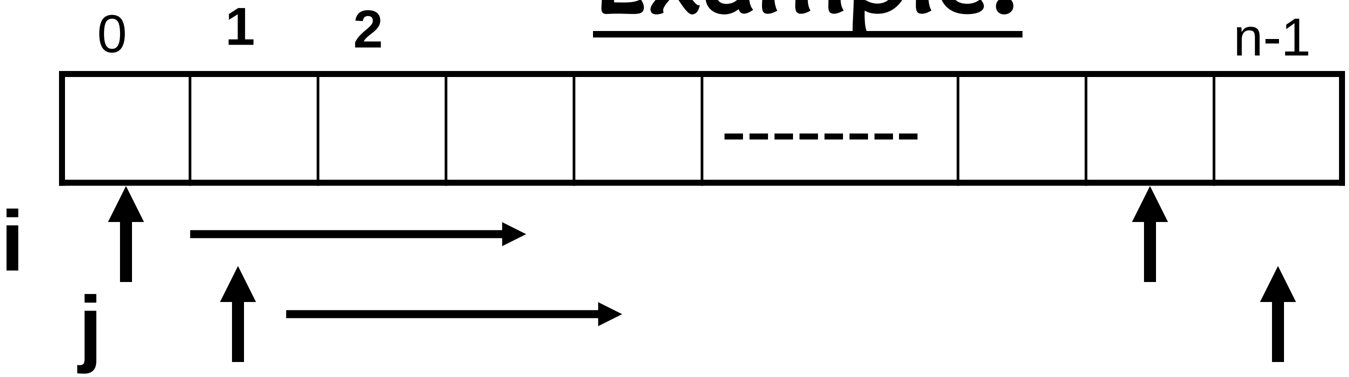
Algorithm:

- In simple sort algorithm the first element is compared with the second, third and all subsequent elements.
- If any one of the other elements is less than the current first element then the first element is swapped with that element.
- Eventually, after the last element of the list is considered and swapped, then the first element has the smallest element in the list.
- The above steps are repeated with the second, third and all subsequent elements.

Implementation:

```
Void SimpleSort(int list[])
{
  for(int i=0; i<=n-2;i++)
      for(int j=i+1; j<=n-1; j++)
          if(list[i] > list[j])
           {
                int temp;
                temp=list[i];
                list[i]=list[j];
                list[j]=temp;
            }
}
```

Example:



		4	2	3	1
i=0	j=1	2	4	3	1
	j=2	2	4	3	1
	j=3	1	4	3	2
i=1	j=2	1	3	4	2
	j=3	1	2	4	3
i=2	j=3	1	2	3	4

Analysis: O(?)

1st pass-----→ (n-1) comparisons

2nd pass----→ (n-2) comparisons

|

|

(n-1)th pass---→ 1 comparison

$T(n)=1+2+3+4+\text{-------}+(n-2)+(n-1)$

$= (n*(n-1))/2$

$=n^2/2-n/2$

$=O(n^2)$

Complexity Analysis:

- Analysis involves number of comparisons and swaps.
- How many comparisons?

  $1+2+3+\ldots+(n-1)= O(n^2)$
- How many swaps?

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

Example: Suppose we have 32 unsorted data.

a). How many comparisons are made by sequential search in the worst-case?

➔ Number of comparisons = **32.**

b). How many comparisons are made by binary search in the worst-case? (Assuming simple sorting).

➔ Number of comparisons = Number of comparisons for sorting + Number of comparisons for binary search

= (n*(n-1))/2 + logn

= 32/2(32-1)+ log 32

=**16*31 + 5**

c). How many comparisons are made by binary search in the worst-case if data is found to be already sorted?

➔ Number of comparisons = $\mathbf{log_2 32 = 5}$.

II.Bubble sort

Algorithm:

I. Compare each element (except the last one) with its neighbor to the right.
- If they are out of order, swap them
- This puts the largest element at the very end
- The last element is now in the correct and final place

II. Compare each element (except the last two) with its neighbor to the right.
- If they are out of order, swap them
- This puts the second largest element before last
- The last two elements are now in their correct and final places

III. Compare each element (except the last three) with its neighbor to the right.
IV. Continue as above until you have no unsorted elements on the left.

