Introduction to Sorting

Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

If you ask me, how will I arrange a deck of shuffled cards in order, I would say, I will start by checking every card, and making the deck as I move on.

It can take me hours to arrange the deck in order, but that's how I will do it. Well, thank god, computers don't work like this. Since the beginning of the programming age, computer scientists have been working on solving the problem of sorting by coming up with various different algorithms to sort data.

The two main criterias to judge which algorithm is better than the other have been:

a. **Time taken to sort the given data.**

b. **Memory Space required to do so.**

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

a. **Bubble Sort**

b. **Insertion Sort**

c. **Selection Sort**

d. **Quick Sort**

e. **Merge Sort**

f. **Heap Sort**

1. *Bubble Sort Algorithm*

Bubble Sort is a simple algorithm which is used to sort a given set of **n** elements provided in form of an array with **n** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the **first element** of the array with the **second element**, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total **n** elements, then we need to repeat this process for **n-1** times.

It is known as bubble sort, because with every complete iteration the largest element in the given array, **bubbles up** towards the last place or the highest index, just like a water bubble rises up to the water surface. Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

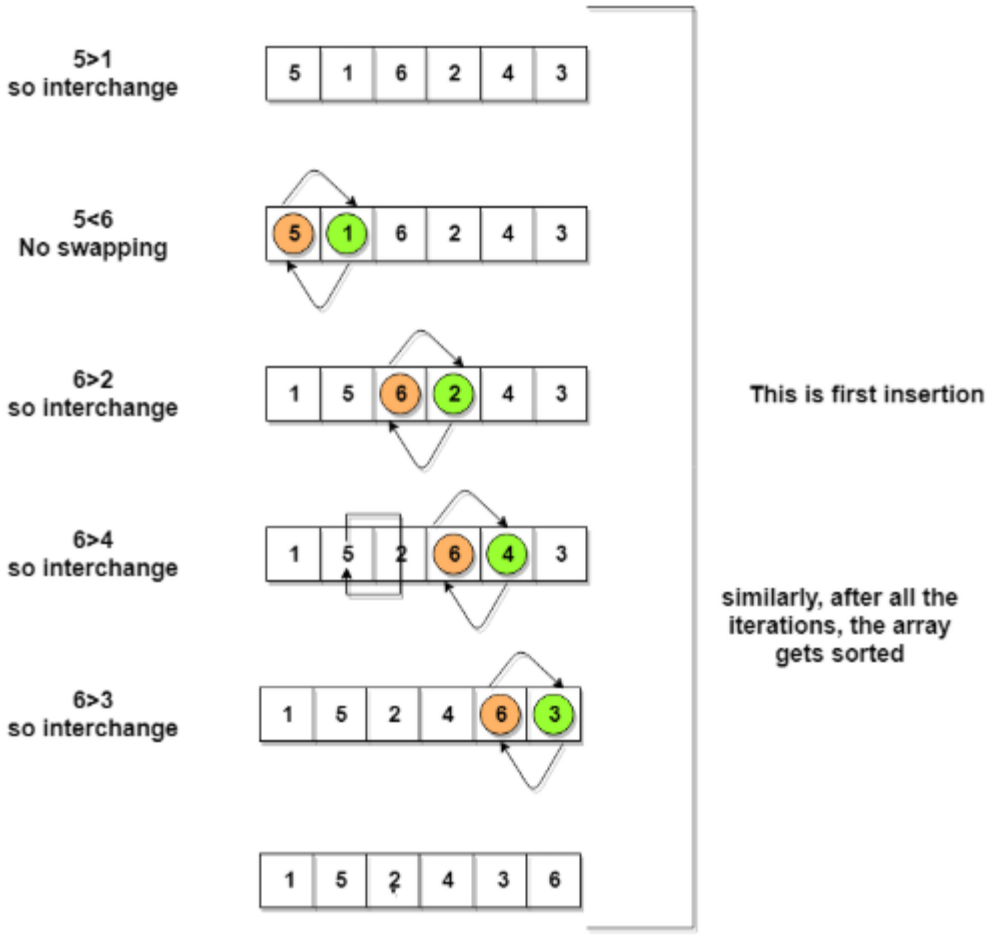
Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(**index = 0**), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. Repeat Step 1.

