C++ Course Notes | Sorting Algorithms | Week 9

Hello everyone welcome to the weekly lecture notes

Topics to be covered:

- Bubble Sort Algorithm
    1. Bubble Sort Time Complexity
    2. Bubble Sort Space Complexity
    3. Worst case scenario in Bubble Sort
    4. How to optimize the bubble sort in the case of nearly sorted arrays?
    5. Stable and Unstable sort
- Selection Sort Algorithm
    1. Example Explanation
    2. Selection Sort Code
    3. Selection Sort Time and Space Complexity
    4. Is selection sort stable?
    5. Applications of selection sort
- Insertion Sort Algorithm
    1. Insertion Sort Code
    2. Insertion Sort Time and Space Complexity
    3. Is Insertion Sort Stable ?
    4. Applications of Insertion sort

Bubble Sort Algorithm

This algorithm works on the principle of swapping adjacent elements if they are not in sorted order and it keeps swapping until the final array becomes sorted. At each pass we keep on checking the adjacent elements and reach the end of the array, this will make the last element of the array sorted and keep repeating this process will result in formation of sorted array at the end part.

Example:

```
Array = [10, 40, 30, 50, 20]
```

In the above example, we keep comparing the adjacent elements and perform the required operations as follows:

First Pass: Check 1 is between 10 and 40, so no need to swap here. The 2nd check is between 40 and 30 so here, we need to swap them. The new array looks like [10,30,40,50,20].

The 3rd check is between 40 and 50, they are in correct order so no need to swap them, and the 4th check is between 50 and 20, and we know that 50 > 20, so we need to swap them.

Array after first pass: `[10, 30, 40, 20, 50]`

Second Pass: In the second pass, we already know that the last element will be the maximum element as all swaps in First pass will move the maximum element to the last index. So we only need to compare elements just one before that only. We start with the 1st check of 10 and 30. They are in order so no need to swap them. Then we check 30 and 40, they are also in order so no need to swap them. The 3rd check will be of 40 and 20, now we know that 40 > 20, so we need to swap them.

The array now becomes: `[10, 30, 20, 40, 50].`

Third Pass: Now we know that the last 2 elements of the array will be at their correct positions in sorted order as 2 passes are already done, so we do not need to check them.

First check will be of 10 and 30, they are already in sorted order so no need to swap them, 2nd check will be of 30 and 20 and we know that 30 > 20, so we need to swap them.

Array after third pass: `[10, 20, 30, 40,50]`

Fourth Pass:

Before starting the fourth pass, the last 3 elements will be at their correct positions as 3 passes are already done, So we do not need to check them. So the first check is 10 and 20, they are already at the correct position so no need to swap them. No need to check further as they are already sorted .

Array after fourth pass: `[10, 20, 30, 40, 50]`

This way we have our sorted array at the end.

Bubble Sort Code:

```
#include <bits/stdc++.h>
using namespace std;

void bubbleSort(vector<int>&a){
  int n = a.size();
  for (int i = 0; i < n - 1; i++){
    // Last i elements are already at correct sorted positions so no need to check
them
        for (int j = 0; j < n - i - 1; j++){
            if (a[j] > a[j + 1]){
                swap(a[j], a[j + 1]);
            }
        }
    }
}

int main(){
    int n;
    cout<<"Enter the size of array"<<endl;
    cin>>n;
    vector<int>v(n);
    cout<<"Enter the elements"<<endl;
    for(int i=0;i<n;i++){
```

Output

```
Enter the size of the array
```

5

```
Enter the elements
```

```
2 5 3 1 7
```

```
Array after sorting
```

```
1 2 3 5 7
```

Bubble Sort Time Complexity:

The time complexity of the following algorithm will be $O(n^2)$ as at each pass we are traversing the array and swapping adjacent elements whenever needed. Each traversal is O(n) and there are n traversals in total so $O(n * n) = O(n^2)$ in worst case time complexity.