- Is the oldest, simplest, and slowest sort in use.
- It works by comparing each item in a list with an item next to it, and swap them if required.
- This causes the larger values to "bubble" to the end of the list while smaller values to "sink" towards the beginning of the list.
- In general case, bubble sort has $O(n^2)$ level of complexity.

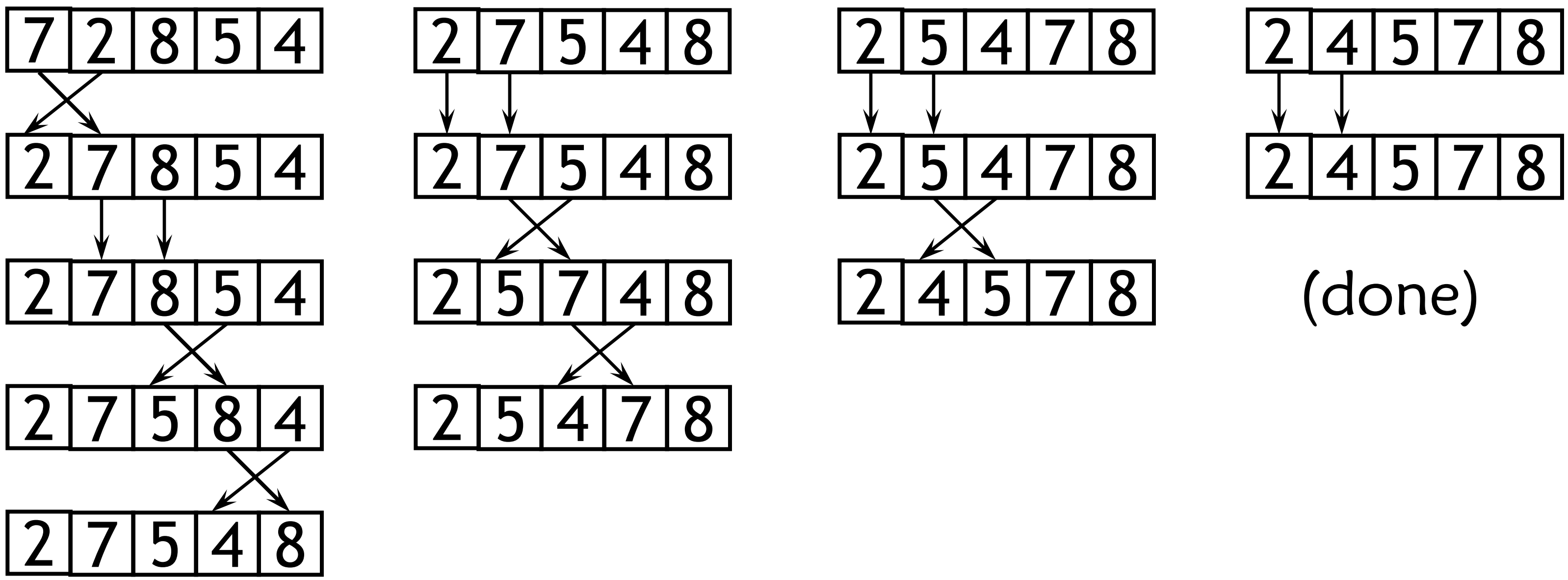
Advantage: Simplicity and ease of implementation.

Disadvantage: Horribly inefficient.

Example of Bubble sort

		4	**2**	3	1
i=3	j=1	2	**4**	**3**	1
	j=2	2	3	**4**	**1**
	j=3	**2**	**3**	1	4
i=2	j=1	2	**3**	**1**	4
	j=2	**2**	**1**	3	4
i=1	j=1	1	2	3	4

Example of Bubble sort



Implementation:

```
Void BubbleSort(int list[ ])
{
    int temp;
  for (int i=n-2; i>=0; i--) {
  for(int j=0;j<=i; j++)
     if (list[j] > list[j+1])
      {
            temp=list[j];
            list[j])=list[j+1];
            list[j]=temp;
      }
    }
}
```

Complexity Analysis:

- Analysis involves number of comparisons and swaps.
- How many comparisons?

