LABORATORY REPORT

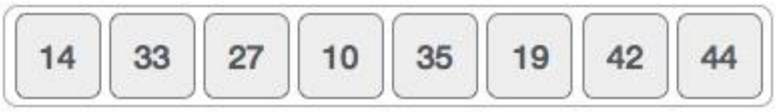
1.

Objective: Write a C program to demonstrating selection sort

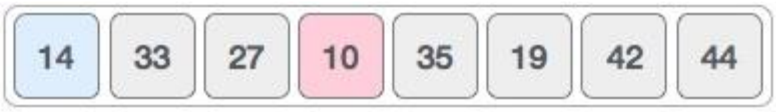
Software: Online Compiler for C and C++ (IDE)C and CPP compiler

Methodology:

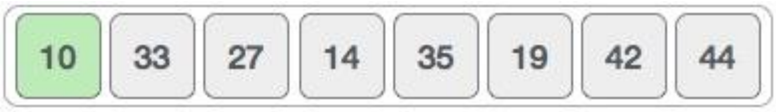
Consider the following depicted array as an example.



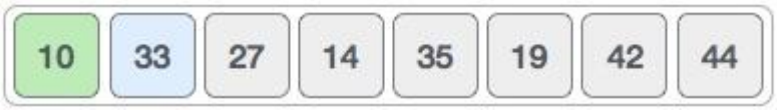
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



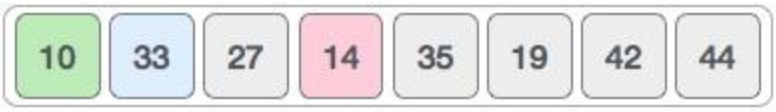
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



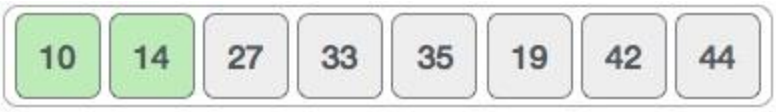
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