Bubble Sort Space Complexity:

It does not require any extra space so space complexity is O(1). Thus it will be an in-place sorting algorithm.

Maximum number of swaps in worst case in Bubble Sort:

The maximum number of swaps in 1st pass can be (n-1) when everything has to be swapped.

Maximum number of swaps in 2nd pass will be (n-2) as last element will already be maximum after 1st pass.

Maximum number of swaps in 3rd pass will be (n-3) as last 2 elements will already be sorted after 2nd pass.

Similarly in the last pass or (n-1)th pass, the maximum number of swaps can be 1.

Total number of maximum swaps = (n-1) + (n-2) + (n-3) + ... + 1 = (n*(n-1))/2

Note: The maximum number of swaps occurs when the array is strictly decreasing in nature.

How to optimize the bubble sort in case of nearly sorted arrays?

If we can identify that the array is sorted, then execution of further passes should be stopped. This is the optimization over the original bubble sort algorithm in case of nearly sorted arrays.

If you find there is no swapping in a particular pass, it means the array has become sorted, so you should not go for the further passes. For this you can use a flag variable which is set to true before each pass and is made to false whenever a swapping operation is executed.

Code :

```
#include <bits/stdc++.h>
using namespace std;

void bubbleSort(vector<int>&a){
    int n = a.size();
    for(int i=0; i<n; i++){
        int n = a.size();
        bool flag = false;
        for(int j=0; j<n-i-1; j++){
            if(a[j]>a[j+1]){
                flag = true;
                swap(a[j], a[j + 1]);
            }
        }
      // No Swapping happened, array is sorted
      if(!flag)return;
   }
}

int main(){
    int n;
    cout<<"Enter the size of array"<<endl;
    cin>>n;
```

Output :

```
Enter the size of the array
5
Enter the elements
2 5 3 1 7
Array after sorting
1 2 3 5 7
```

Note- if all the passes are performed, then our optimized algorithm will in fact perform a little slower than the original one.

Stable and Unstable Sort

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appeared in the input unsorted array.

Is Bubble Sort Stable?

Yes, Bubble Sort is a stable sorting algorithm. We swap elements only when A is less than B. If A is equal to B, we do not swap them, hence relative order between equal elements will be maintained.

Selection Sort Algorithm

This algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

- The subarray which is already sorted.
- The remaining subarray was unsorted

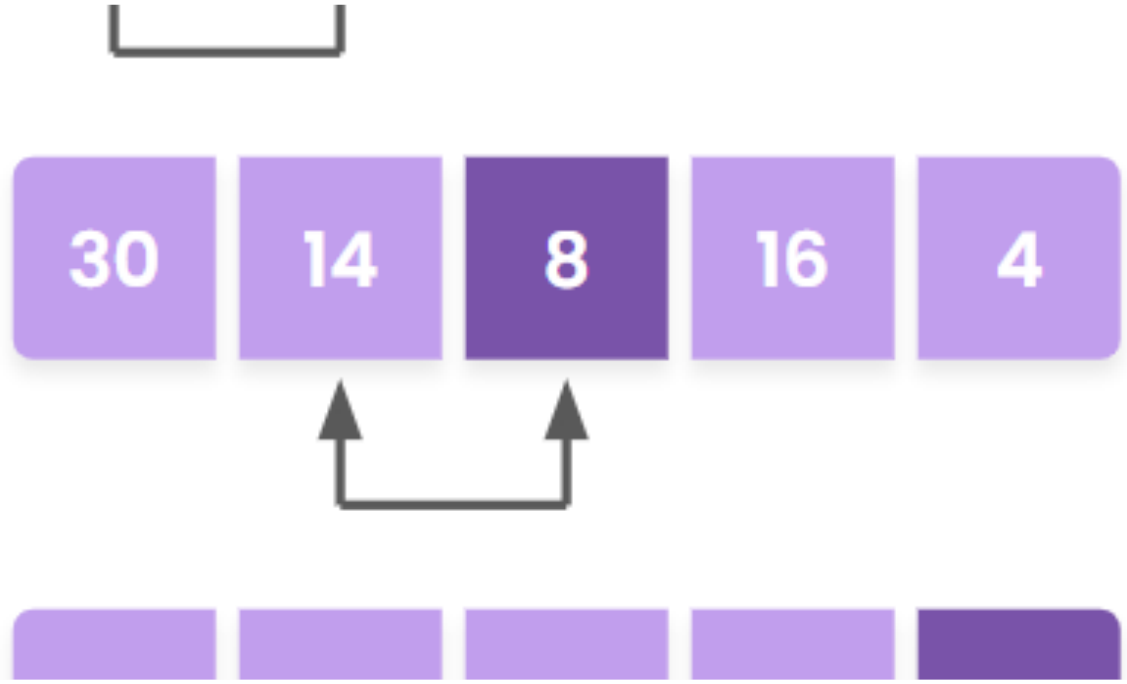
As the name suggests, in the Selection Sort algorithm we select the index having minimum element by traversing through the array and swap it with the current index, then increment the current index. This way we keep on traversing the rest of the array and finding the minimum index and swap it with the current index. This will result in forming a sorted array on the left side.

