RIPHAH INTERNATIONAL UNIVERSITY

Name: Muhammad Fahad

Sap ID: 37125

Section: BSCS-3B

Course: DSA

Assignment#1

Instructor: Noor Ullah Khan

What Is Bubble Sort?

Works by swapping adjacent elements in repeated passes, if they are not correct order. High time complexity and not suitable for large datasets.

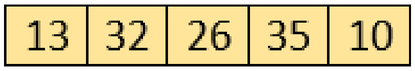
Time Complexity: $O(n^2)$

Working of Bubble sort Algorithm:

Now, let's see the working of Bubble sort Algorithm.

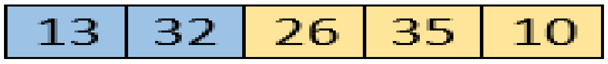
To understand the working of bubble sort algorithm, let's take an unsorted array. We are taking a short and accurate array, as we know the complexity of bubble sort is **$O(n^2)$.**

Let the elements of array are -

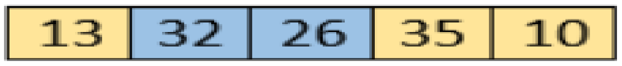


<u>1ST Pass</u>

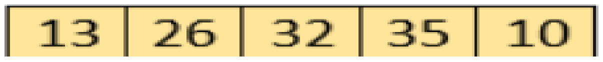
Sorting will start from the initial two elements. Let compare them to check which is greater.



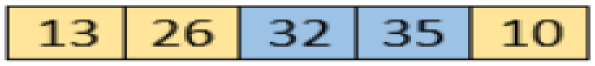
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like –

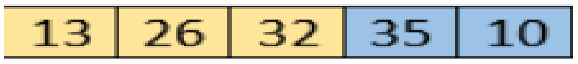


Now, compare 32 and 35.

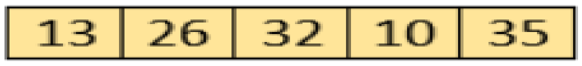


Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.

Now, the comparison will be in between 35 and 10.



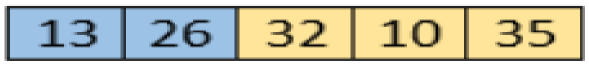
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be –

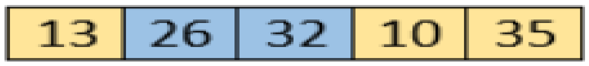


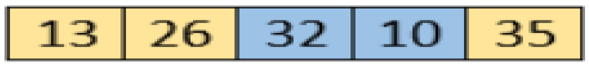
Now, move to the second iteration.

2nd Pass:

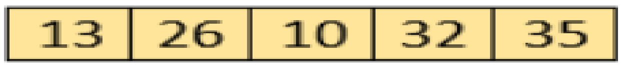
The same process will be followed for second iteration.

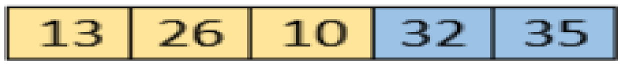






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

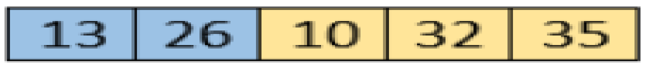


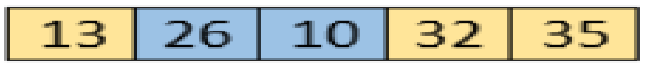


Now, move to the third iteration.

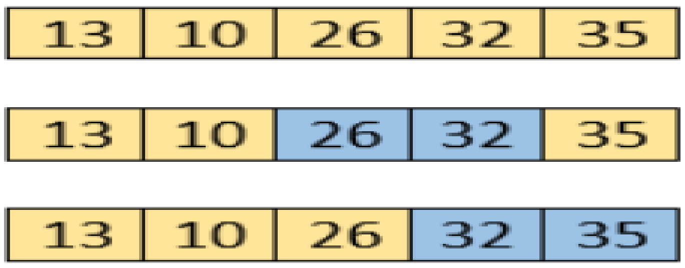
3rd Pass

The same process will be followed for third iteration.