Let's consider an array with values **{5, 1, 6, 2, 4, 3}**

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, **6** is placed at the **last index**, which is the correct position for it. Similarly after the second iteration, 5 will be at the second last index, and so on.

Following are the Time and Space complexity for the Bubble Sort algorithm.

Worst Case Time Complexity [Big-O]: **O(n2)**

Best Case Time Complexity [Big-omega]: **O(n)**

Average Time Complexity [Big-theta]: **O(n2)**

Space Complexity: **O(1)**

2. ***Isertion Sort Algorithm***

Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers. If I give you another card, and ask you to insert the card in just the right position, so that the cards in your hand are still sorted. What will you do?

Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will insert the

card there.

Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

This is exactly how insertion sort works. It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left.

Following are some of the important **characteristics** of Insertion Sort:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a stable sorting technique, as it does not change the relative order of elements which are equal.

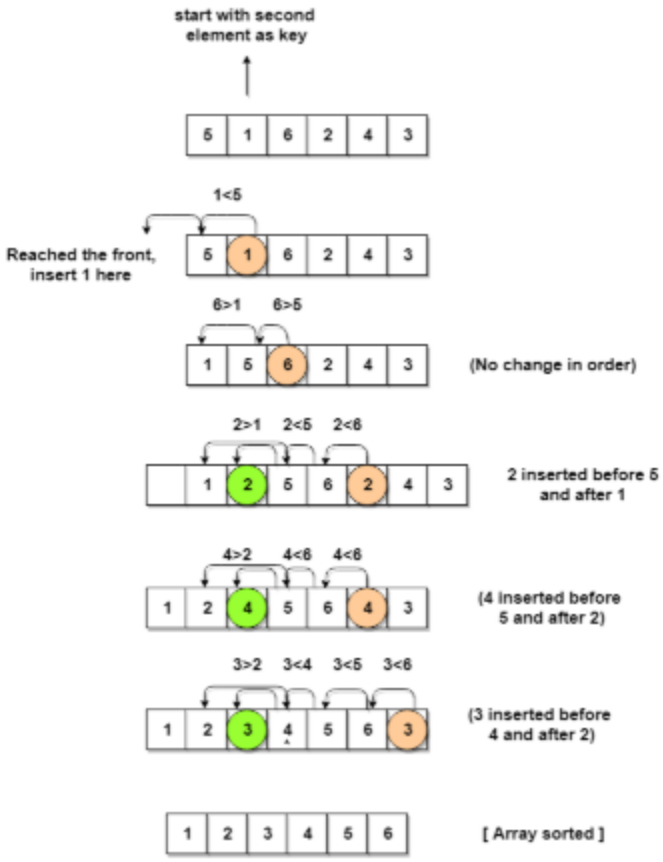
How Insertion Sort Works?

Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at **index 1**, the **key**. The **key** element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We compare the **key** element with the **element(s)** before it, in this case, element at **index 0**:
3. If the key element is less than the first element, we insert the **key** element before the first element.
4. If the **key** element is greater than the first element, then we insert it after the first element.
5. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.

And we go on repeating this, until the array is sorted.

Let's consider an array with values {5, 1, 6, 2, 4, 3}



Worst Case Time Complexity [Big-O]: **O(n2)**

Best Case Time Complexity [Big-omega]: **O(n)**

Average Time Complexity [Big-theta]: **O(n2)**

Space Complexity: **O(1)**

3. ***Selection Sort Algorithm***

Selection sort is conceptually the most simplest sorting algorithm. This algorithm will first find the smallest element in the array and swap it with the element in the **first position**, then it will find the **second smallest** element and **swap** it with the element in the second position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly selects the next-smallest element and swaps it into the right place.

How Selection Sort Works?