- Iterate through the array and consider the current element index.
- Now traverse the rest of the array elements and find the minimum element index.
- Swap it with the current element index and increment the current element index by one.
- Repeat until we reach the end of the array.

Working of Selection Sort

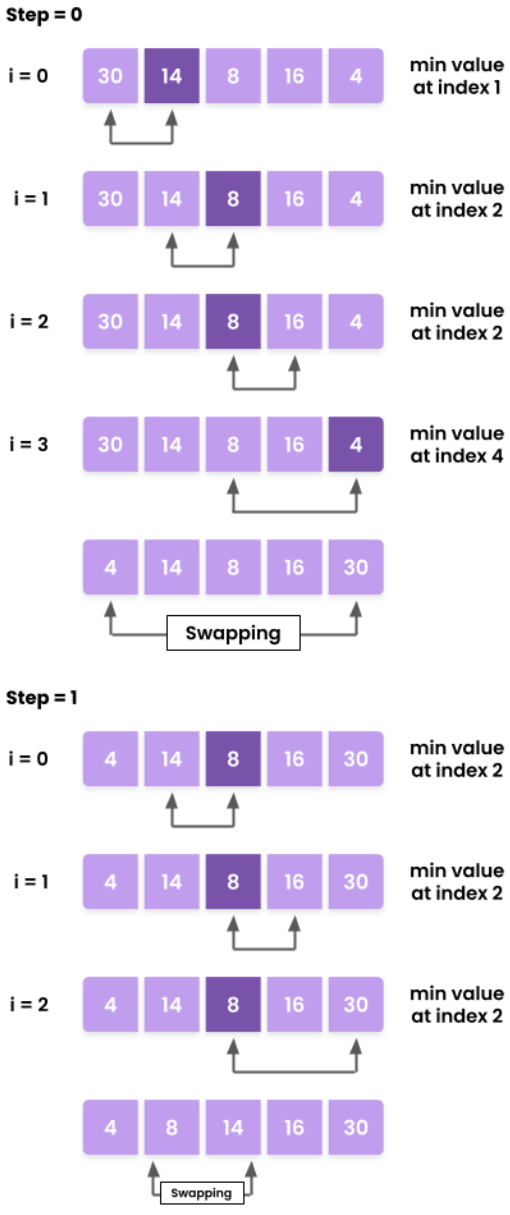
In selection sort we select an element for a given index, and let say if we are selecting an element for i'th index then it should have <=i element which are smaller or equal to it and remaining greater to it. This whole thing is difficult for any random index but for the first or last index we can do it easily as this will be the minimum or maximum valued index for them respectively. and once the minimum (or maximum) element is placed correctly then we can ignore it and do the same for remaining array of size n-1