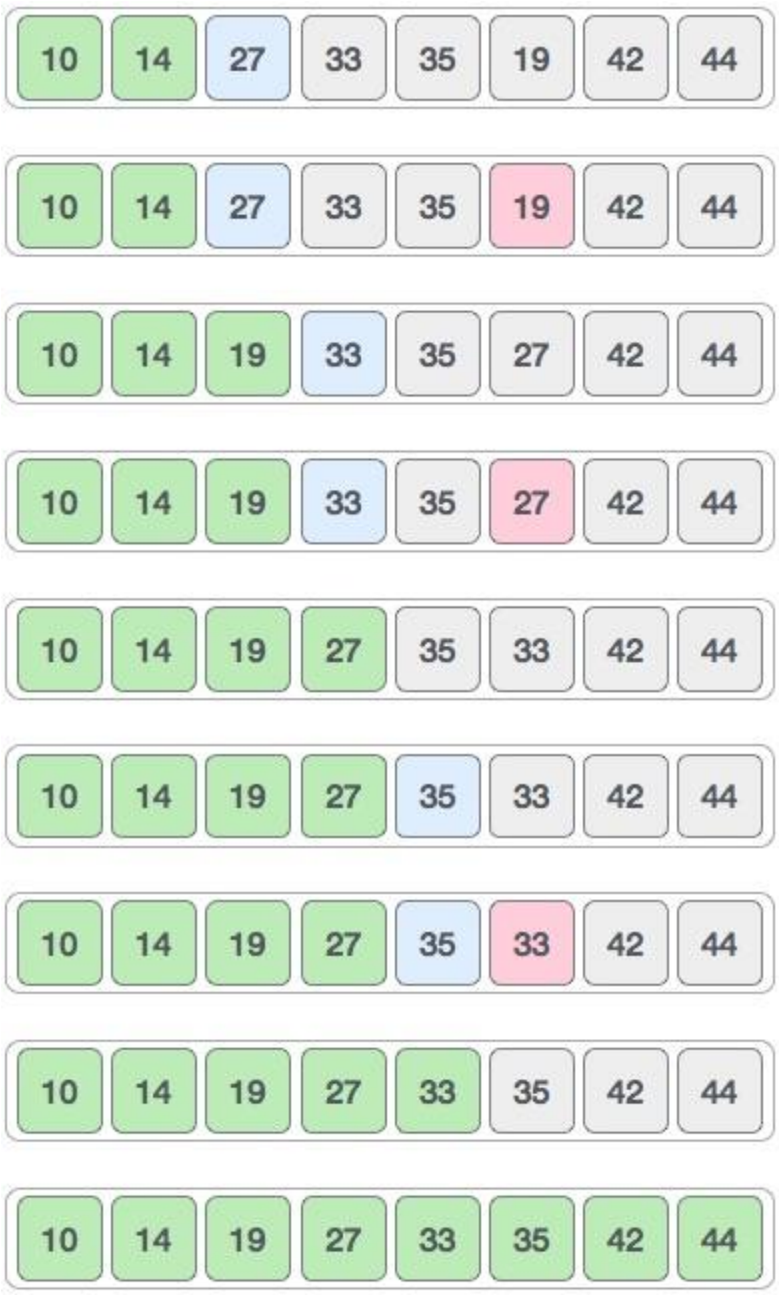


After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.

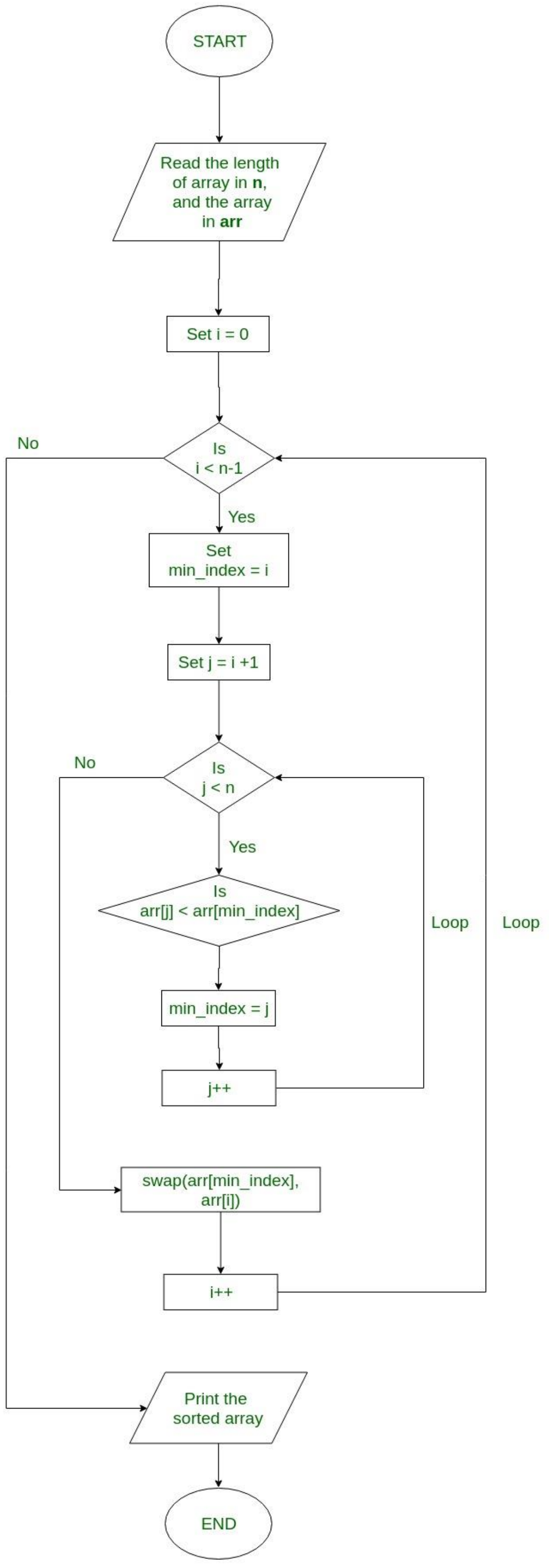
Following is a pictorial depiction of the entire sorting process −



