# Unit2

Bubble Sort

# Bubble sort Algorithm

- Bubble sort works on the repeatedly swapping of adjacent elements until they reach the intended order.
- It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water.
- Bubbles in water rise up to the surface;
- similarly, the array elements in bubble sort move to the end in each iteration.

- An analogy would be bubbles in water – large bubbles tend to rise to the top over time while smaller ones sink down.
- Similarly, after each pass through our list, the largest remaining value gets positioned rightwards while smallest values filter towards the left.

- It is not suitable for large data sets.
- The average and worst-case complexity of Bubble sort is **$O(n^2)$**
- where n is a number of items.

# Algorithm

- In the algorithm given below, suppose arr is an array of n elements.
- The assumed swap function in the algorithm will swap the values of given array elements.

- begin BubbleSort(arr)
- for all array elements
- if arr[i] > arr[i+1]
- swap(arr[i], arr[i+1])
- end if
- end for
- return arr
- end BubbleSort

- Start at index zero, compare the element with the next one (a[0] & a[1] (a is the name of the array)), and swap if a[0] > a[1].
- Now compare a[1] & a[2] and swap if a[1] > a[2]. Repeat this process until the end of the array.
- After doing this, the largest element is present at the end.
- This whole thing is known as a pass.

- In the first pass, we process array elements from [0,n-1].
- Repeat step one but process array elements [0, n-2] because the last one, i.e., a[n-1], is present at its correct position.
- After this step, the largest two elements are present at the end.
- Repeat this process n-1 times.

# Working of Bubble sort Algorithm

Let the elements of array are -

| | | | | |
|---|---|---|---|---|
| 13 | 32 | 26 | 35 | 10 |

## First Pass

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

| | | | | |
|---|---|---|---|---|
| 13 | 32 | 26 | 35 | 10 |

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

| | | | | |
|---|---|---|---|---|
| 13 | 32 | 26 | 35 | 10 |

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

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 32 | 35 | 10 |

- Now, compare 32 and 35.

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 32 | 35 | 10 |

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.

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 32 | 35 | 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 -

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 32 | 10 | 35 |

Now, move to the second iteration.

- Second Pass
- The same process will be followed for second iteration.

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 32 | 10 | 35 |

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 32 | 10 | 35 |

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 32 | 10 | 35 |

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

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 10 | 32 | 35 |

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 10 | 32 | 35 |

Now, move to the third iteration.

- Third Pass
- The same process will be followed for third iteration.

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 10 | 32 | 35 |

| | | | | |
|---|---|---|---|---|
| 13 | 26 | 10 | 32 | 35 |

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

| | | | | |
|---|---|---|---|---|
| 13 | 10 | 26 | 32 | 35 |

| | | | | |
|---|---|---|---|---|
| 13 | 10 | 26 | 32 | 35 |

| | | | | |
|---|---|---|---|---|
| 13 | 10 | 26 | 32 | 35 |

Now, move to the fourth iteration.

## Fourth pass

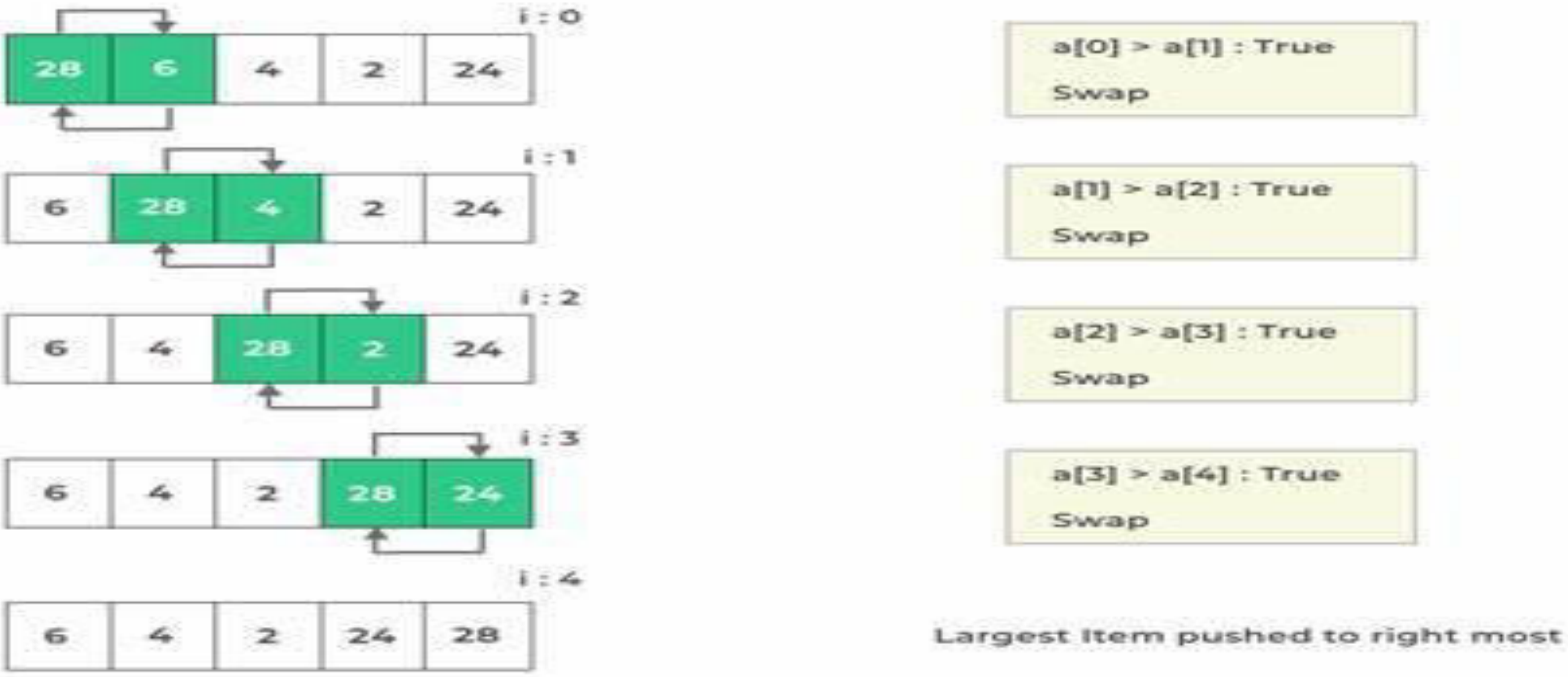
Similarly, after the fourth iteration, the array will be -