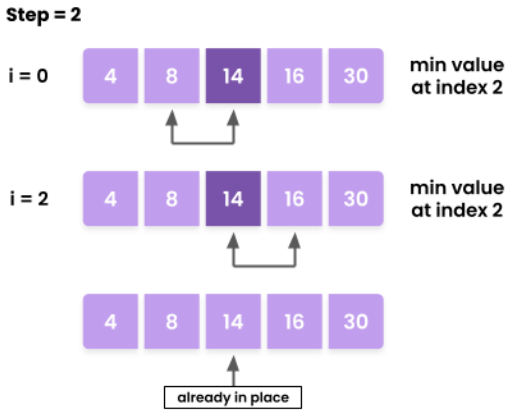
1. Set the first element as minimum.
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

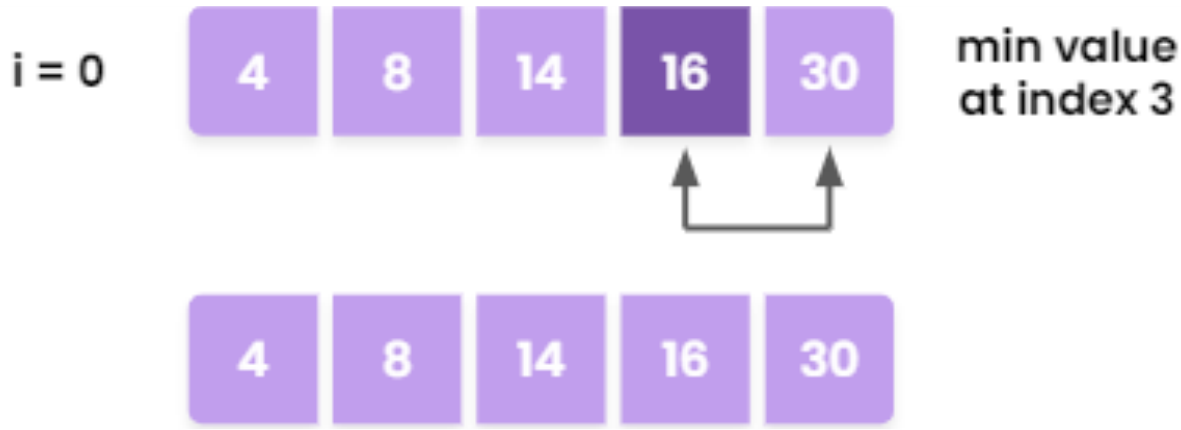


Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

3. After each iteration, minimum is placed in the front of the unsorted list.


Step = 0
i = 0
30 14 8 16 4
min value at index 1
i = 1
30 14 8 16 4
min value at index 2
i = 2
30 14 8 16 4
min value at index 2
i = 3
30 14 8 16 4
min value at index 4
4 14 8 16 30
Swapping
Step = 1
i = 0
4 14 8 16 30
min value at index 2
i = 1
4 14 8 16 30
min value at index 2
i = 2
4 14 8 16 30
min value at index 2
4 8 14 16 30
Swapping





4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions .

Example: Sort the given array using Selection Sort Algorithm.

```
Array = [10, 40, 30, 50, 20]
```

In the above example, as we iterate through the array from left to right, we keep finding the minimum index element and keep swapping it with the current element index and then increment the current index. Repeat the process until we reach the end of the array. This way we will have a sorted array at the end.

First Pass: Current index is 0 and current element is 10. Now just take min_index to be the current index and traverse through the rest of the array and update when we find a more minimal value. This way we have our minimum value index and swap it. In this case, the minimum value is 10 only so no need to swap anything and just increment the iterator by one.

Second Pass: Now the current element is 40. We find the minimum value index by traversing through the rest of the array and find that the minimum value in the rest of the array is 20 with index being 4. (taking 0-based indexing). Thus we swap the value at current index i.e. 40 with minimum index value i.e. 20. And move the iterator by one.