Algorithm:

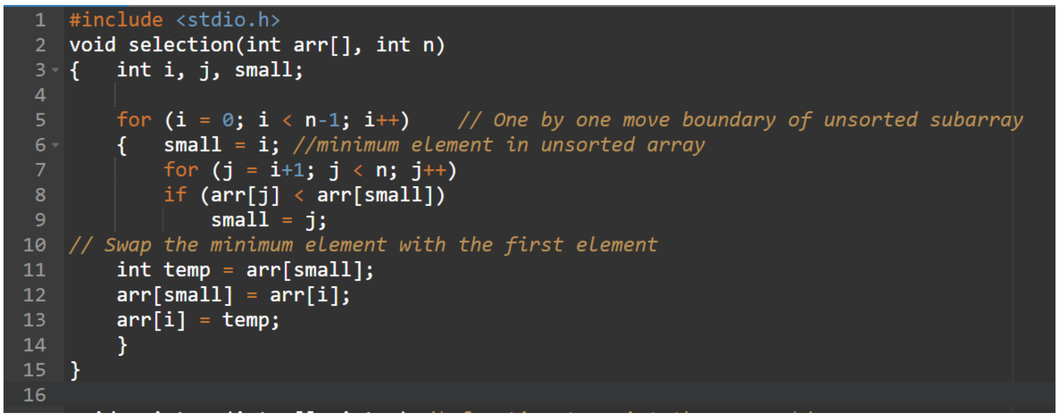
1. Start
2. Set MIN to location 0
3. Search the minimum element in the list
4. Swap with value at location MIN
5. Increment MIN to point to next element
6. Repeat until list is sorted
7. stop

Flowchart:



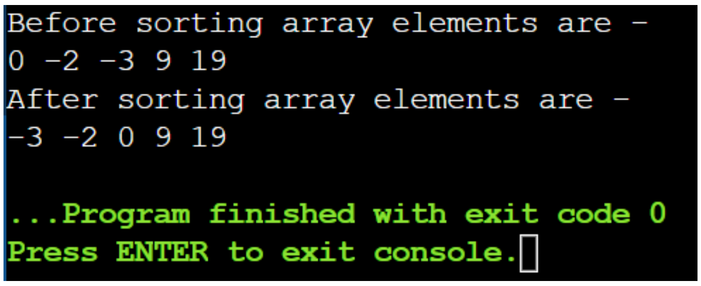
Flowchart for Selection Sort

Code:

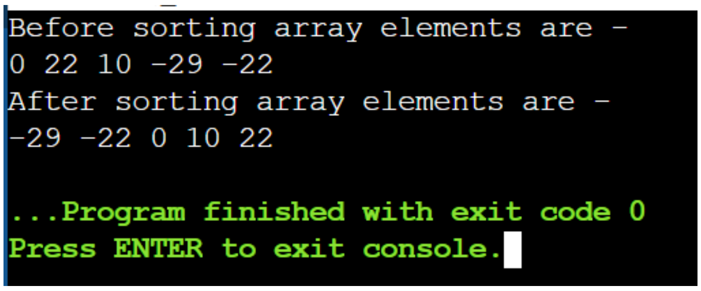


```c
#include <stdio.h>
void selection(int arr[], int n)
{   int i, j, small;

    for (i = 0; i < n-1; i++)    // One by one move boundary of unsorted subarray
    {   small = i; //minimum element in unsorted array
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[small])
            small = j;
// Swap the minimum element with the first element
    int temp = arr[small];
    arr[small] = arr[i];
    arr[i] = temp;
    }
}
```