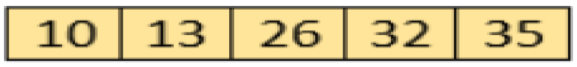
Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be –



Now, move to the fourth iteration.

<u>4th Pass</u>

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is **O(n).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is **$O(n^2)$.**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is **$O(n^2)$.**

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of bubble sort is **O(1).** It is because, in bubble sort, an extra variable is required for swapping.
- The space complexity of optimized bubble sort is **O(2).** It is because two extra variables are required in optimized bubble sort.

Optimized Bubble sort Algorithm

In the bubble sort algorithm, comparisons are made even when the array is already sorted. Because of that, the execution time increases.

To solve it, we can use an extra variable swapped. It is set to true if swapping requires; otherwise, it is set to false.

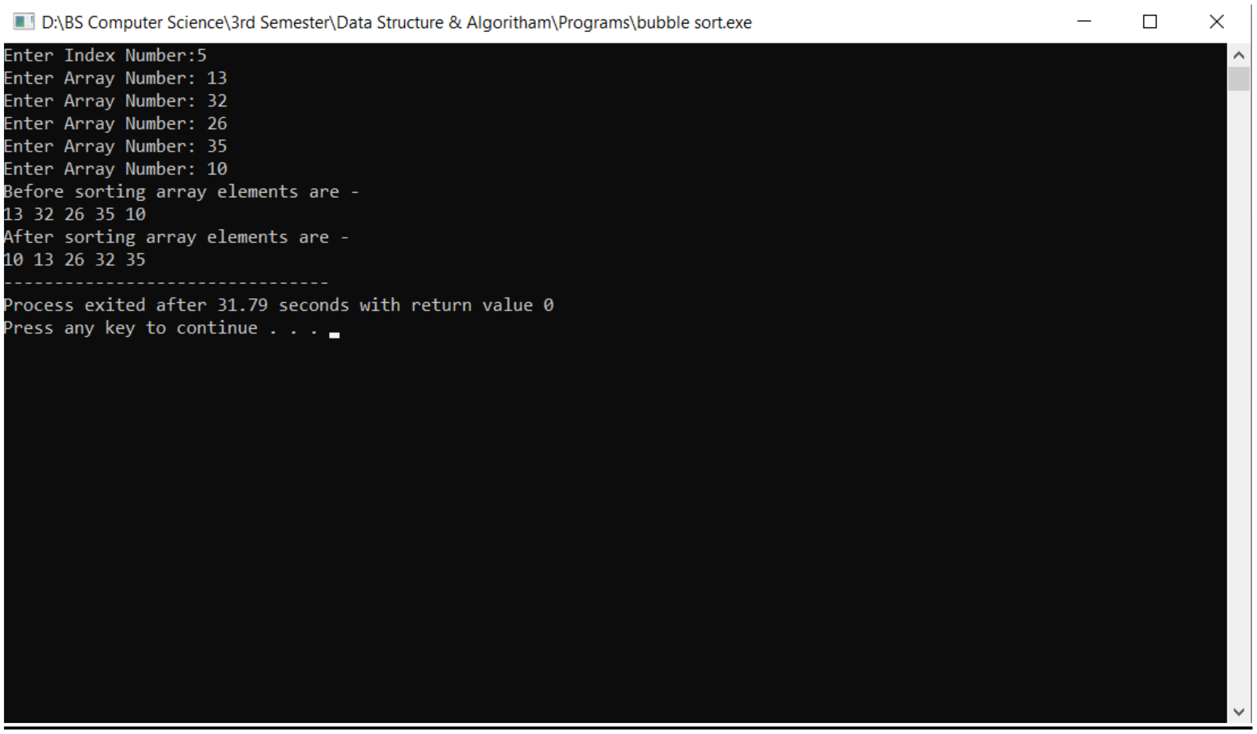
It will be helpful, as suppose after an iteration, if there is no swapping required, the value of variable swapped will be false**.** It means that the elements are already sorted, and no further iterations are required.