Array after second pass: `[10, 20, 30, 50, 40]`

Third Pass: The current element now becomes 30 and the minimum value is found by traversing the rest of the array and here we found that 30 is the minimum value only so no need to swap it with anything and simply increment the iterator by one.

Fourth Pass: The current element now is 50. And we find the minimum value in rest of the array is 40 so we swap it with current value index and increment the iterator by one

Array after fourth pass: `[10, 20, 30, 40, 50]`

This way we have our sorted array at the end.

Selection Sort Code:

```
#include <bits/stdc++.h>
using namespace std;
void selectionSort(vector<int> &a){
    int n = a.size();
    for (int i = 0; i < n-1; i++){
        // Find the minimum element in unsorted part of the array
        int min_index = i;
        for (int j = i+1; j < n; j++){
            if (a[j] < a[min_index]) min_index = j;
        }
        // Swap the found minimum element with the current element
        if(min_index!=i) swap(a[min_index], a[i]);
    }
}
int main(){
    int n;
    cout<<"Enter the size of array"<<endl;
    cin>>n;
    vector<int>v(n);
    cout<<"Enter the elements"<<endl;
    for(int i=0;i<n;i++){
        cin>>v[i];
    }
```

Output :

```
Enter the size of the array
```

```
5
Enter the elements
2 5 1 4 6
Array after sorting
1 2 4 5 6
```

Selection Sort Time Complexity

The time complexity of the following algorithm will be $O(n^2)$ as at each particular index from left to right as we are iterating in an array, we are finding the minimum value index by traversing through the rest of the array. Thus we require O(n) to find the minimum value index and we are doing this process n time so final time complexity will be $O(n^2)$.

Selection Sort Space Complexity

It does not require any extra space so space complexity is O(1). Thus it will be an in-place sorting algorithm.

Is selection sort stable?

As we know a sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appeared in the input unsorted array.

Selection sort works by finding the minimum element in the array and then inserting it in its correct position by swapping with the element which has minimum value. This is what makes it unstable.

Let's see an example :

`arr[] ={Deer,Monkey,Deer,Camel}`

In the first iteration of the outer for-loop, the algorithm determines that "`Camel`"is the minimal element and exchanges it with the blue "Deer" :

`arr[] ={Camel,Monkey,Deer,Deer}`

Then, it finds that the red Deer is the minimal item in the rest of the array and swaps it with "Monkey":

`arr[] ={Camel,Deer,Monkey,Deer}`

Finally, since Deer < Monkey in the usual lexicographic order, Selection Sort swaps the blue Deer with Monkey:

`arr[] ={Camel,Deer,Deer,Monkey}`

As you can see, `arr` doesn't maintain the relative order of the two strings. Since they are equal, the one that was initially before the other should come first in the output array. But, Selection Sort places the red Deer before the blue one even though their initial relative order was opposite.

So, we can conclude that Selection Sort isn't stable.

Selection Sort Applications

The selection sort is preferred when

- the cost of swapping is not an issue
- Number of swaps in the selection sort is of O(N) while O(N^2) in other brute force sorting algorithms.

Insertion Sort

Algorithm Explanation:

Let's suppose you have an array of integer values and you want to sort these values in non-decreasing order. Then the logic of Insertion sort states that you need to insert the element into the correct position by looping over the array, thereby forming a sorted array on the left.

Then increase the pointer and similarly for the next element find its correct position in the left sorted array and insert it. Do this till the pointer reaches the end of the array.

Insertion sort is a way of sorting where we pick one element , find its correct position and then insert it there or vice versa where we pick a position and find the correct element which should be there in the sorted array and then put it there. If we pick the positions from left to right then it boils down to finding the smallest element because the leftmost position should have the smallest element and put it there and then repeat the same process for all the indices.