Results:



```
Before sorting array elements are -
0 -2 -3 9 19
After sorting array elements are -
-3 -2 0 9 19

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Before sorting array elements are -
0 22 10 -29 -22
After sorting array elements are -
-29 -22 0 10 22

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: The C code has been executed successfully and the desired results are obtained.

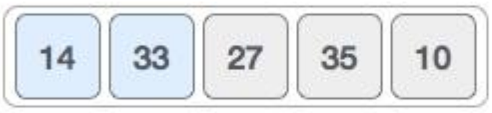
2.

Objective: Write C program to demonstrate bubble sort

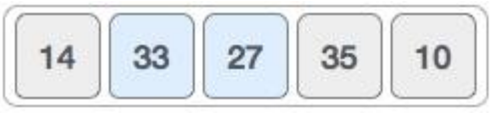
Software: Online Compiler for C and C++ (IDE)C and CPP compiler

Methodology :

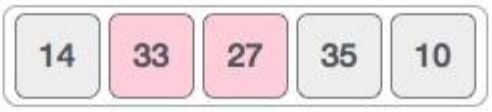
Bubble sort starts with very first two elements, comparing them to check which one is greater.



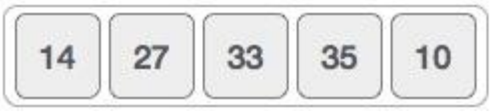
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



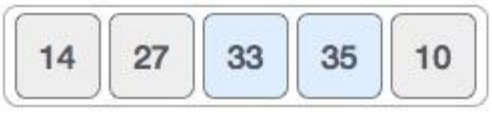
We find that 27 is smaller than 33 and these two values must be swapped.



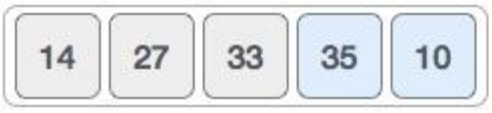