  $1+2+3+\ldots+(n-1)= O(n^2)$
- How many swaps?

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

III. Selection Sort

Algorithm

- The selection sort algorithm is in many ways similar to simple sort algorithms.
- The idea of algorithm is quite simple. Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part is empty, while unsorted one contains whole array.
- At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one.
- When unsorted part becomes empty, algorithm *stops*.

- Works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled.
- Similar to the more efficient insertion sort.
- It yields a 60% performance improvement over the bubble sort.

Advantage: Simple and easy to implement.

Disadvantage: Inefficient for larger lists.

Example:

		7 9 11 3
i=0	j=1	7 9 11 3
	j=2	7 9 11 3
	j=3	3 9 11 7
i=1	j=2	3 9 11 7
	j=3	3 7 11 9
i=2	j=3	3 7 9 11


5 1 12 -5 16 2 12 14
5 1 12 -5 16 2 12 14
-5 1 12 5 16 2 12 14
-5 1 12 5 16 2 12 14
-5 1 2 5 16 12 12 14
5 1 2 5 16 12 12 14
-5 1 2 5 12 16 12 14
-5 1 2 5 12 12 16 14
-5 1 2 5 12 12 14 16

Implementation:

```
void selectionSort(int list[ ] ) {
      int minIndex, temp;
      for (int i = 0; i <= n - 2; i++) {
                minIndex = i;
           for (j = i + 1; j <= n-1; j++)
              if (list[j] < list[minIndex])
                   minIndex = j;
            if (minIndex != i) {
                 temp = list[i];
                 list[i] = list[minIndex];
                 list[minIndex] = temp;
            }
      }
}
```

Complexity Analysis

- Selection sort stops, when unsorted part becomes empty.
- As we know, on every step number of unsorted elements decreased by one.
- Therefore, selection sort makes n-1 steps (*n* is number of elements in array) of outer loop, before stop.
- Every step of outer loop requires finding minimum in unsorted part. Summing up, $(n - 1) + (n - 2) + \dots + 1$, results in $O(n^2)$ number of comparisons.
- Number of swaps may vary from zero (in case of sorted array) to n-1 (in case array was sorted in reversed order), which results in $O(n)$ number of swaps.
- Overall algorithm complexity is $O(n^2)$.
- Fact, that selection sort requires n-1 number of swaps at most, makes it very efficient in situations, when write operation is significantly more expensive, than read operation.

IV. Insertion Sort

Algorithm:

- Insertion sort algorithm somewhat resembles Selection Sort and Bubble sort.
- Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part contains first element of the array and unsorted one contains the rest.
- At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one.
- When unsorted part becomes empty, algorithm stops.