- Traverse the array from left to right
- Compare the current element to its previous element and keep swapping it until the previous element is smaller than the current element.
- This way each element reaches it correct position

Example (with all the steps):

`Array = [10, 40, 30, 20, 50]`

In the above example, as we iterate through the array from left to right, we keep inserting elements into their corresponding correct positions thereby forming a sorted array on the left.

First Pass: 10 is to be compared with its predecessor, here there is no previous element so simply move to the next element.

The sorted part of the array on the left is currently 10.

`[10, 40, 30, 20, 50]`

Second Pass: 40 is compared with its predecessor and the previous element is already smaller than the current element (10 < 40) so no need to swap anything and move to the next element.

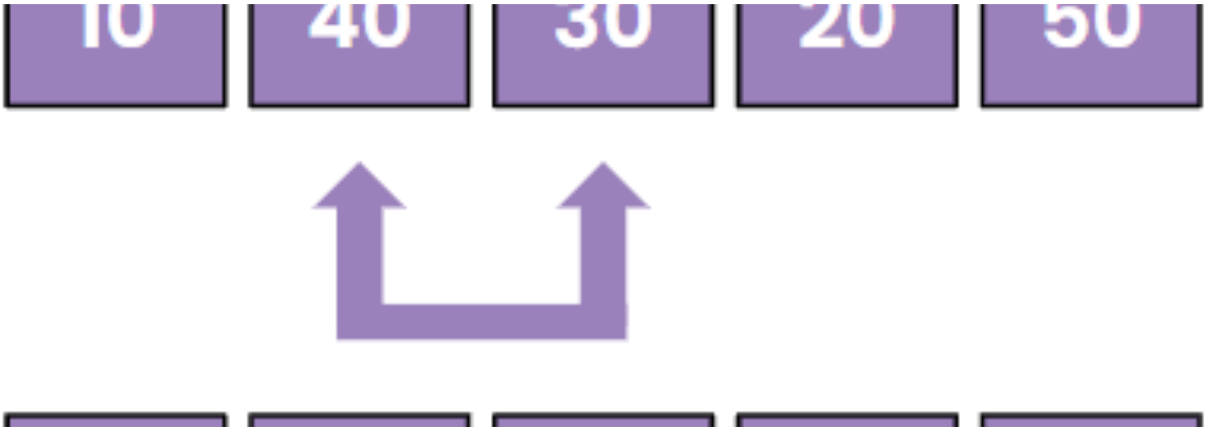
The sorted part of the array on the left is currently `[10,40]`.

`[10, 40, 30, 20, 50]`

Third Pass: 30 is now compared with its predecessor and here it is 40. As 40 < 30, we need to keep swapping the current element with its predecessors until the previous element is smaller than the current element. This way 30 will reach its correct position.

`[10, 40, 30, 20, 50]` (arrow between 40 and 30)