Code:

```
#include<iostream>

using namespace std;


class BubbleArray {

private:


    int i;

    int j;

    int temp;

public:

    int* a;

    int n;


    BubbleArray()

    {

        cout << "Enter Index Number:";
    cin >> n;

    a = new int[n];




}





void bubble(int a[],int n)

{


   cout << "Before sorting array elements are - \n";

    for (i = 0; i < n; i++)

    {

        cout << a[i] << " ";

    }

    for (i = 0; i < n; i++)

    {

        for (j = i + 1; j < n; j++)

        {

            if (a[j] < a[i])

            {

                temp = a[i];

                a[i] = a[j];

                a[j] = temp;

            }
            }
        }


         //Display Function
         cout << "\nAfter sorting array elements are - \n";
          for (i = 0; i < n; i++)
        {
            cout << a[i] << " ";
        }

    }




};



int main()
{
    BubbleArray b;
    for (int i = 0; i < b.n; i++)
        {
            cout << "Enter Array Number: ";
            cin >> b.a[i];
        }
```

```
//    int n = sizeof(b.a)/sizeof(b.a[0]);

    b.bubble(b.a,b.n);

    return 0;
}
```

Output



D:\BS Computer Science\3rd Semester\Data Structure & Algoritham\Programs\bubble sort.exe

```
Enter Index Number:5
Enter Array Number: 13
Enter Array Number: 32
Enter Array Number: 26
Enter Array Number: 35
Enter Array Number: 10
Before sorting array elements are -
13 32 26 35 10
After sorting array elements are -
10 13 26 32 35
--------------------------------
Process exited after 31.79 seconds with return value 0
Press any key to continue . . .
```

What Is Insertion Sort?

The array is split into sorted and unsorted parts. Unsorted elements are picked and placed at their correct position in the sorted part

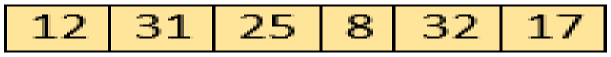
Time Complexity: **O(n^2)**

Working of Bubble sort Algorithm:

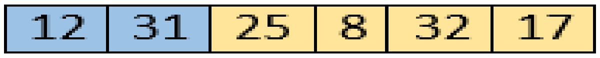
Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

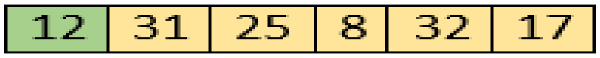
Let the elements of array are -



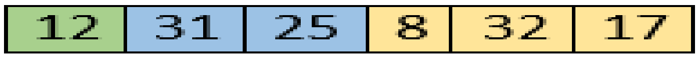
Initially, the first two elements are compared in insertion sort.

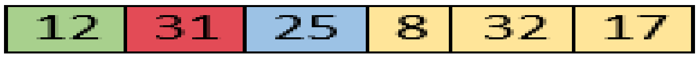


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



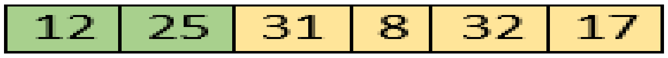
Now, move to the next two elements and compare them.



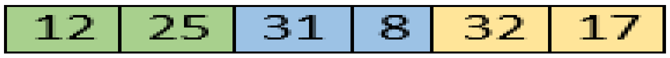


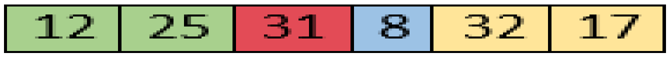
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

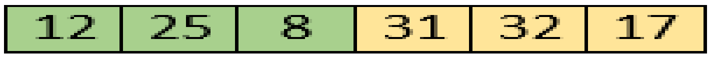


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

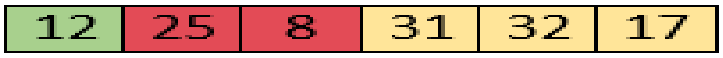




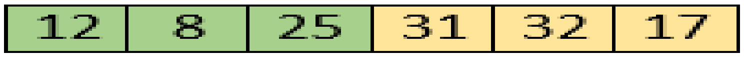
Both 31 and 8 are not sorted. So, swap them.