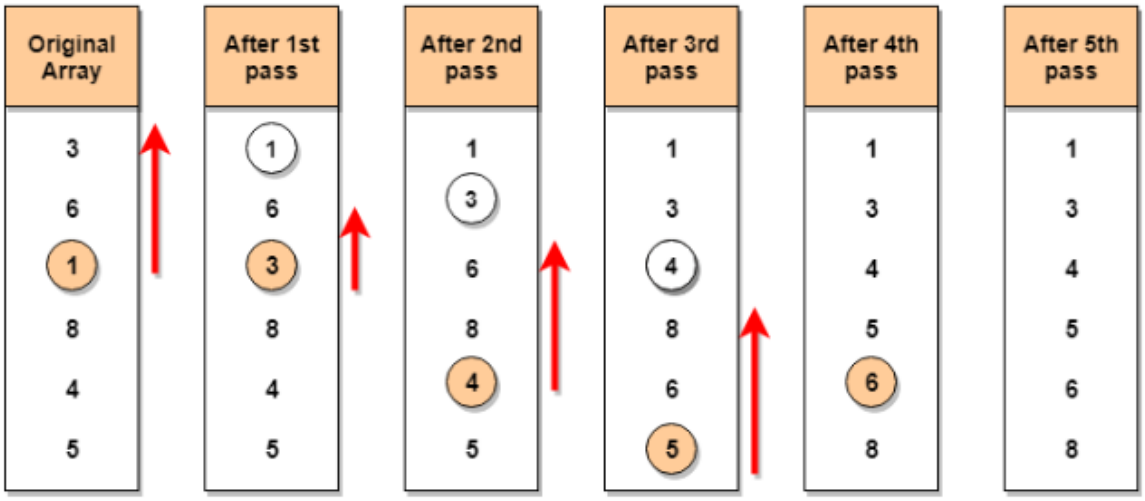
Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.

2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the second position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Let's consider an array with values {3, 6, 1, 8, 4, 5}

Below, we have a pictorial representation of how selection sort will sort the given array.



In the first pass, the **smallest element** will be **1**, so it will be placed at the first position.

Then leaving the first element, next **smallest element** will be searched, from the remaining elements. We will get **3** as the smallest, so it will be then placed at the second position.

Then leaving **1** and **3**(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

Worst Case Time Complexity [Big-O]: **O(n2)**

Best Case Time Complexity [Big-omega]: **O(n2)**

Average Time Complexity [Big-theta]: **O(n2)**

Space Complexity: **O(1)**

4. CombSort Algorithm

The basic idea of comb sort and the bubble sort is same. In other words, comb sort is an improvement on the bubble sort. In the bubble sorting technique, the items are compared with the next item in each phase. But for the comb sort, the items are sorted in a specific gap. After completing each phase, the gap is decreased. The decreasing factor or the shrink factor for this sort is 1.3. It means that after completing each phase the gap is divided by 1.3.

```
Begin
   gap := size
   flag := true
   while the gap ≠ 1 OR flag = true do
      gap = floor(gap/1.3) //the the floor value after division
      if gap < 1 then
         gap := 1
      flag = false
      for i := 0 to size – gap -1 do
         if array[i] > array[i+gap] then
            swap array[i] with array[i+gap]
            flag = true;
      done
   done
End
```

Time Complexity is O(n log n) for best case. O(n2/2nP) (p is number of increment) for average case and O(n2) for worst case.

Worst Case Time Complexity : **O(n2)**

Best Case Time Complexity : **O(nlogn)**

Average Time Complexity : **O(n2/2np)**

Space Complexity: **O(1)**

Exercise

Question # 01:

Will bubblesort() work properly if the inner loop

for (int j = n-1; j > i; --j)

is replaced by

for (int j = n-1; j > 0; --j)

What is the complexity of the new version?

Question # 02:

Cretically analyze Bubble sort and write improved version of bubble sort in CPP.

Question # 03:

Implement the insertion and selection sort in CPP. And compare the time complexities of best, average and worst case.