The new array should look like this −



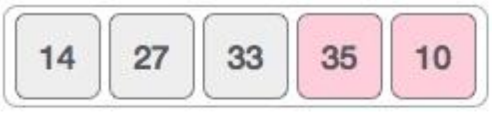
Next we compare 33 and 35. We find that both are in already sorted positions.



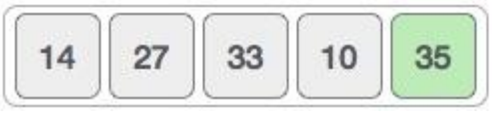
Then we move to the next two values, 35 and 10.



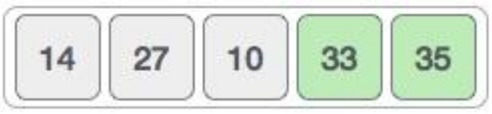
We know then that 10 is smaller 35. Hence they are not sorted.



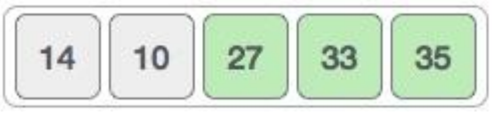
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −



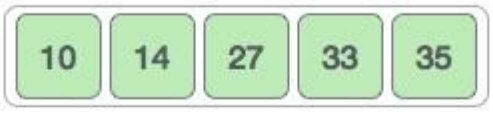
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.

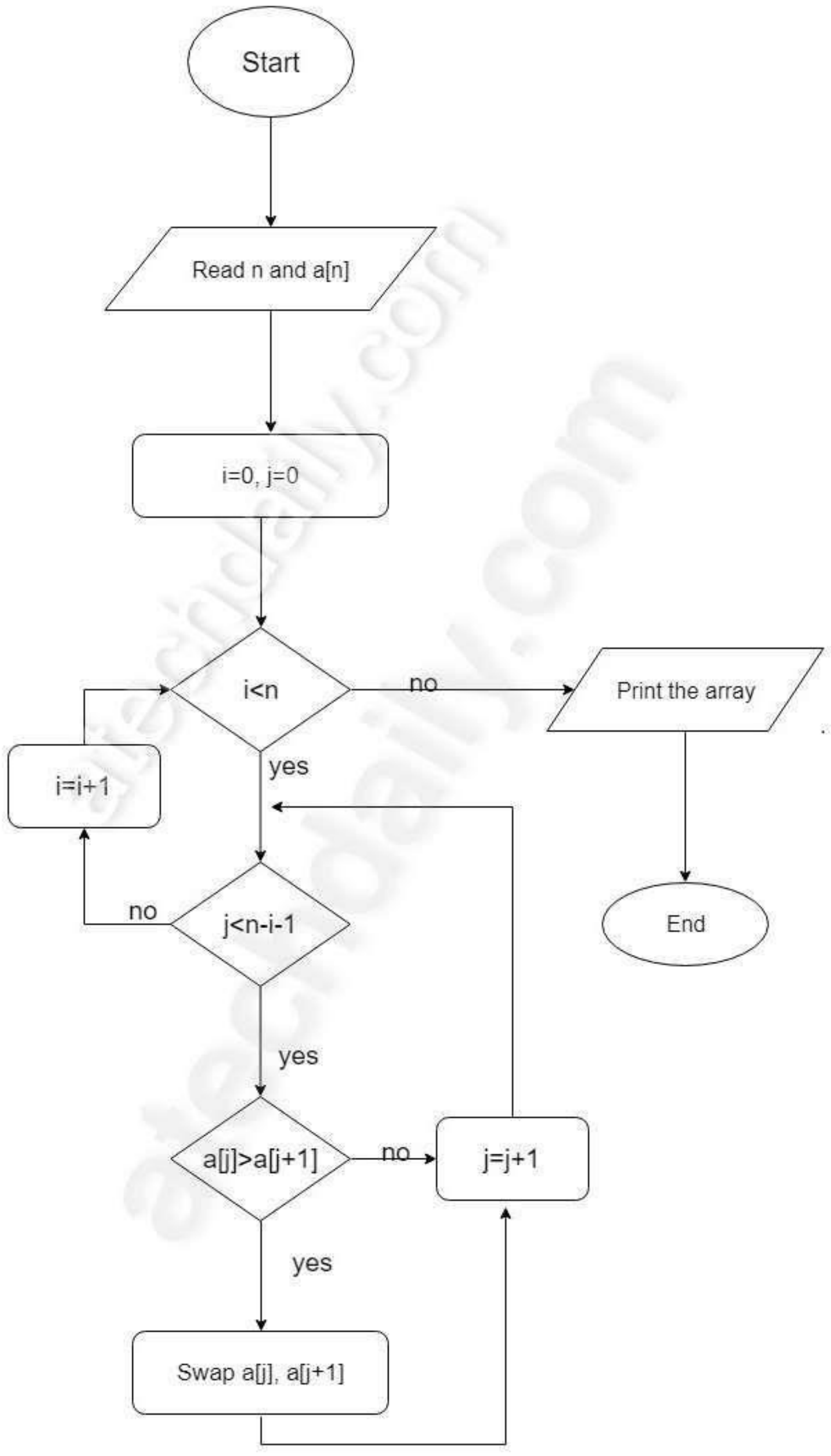


Now we should look into some practical aspects of bubble sort.

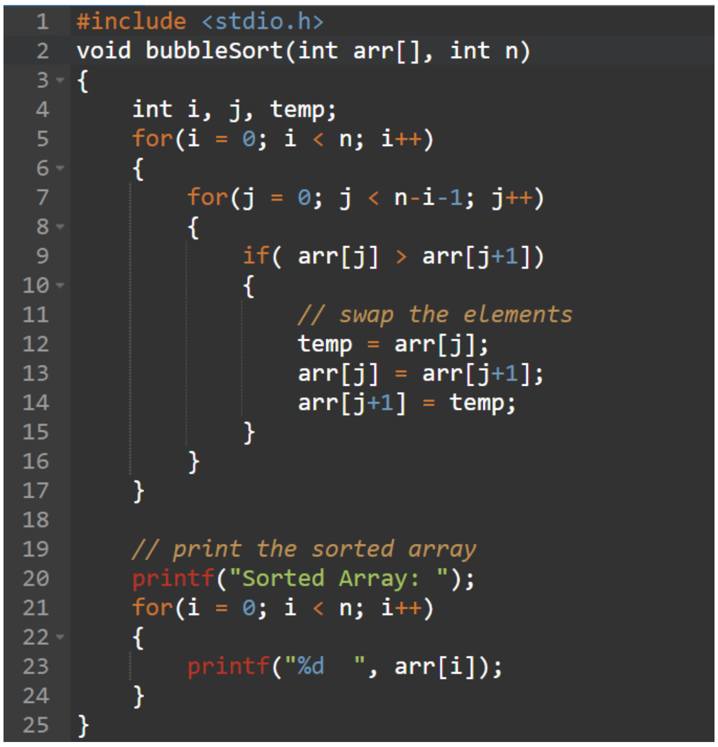
Algorithm:

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

Flowchart:

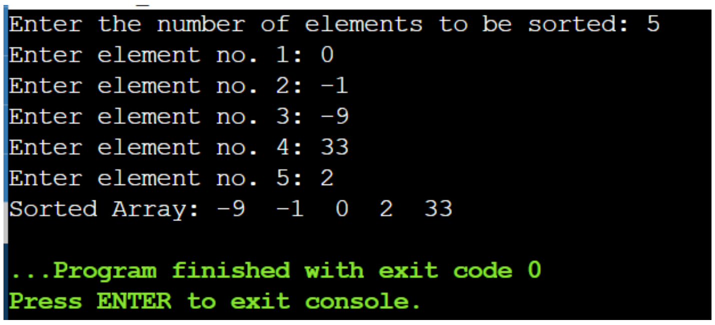

Start
Read n and a[n]
i=0, j=0
i<n
no
Print the array
yes
i=i+1
End
no
j<n-i-1
yes
a[j]>a[j+1]
no
j=j+1
yes
Swap a[j], a[j+1]

Code:

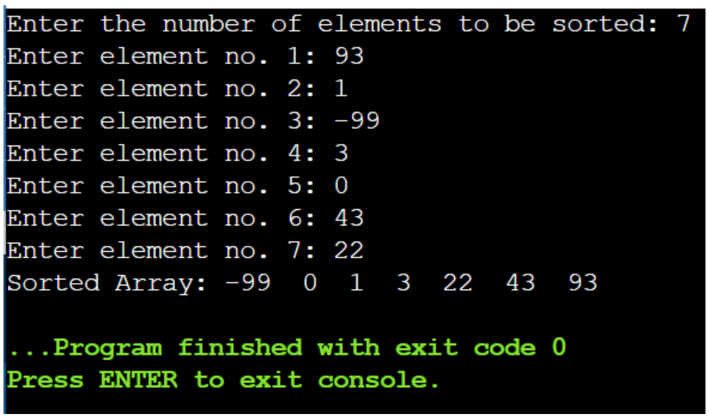