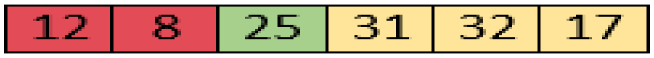
After swapping, elements 25 and 8 are unsorted.



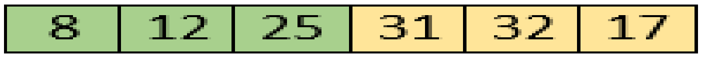
So, swap them.



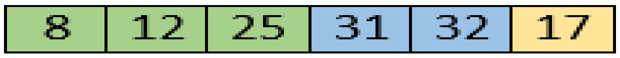
Now, elements 12 and 8 are unsorted.



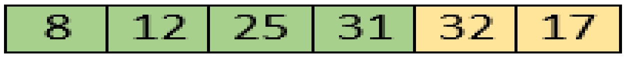
So, swap them too.



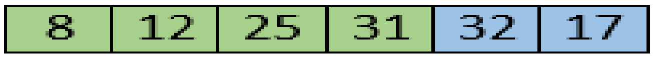
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



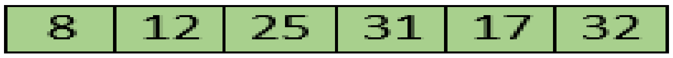
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.

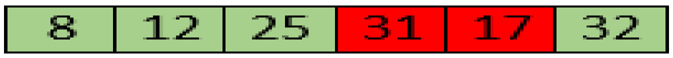


Move to the next elements that are 32 and 17.

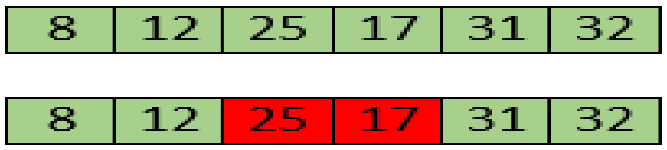


17 is smaller than 32. So, swap them.

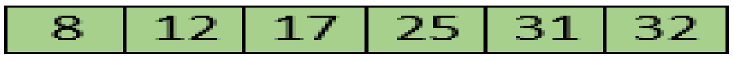




Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Insertion sort complexity

Now, let's see the time complexity of insertion sort in best case, average case, and in worst case. We will also see the space complexity of insertion sort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is **O(n)**.

- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is **O(n^2)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **O(n^2)**.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