| | | | | |
|---|---|---|---|---|
| 10 | 13 | 26 | 32 | 35 |

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

# Another example

# Bubble Sort in C++



# Demo

- an example to demonstrate how Bubble Sort works:
- Example: Sorting [5, 3, 8, 6, 2]
- Initial Array: [5, 3, 8, 6, 2]
- Pass 1:
- Compare 5 and 3 → Swap → [3, 5, 8, 6, 2]
- Compare 5 and 8 → No Swap → [3, 5, 8, 6, 2]
- Compare 8 and 6 → Swap → [3, 5, 6, 8, 2]
- Compare 8 and 2 → Swap → [3, 5, 6, 2, 8]
- End of Pass 1: Largest element (8) is now in its correct position.

- Pass 2:
- Compare 3 and 5 → No Swap → [3, 5, 6, 2, 8]
- Compare 5 and 6 → No Swap → [3, 5, 6, 2, 8]
- Compare 6 and 2 → Swap → [3, 5, 2, 6, 8]
- End of Pass 2: Second largest element (6) is now in its correct position.

- Pass 3:
- Compare 3 and 5 → No Swap → [3, 5, 2, 6, 8]
- Compare 5 and 2 → Swap → [3, 2, 5, 6, 8]
- End of Pass 3: Third largest element (5) is now in its correct position.

- Pass 4:
- Compare 3 and 2 → Swap → [2, 3, 5, 6, 8]
- End of Pass 4: Fourth largest element (3) is now in its correct position.

- Final Sorted Array: [2, 3, 5, 6, 8] Key Points:
- Bubble Sort has a time complexity of $O(n^2)$ in the worst and average cases.
- It is simple but inefficient for large datasets.
- It works well for small or nearly sorted arrays.