```c
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}
```

Results :



```
Enter the number of elements to be sorted: 5
Enter element no. 1: 0
Enter element no. 2: -1
Enter element no. 3: -9
Enter element no. 4: 33
Enter element no. 5: 2
Sorted Array: -9  -1  0  2  33

...Program finished with exit code 0
Press ENTER to exit console.
```



```
Enter the number of elements to be sorted: 7
Enter element no. 1: 93
Enter element no. 2: 1
Enter element no. 3: -99
Enter element no. 4: 3
Enter element no. 5: 0
Enter element no. 6: 43
Enter element no. 7: 22
Sorted Array: -99  0  1  3  22  43  93

...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: The C code has been executed successfully and the desired results are obtained.

3. Comparison of computational complexity of selection sort and bubble sort

Time Complexity Analysis of Selection Sort

At the beginning, the size of sorted sub-array (say S1) is 0 and the size of unsorted sub-array (say S2) is N.

At each step, the size of sorted sub-array increases by 1 and size of unsorted sub-array decreases by 1. Hence, for a few steps are as follows:

- Step 1: S1: 0, S2: N
- Step 2: S1: 1, S2: N-1
- Step 3: S1: 2, S2: N-2

  and so on till S1 = N. Hence, there will be N+1 steps.

Hence, S2 = N - S1

The Time Complexity of finding the smallest element in a list of M elements is O(M). This is constant for all worst case, average case and best case.

The time required for finding the smallest element is the size of unsorted sub-array that is O(S2). The exact value of S2 is dependent of the step.