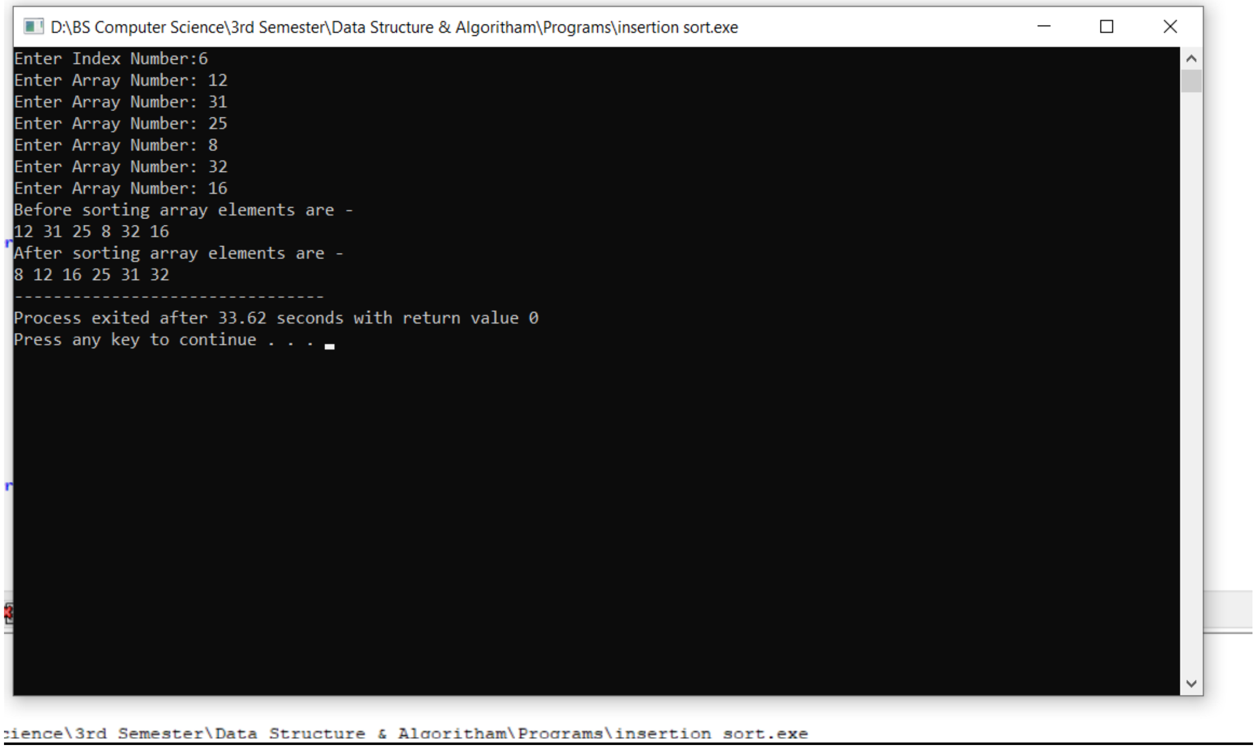
- The space complexity of insertion sort is O(1). It is because, in insertion sort, an extra variable is required for swapping.

<u>Code:</u>

```
#include<iostream>
using namespace std;

class InsertSort {
private:

    int i;
    int j;
    int temp;
```

```
public:
    int* a;
    int n;

    InsertSort()
    {
        cout << "Enter Index Number:";
        cin >> n;
        a = new int[n];



    }




    void Insert(int a[], int n)
    {

        cout << "Before sorting array elements are - \n";
        for (i = 0; i < n; i++)
        {
            cout << a[i] << " ";
        }
        for (i = 1; i < n; i++) {
            temp = a[i];
            j = i - 1;
            while ( temp <= a[j] && j >= 0 )
            {
                a[j + 1] = a[j];
                j = j - 1;
            }
            a[j + 1] = temp;
        }




        //Display Function
        cout << "\nAfter sorting array elements are - \n";
        for (i = 0; i < n; i++)
        {
            cout << a[i] << " ";
        }
    }
};


int main()
{
    InsertSort  b;
    for (int i = 0; i < b.n; i++)
    {
        cout << "Enter Array Number: ";
        cin >> b.a[i];
    }
    //    int n = sizeof(b.a)/sizeof(b.a[0]);

    b.Insert(b.a, b.n);

    return 0;
}
```

Output:



D:\BS Computer Science\3rd Semester\Data Structure & Algoritham\Programs\insertion sort.exe

```
Enter Index Number:6
Enter Array Number: 12
Enter Array Number: 31
Enter Array Number: 25
Enter Array Number: 8
Enter Array Number: 32
Enter Array Number: 16
Before sorting array elements are -
12 31 25 8 32 16
After sorting array elements are -
8 12 16 25 31 32
--------------------------------
Process exited after 33.62 seconds with return value 0
Press any key to continue . . .
```

```
cience\3rd Semester\Data Structure & Algoritham\Programs\insertion sort.exe
```