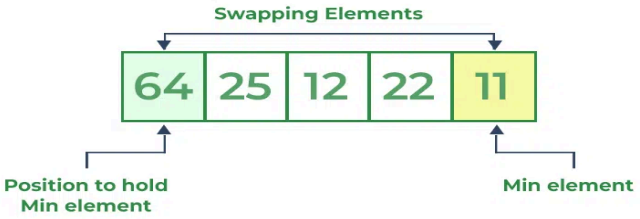
SORTING
Selection Sort

- Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.
- The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

Let's consider the following array as an example: **arr[] = {64, 25, 12, 22, 11}**

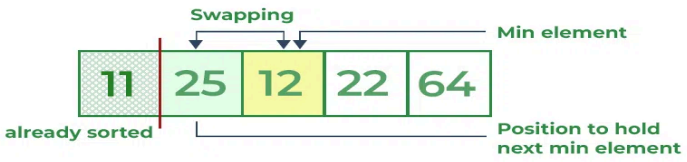
First pass:

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing whole array it is clear that **11** is the lowest value.
- Thus, replace 64 with 11. After one iteration **11**, which happens to be the least value in the array, tends to appear in the first position of the sorted list.



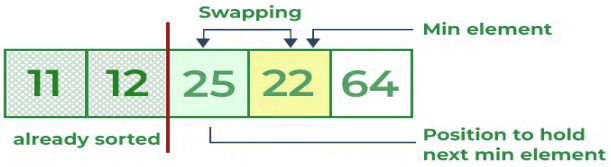
Second Pass:

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
- After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



Third Pass:

- Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.
- While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.



Fourth pass:

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As **25** is the 4th lowest value hence, it will place at the fourth position.

Fifth Pass:

- At last the largest value present in the array automatically get placed at the last position in the array
- The resulted array is the sorted array.

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is **$O(N^2)$** as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)
- Therefore overall complexity = O(N) * O(N) = O(N*N) = $O(N^2)$
- $O(N^2)$, (where N = size of the array), for the best, worst, and average cases.

Space Complexity: O(1)

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

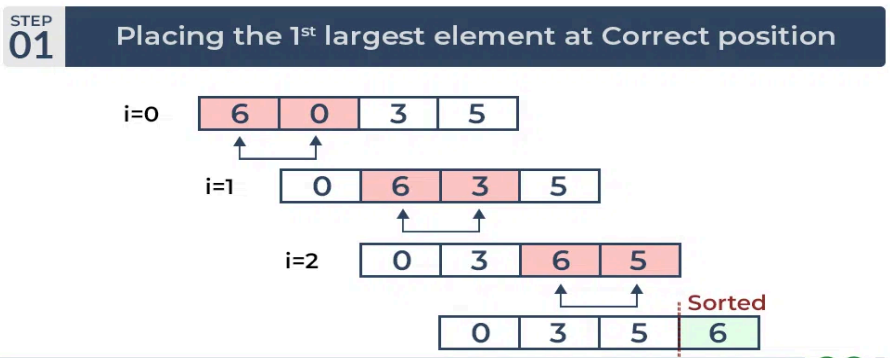
In Bubble Sort algorithm,

- traverse from the left and compare adjacent elements and the higher one is placed at the right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

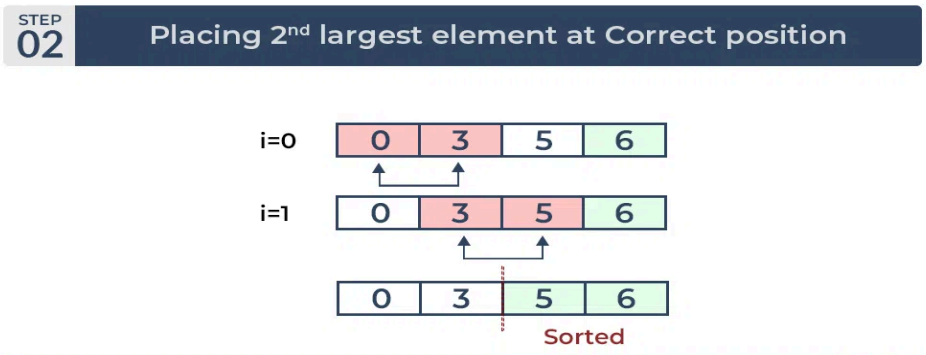
How does Bubble Sort Work?