For step I, S1 will be I-1 and S2 will be N-S1 = N-I+1.
So, the time complexity for step I will be:

- O(N-I+1) for find the smallest element
- O(1) for swapping the smallest element to the front of unsorted sub-array

I will range from 1 to N+1.

Hence, the sum of time complexity of all operations will be as follows:

Sum [O(N-I+1) + O(1)] for I from 1 to N+1
= Sum [O(N-I+1)] + Sum[O(1)] ... Equation 1

Sum [O(1)] = 1 + 1 + ... + 1 [(N+1) times] = N+1 = O(N)

Sum [O(N-I+1)] = N + (N-1) + ... + 1 + 0 =
= 1 + 2 + ... + N
= N * (N+1) / 2
= (N^2 + N) / 2 = O(N^2) + O(N) = O(N^2) [as N^2 is dominant term]

Therefore, from Equation 1, we get:
Sum [O(N-I+1)] + Sum[O(1)]
= O(N^2) + O(N)
= O(N^2)

Hence, the time complexity of Selection Sort is O(N^2).

Worst Case Time Complexity of Selection Sort

The worst case is the case when the array is already sorted (with one swap) but the smallest element is the last element. For example, if the sorted number as a1, a2, ..., aN, then:

a2, a3, ..., aN, a1 will be the worst case for our particular implementation of Selection Sort.

The cost in this case is that at each step, a swap is done. This is because the smallest element will always be the last element and the swapped element which is kept at the end will be the second smallest element that is the smallest element of the new unsorted sub-array. Hence, the worst case has:

- N * (N+1) / 2 comparisons
- N swaps

Hence, the time complexity is O(N^2).

Best Case Time Complexity of Selection Sort

The best case is the case when the array is already sorted. For example, if the sorted number as a1, a2, ..., aN, then:

a1, a2, a3, ..., aN will be the best case for our particular implementation of Selection Sort.

This is the best case as we can avoid the swap at each step but the time spend to find the smallest element is still O(N). Hence, the best case has:

- N * (N+1) / 2 comparisons
- 0 swaps

Note only the number of swaps has changed. Hence, the time complexity is O(N^2).

Average Case Time Complexity of Selection Sort

Based on the worst case and best case, we know that the number of comparisons will be the same for every case and hence, for average case as well, the number of comparisons will be constant.

Number of comparisons = N * (N+1) / 2

Therefore, the time complexity will be O(N^2).

To find the number of swaps,

- There are N! different combination of N elements
- Only for one combination (sorted order) there is 0 swaps.
- In the worst case, a combination will have N swaps. There are several such combinations.
- Number of ways to select 2 elements to swap = nC2 = N * (N-1) / 2
- From sorted array, this will result in O(N^2) combinations which need 1 swap.

So,
0 swap = 1 combination
1 swap = O(N^2) combinations
2 swap = O(N^4) combinations
...
N swaps = O(N) combinations

Hence, the total number of swaps will be:

Hence, the average number of swaps will be N that is O((N+1)!) / O(N!). Hence, the average case has:

- N * (N+1) / 2 comparisons
- N swaps

Space Complexity of Selection Sort

The space complexity of Selection Sort is O(1).

This is because we use only constant extra space such as:

- 2 variables to enable swapping of elements.

- One variable to keep track of smallest element in unsorted array.

Hence, in terms of Space Complexity, Selection Sort is optimal as the memory requirements remain same for every input.

Conclusion

- Worst Case Time Complexity is: $O(N^2)$
- Average Case Time Complexity is: $O(N^2)$
- Best Case Time Complexity is: $O(N^2)$
- Space Complexity: $O(1)$

Time Complexity Analysis of Bubble sort

In this version the run time complexity is `Θ(N^2)`. This applies to all the cases including the worst, best and average cases because even if the array is already sorted the algorithm doesn't check that at any point and runs through all iterations. Although the number of swaps would differ in each case.
The number of times the inner loop runs

$=\sum N-i-1j=11(2)(2)=\sum j=1N-i-11$
$=(N-i)(1)(1)=(N-i)$
The number of times the outer loop runs
$=\sum N-1i=1\sum N-i-1j=11=\sum N-1i=1N-i(3)(3)=\sum i=1N-1\sum j=1N-i-11=\sum i=1N-1N-i$
$=N*(N-1)2(4)(4)=N*(N-1)2$

Worst Case Time Complexity

`Θ(N^2)` is the Worst Case Time Complexity of Bubble Sort.
This is the case when the array is **reversely sort** i.e. in descending order but we require ascending order or ascending order when descending order is needed.
The number of swaps of two elements is equal to the number of comparisons in this case as every element is out of place.