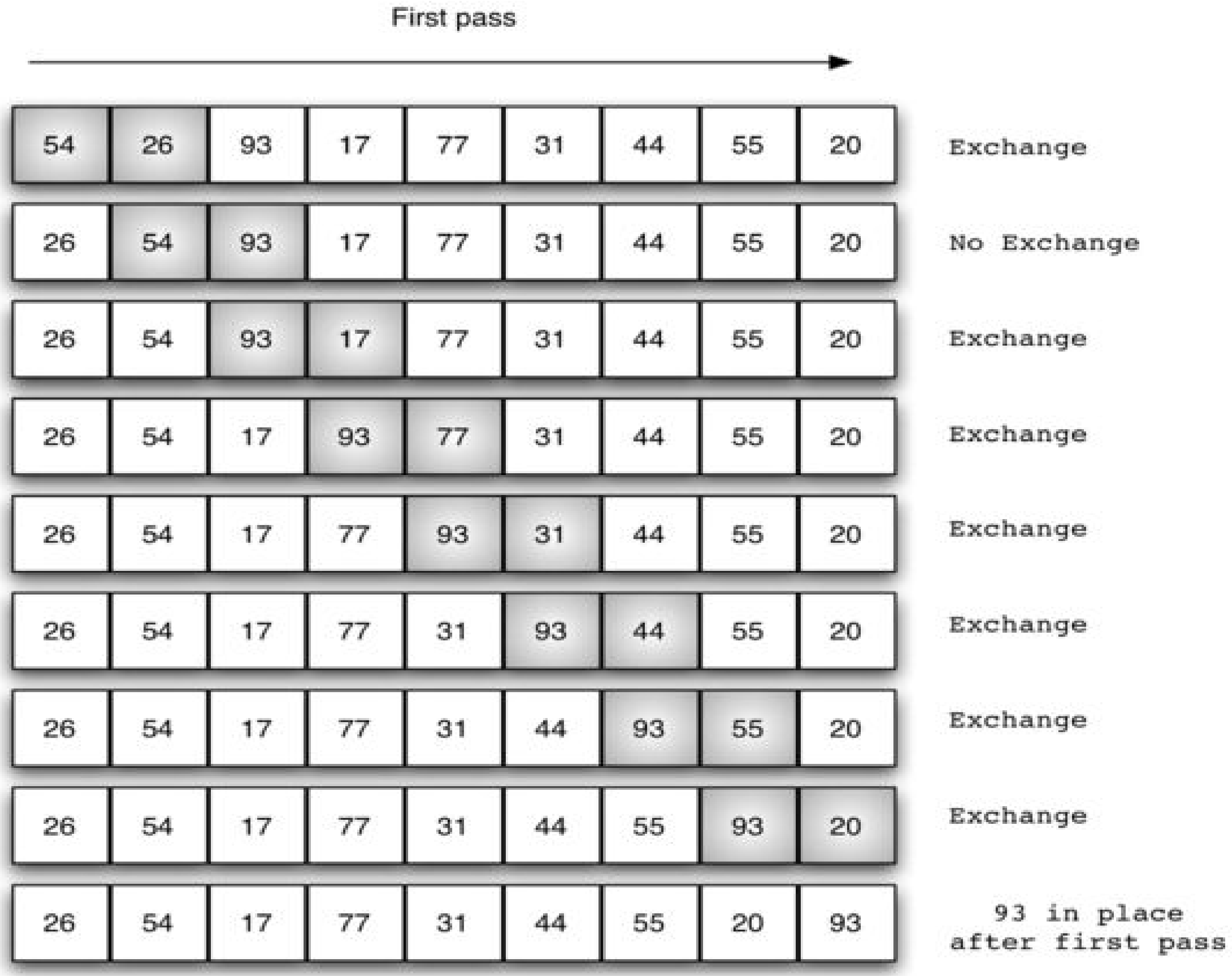



Figure 1: `bubbleSort` : The First Pass

≡ File Edit Search Run Compile Debug Project Options Window Help

GS\DST\BUBBLE~1.C 1

```c
//Bubble Sort Program
#include <stdio.h>
void  main()
{
  int array[10]={13,32,26,35,10};
  int n, i, j, temp,k;
  n=5;
  clrscr();
  for (i = 0 ; i < n-1 ; i++)
  {
    for (j = 0 ; j <n-i-1; j++)
    {
      if (array[j] > array[j+1])
      {
        temp = array[j];
        array[j]   = array[j+1];
        array[j+1] = temp;
      }
    }
  printf("Array after pass %d:\n",i+1);
  for (k = 0; k < n; k++)
```

1:1

F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
≡  File  Edit  Search  Run  Compile  Debug  Project  Options   Window  Help
[■]══════════ GS\DST\BUBBLE~1.C ══════════1=[↕]
  for (i = 0 ; i < n-1 ; i++)
  {
    for (j = 0 ; j <n-i-1; j++)
    {
      if (array[j] > array[j+1])
      {
        temp = array[j];
        array[j]   = array[j+1];
        array[j+1] = temp;
      }
    }
  printf("Array after pass %d:\n",i+1);
  for (k = 0; k < n; k++)
  printf("%d\n", array[k]);
  getch();
  }
 printf("Sorted list in ascending order:\n");
  for (i = 0; i < n; i++)
     printf("%d\n", array[i]);
    getch();
  }
═══ 30:1 ═══
F1 Help  Alt-F8 Next Msg  Alt-F7 Prev Msg  Alt-F9 Compile  F9 Make  F10 Menu
```

```
Array after pass 1:
13
26
32
10
35
Array after pass 2:
13
26
10
32
35
Array after pass 3:
13
10
26
32
35
Array after pass 4:
10
13
26
32
35
```

```
Array after pass 2:
13
26
10
32
35
Array after pass 3:
13
10
26
32
35
Array after pass 4:
10
13
26
32
35
Sorted list in ascending order:
10
13
26
32
35
_
```

## 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 **$O(n^2)$.**

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