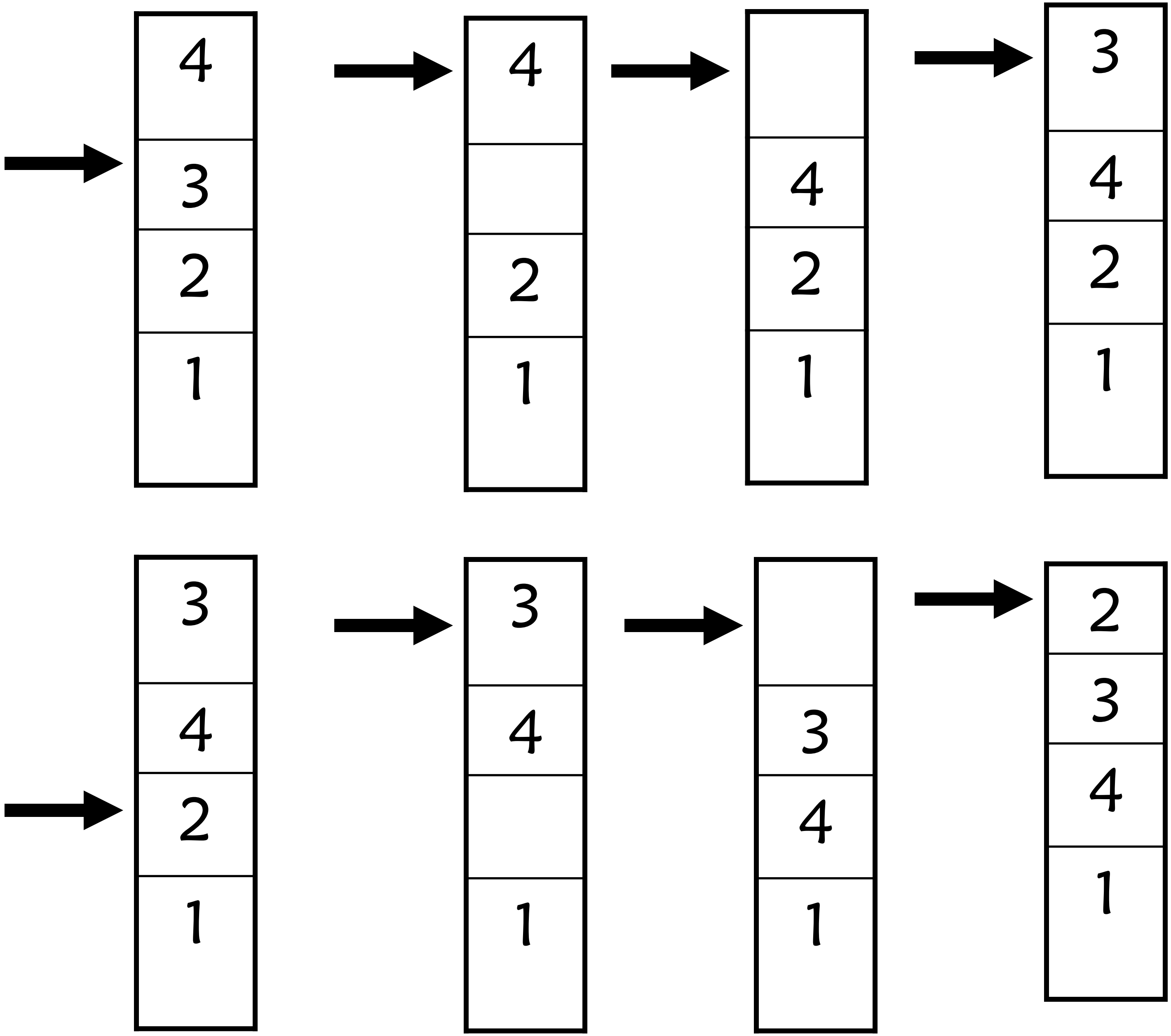
Using binary search

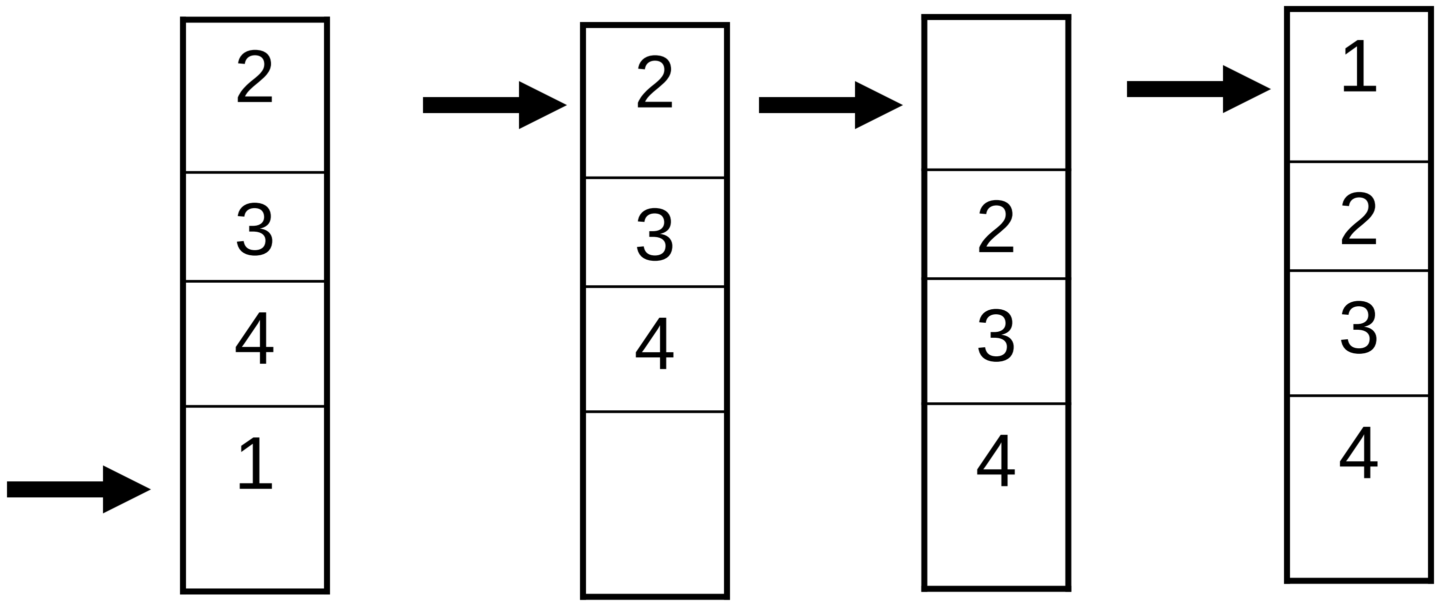
- It is reasonable to use binary search algorithm to find a proper place for insertion.
- This variant of the insertion sort is called binary insertion sort.
- After position for insertion is found, algorithm shifts the part of the array and inserts the element.

- Insertion sort works by inserting item into its proper place in the list.
- Insertion sort is simply like playing cards: To sort the cards in your hand, you extract a card, shift the remaining cards and then insert the extracted card in the correct place.
- This process is repeated until all the cards are in the correct sequence.
- Is over twice as fast as the bubble sort and is just as easy to implement as the selection sort.

Advantage: Relatively simple and easy to implement.

Disadvantage: Inefficient for large lists.


4
3
2
1
4
2
1
4
2
1
3
4
2
1
3
4
2
1
3
4
1
3
4
1
2
3
4
1


2
3
4
1
2
3
4
2
3
4
1
2
3
4

7	-5	2	16	4	unsorted
7	-5	2	16	4	-5 to be inserted
?	7	2	16	4	7 > -5, shift
-5	7	2	16	4	reached left boundary, insert -5
-5	7	2	16	4	2 to be inserted
-5	?	7	16	4	7 > 2, shift
-5	2	7	16	4	-5 < 2, insert 2
-5	2	7	16	4	16 to be inserted
-5	2	7	16	4	7 < 16, insert 16
-5	2	7	16	4	4 to be inserted
-5	2	7	?	16	16 > 4, shift
-5	2	?	7	16	7 > 4, shift
-5	2	4	7	16	2 < 4, insert 4
-5	2	4	7	16	sorted

C++ implementation

```
void InsertionSort(int list[])
 {
      for (int i = 1; i <= n-1; i++) {
        for(int j = i;j>=1; j--) {
          if(list[j-1] > list[j])
           {
             int temp = list[j];
             list[j] = list[j-1];
             list[j-1] = temp;
           }
          else
             break;
        }
      }
 }
```

Complexity Analysis

- The complexity of insertion sorting is O(n) at best case of an already sorted array and $O(n^2)$ at worst case, regardless of the method of insertion.
- Number of comparisons may vary depending on the insertion algorithm.
  - $O(n^2)$ for shifting or swapping methods.
  - O(nlogn) for binary insertion sort.