$T(N)=C(N)=S(N)=N*(N-1)2T(N)=C(N)=S(N)=N*(N-1)2$, from equation 2 and 4
Therefore, in the worst case:

- Number of Comparisons: O(N^2) time
- Number of swaps: O(N^2) time

Best Case Time Complexity

`Θ(N)` is the Best Case Time Complexity of Bubble Sort.
This case occurs when the given array is **already sorted**.

For the algorithm to realise this, only one walk through of the array is required during which no swaps occur (lines 9-13) and the swapped variable (false) indicates that the array is already sorted.

T(N)=C(N)=NT(N)=C(N)=N
S(N)=0S(N)=0
Therefore, in the best case:

- Number of Comparisons: N = O(N) time
- Number of swaps: 0 = O(1) time

Average Case Time Complexity

`Θ(N^2)` is the Average Case Time Complexity of Bubble Sort.
The number of comparisons is constant in Bubble Sort so in average case, there is O(N^2) comparisons. This is because irrespective of the arrangement of elements, the number of comparisons C(N) is same.

For the number of swaps, consider the following points:

- If an element is in index I1 but it should be in index I2, then it will take a minimum of I2-I1 swaps to bring the element to the correct position.
- An element E will be at a distance of I3 from its position in sorted array. Maximum value of I3 will be N-1 for the edge elements and it will be N/2 for the elements at the middle.
- The sum of maximum difference in position across all elements will be:

(N-1) + (N-3) + (N-5) ... + 0 + ... + (N-3) + (N-1)
= N x N - 2 x (1 + 3 + 5 + ... + N/2)
= N^2 - 2 x N^2 / 4
= N^2 - N^2 / 2
= N^2 / 2

Therefore, in average, the number of swaps = O(N^2).

Therefore, in the average case time complexity of Bubble sort:

- Number of Comparisons: O(N^2) time
- Number of swaps: O(N^2) time

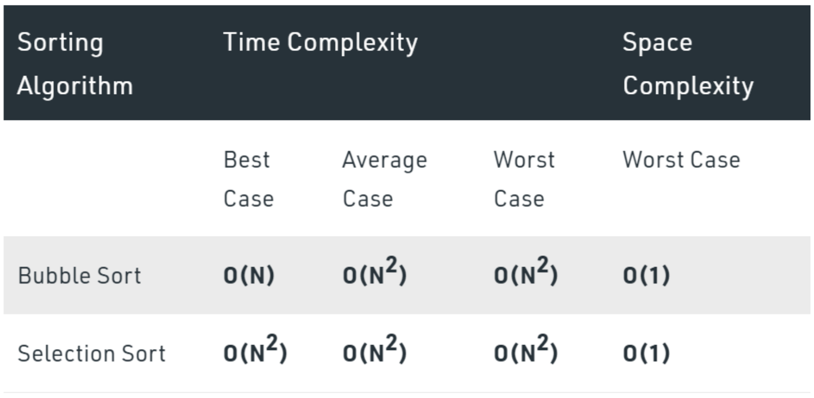
Space Complexity

The algorithm only requires auxiliary variables for flags, temporary variables and thus the space complexity is O(1).

Conclusion

- Worst Case Time Complexity is: O(N^2)
- Average Case Time Complexity is: O(N^2)
- Best Case Time Complexity is: O(N)
- Space Complexity: O(1)

Overall comparison :



Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	**O(N)**	**O(N^2)**	**O(N^2)**	**O(1)**
Selection Sort	**O(N^2)**	**O(N^2)**	**O(N^2)**	**O(1)**