Input: arr[] = {6, 3, 0, 5}

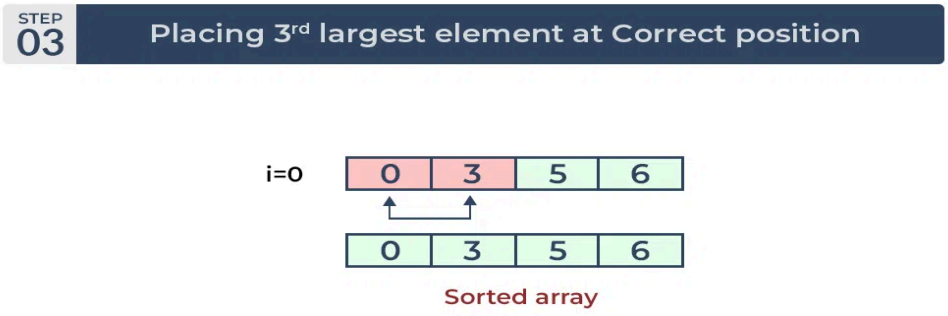
First Pass: The largest element is placed in its correct position, i.e., the end of the array.



Second Pass: Place the second largest element at correct position



Third Pass: Place the remaining two elements at their correct positions.



- **Total no. of passes:** n-1
- **Total no. of comparisons:** n*(n-1)/2
- **Time Complexity:** O(N^2) in the worst and average case.
  O(N) in the best case.
- **Auxiliary Space:** O(1)

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of O(N^2) which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

Insertion Sort

- **Insertion sort** is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.
- To sort an array of size N in ascending order, iterate over the array and compare the current element (key) to its predecessor, 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.

Working of Insertion Sort algorithm

Consider an example: arr[]: **{12, 11, 13, 5, 6}**

12 11 13 5 6

First Pass:

- Initially, the first two elements of the array are compared in insertion sort.

12 **11** 13 5 6

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11 **12** 13 5 6

Second Pass:

- Now, move to the next two elements and compare them

11 **12** **13** 5 6

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted subarray along with 11.

Third Pass:

- Now, two elements are present in the sorted sub-array which are **11** and **12**
- Moving forward to the next two elements which are 13 and 5

11 12 **13** **5** 6

- Both 5 and 13 are not present at their correct place so swap them

11 12 **5** **13** 6

- After swapping, elements 12 and 5 are not sorted, thus swap again

11 **5** **12** 13 6

- Here, again 11 and 5 are not sorted, hence swap again

5 **11** 12 13 6

- Here, 5 is at its correct position

Fourth Pass:

- Now, the elements which are present in the sorted sub-array are **5, 11** and **12**
- Moving to the next two elements 13 and 6

5 11 12 **13** **6**

- Clearly, they are not sorted, thus perform swap between both

5 11 12 **6** **13**

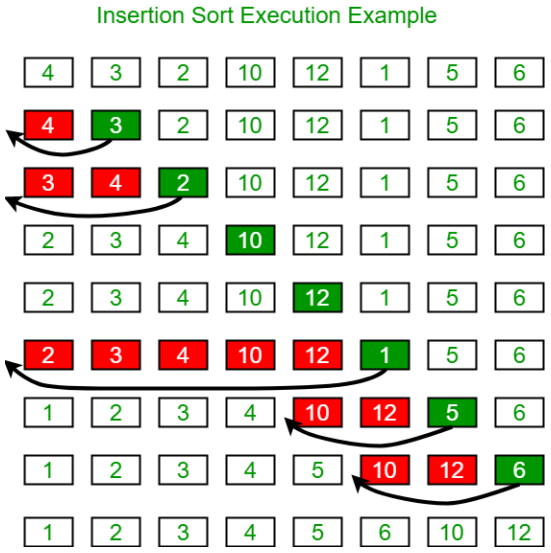
- Now, 6 is smaller than 12, hence, swap again

5 11 **6** **12** 13

- Here, also swapping makes 11 and 6 unsorted hence, swap again

5 **6** **11** 12 13

- Finally, the array is completely sorted.



Time Complexity of Insertion Sort

- The **worst-case** time complexity of the Insertion sort is **O(N^2)**
- The **average case** time complexity of the Insertion sort is **O(N^2)**
- The time complexity of the **best case** is **O(N)**.

Space Complexity of Insertion Sort

The auxiliary space complexity of Insertion Sort is **O(1)**

Characteristics of Insertion Sort

- This algorithm is one of the simplest algorithms with a simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted.

Best Case Time Complexity:

The best case occurs if the given array is already sorted. And if the given array is already sorted, the outer loop will only run and the inner loop will run for 0 times. So, our overall **time complexity in the best case will boil down to O(N),** where N = size of the array.