In **Bubble Sort,** sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required. With each interation the largest element in the given array, is shifted towards the last place or the highest index in the array.

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.

It includes two loops :

1. Outer loop decides the number of traversals the algorithm makes over the array i.e. n-1
2. Inner loop decides upto where the algorithm parses the array in each traversal i.e. n-i-1

In traversal n, the nth largest element is pushed to the right side of the array while comparing two elements at a time.

Bubble Sort makes n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

$(n-1) + (n-2) + (n-3) + ... + 1 = (n*(n+1))/2 = \mathbf{O(n^2)}$

Hence the **time complexity** of Bubble Sort is $\mathbf{O(n^2)}$.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. fortempvariable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

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

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **$O(n_2)$**
- Space Complexity: **O(1)**

Dry run of the algorithm :

arr = [3,5,2,4,0,1]

Traversal i = 0 :
Step j = 0 :

arr[j] = 3, arr[j+1] = 5
arr[j]>arr[j+1] -> False
Not Swapped!
Final Array = 3,5,2,4,0,1

Step j = 1 :

arr[j] = 5, arr[j+1] = 2
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 3,2,5,4,0,1

Step j = 2 :

arr[j] = 5, arr[j+1] = 4
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 3,2,4,5,0,1

Step j = 3 :

arr[j] = 5, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 3,2,4,0,5,1

Step j = 4 :

arr[j] = 5, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 3,2,4,0,1,5

Traversal i = 1 :
Step j = 0 :

arr[j] = 3 , arr[j+1] = 2
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 2,3,4,0,1,5

Step j = 1 :

arr[j] = 3, arr[j+1] = 4
arr[j]>arr[j+1] -> False
Not Swapped!
Final Array = 2,3,4,0,1,5

Step j = 2 :

arr[j] = 4, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 2,3,0,4,1,5

Step j = 3 :

arr[j] = 4, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 2,3,0,1,4,5

Traversal i = 2 :

Step j = 0 :

arr[j] = 2, arr[j+1] = 3
arr[j]>arr[j+1] -> False
Not Swapped!
Final Array = 2,3,0,1,4,5

Step j = 1 :

arr[j] = 3, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 2,0,3,1,4,5

Step j = 2 :

arr[j] = 3, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 2,0,1,3,4,5

Traversal i = 3:

Step j = 0 :

arr[j] = 2, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 0,2,1,3,4,5

Step j = 1 :

arr[j] = 2, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Final Array = 0,1,2,3,4,5

Traversal i = 4 :

Step j = 0 :

arr[j] = 0, arr[j+1] = 1
arr[j]>arr[j+1] -> False
Not Swapped!
Final Array = 0,1,2,3,4,5

Traversal i = 5 :

Condition for inner loop is not satisfied and hence it will break the loop and return the sorted array

In **Optimized Bubble Sort,** sorting takes place in the same way as in Bubble Sort except the fact that if no swaps take place in any iteration the outer loop breaks and the sorted array is returned.

In the above example itself, if we add the flag variable **swapped** it will be **false** after the 4^{th} iteration and sorted array will be returned after four interations only.

Dry run of the algorithm after optimization :

arr = [3,5,2,4,0,1]

Traversal i = 0 :
Step j = 0 :

arr[j] = 3, arr[j+1] = 5
arr[j]>arr[j+1] -> False
Not Swapped!
Swapped = False
Final Array = 3,5,2,4,0,1

Step j = 1 :

arr[j] = 5, arr[j+1] = 2
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 3,2,5,4,0,1

Step j = 2 :

arr[j] = 5, arr[j+1] = 4
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 3,2,4,5,0,1

Step j = 3 :

arr[j] = 5, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 3,2,4,0,5,1

Step j = 4 :

arr[j] = 5, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 3,2,4,0,1,5

Traversal i = 1 :
Step j = 0 :

arr[j] = 3 , arr[j+1] = 2
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 2,3,4,0,1,5

Step j = 1 :

arr[j] = 3, arr[j+1] = 4
arr[j]>arr[j+1] -> False
Not Swapped!
Swapped = True
Final Array = 2,3,4,0,1,5

Step j = 2 :

arr[j] = 4, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 2,3,0,4,1,5

Step j = 3 :

arr[j] = 4, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 2,3,0,1,4,5

Traversal i = 2 :

Step j = 0 :

arr[j] = 2, arr[j+1] = 3
arr[j]>arr[j+1] -> False
Not Swapped!
Swapped = False
Final Array = 2,3,0,1,4,5

Step j = 1 :

arr[j] = 3, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 2,0,3,1,4,5

Step j = 2 :

arr[j] = 3, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 2,0,1,3,4,5

Traversal i = 3:

```
Step j = 0 :

arr[j] = 2, arr[j+1] = 0
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 0,2,1,3,4,5

Step j = 1 :

arr[j] = 2, arr[j+1] = 1
arr[j]>arr[j+1] -> True
Swapped!
Swapped = True
Final Array = 0,1,2,3,4,5
```

Traversal i = 4 :

```
Step j = 0 :

arr[j] = 0, arr[j+1] = 1
arr[j]>arr[j+1] -> False
Not Swapped!
Swapped = False
Final Array = 0,1,2,3,4,5

Since Swapped is False

Break the outer loop

Sorted Array = 0,1,2,3,4,5
```