`[10, 30, 40, 20, 50]`



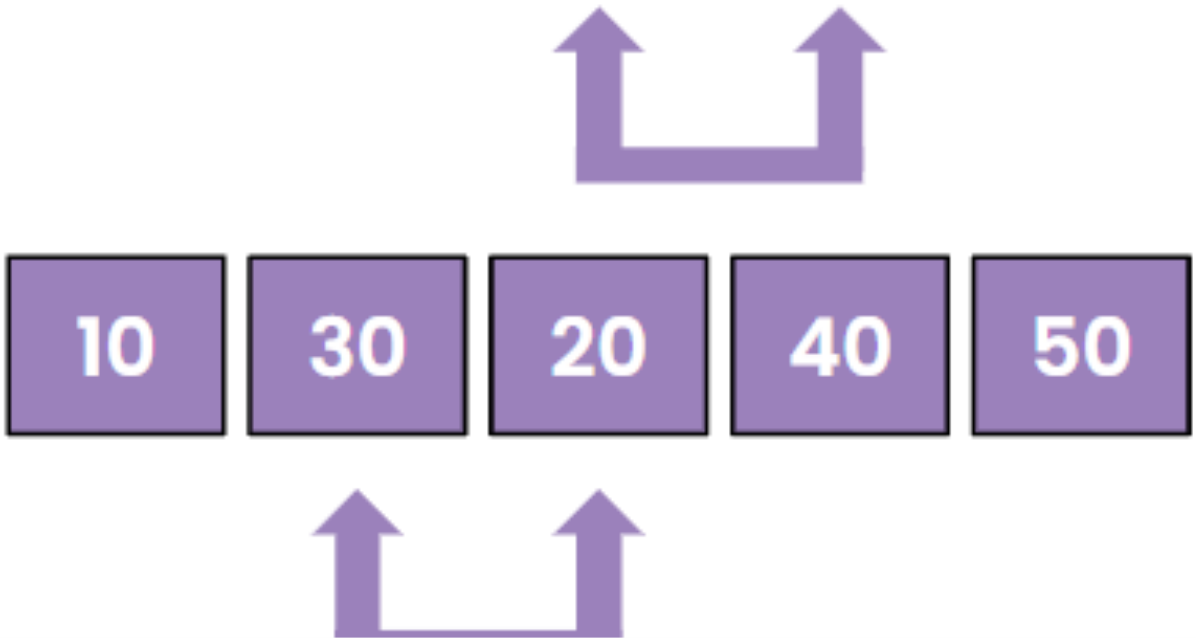
The sorted part of the array on the left is currently `[10,30,40]`.

Fourth Pass: 20 is now compared with its predecessor that is 40 so as it is smaller than 40. We keep swapping it, it now reaches and array looks after swapping will be [10,30,20,40]. Again 20 is checked with its predecessor and as it is smaller than 30 also, it will be swapped again and the array will look like [10,20,30,40]. Now 20 has reached its correct position and no more swaps are needed.

`[10, 30, 40, 20, 50]` (arrow between 20 and 40)

`[10, 30, 20, 40, 50]` (arrow between 30 and 20)

`[10, 20, 30, 40, 50]`

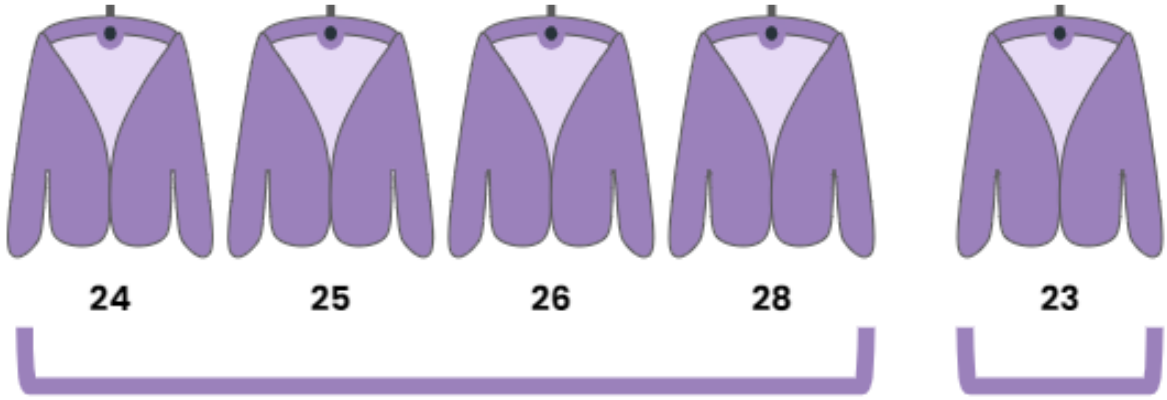


Fifth Pass: Here the current element is 50 and is compared with its predecessor and no swaps are needed here. So the final array looks like this - `[10,20,30,40,50]`

`[10,20,30,40,50]`

Real life example 1 :

Have you taken note, how tailors organize customers' shirts in their closet, according to measure. So they embed a new shirt at the right position, for that, they move existing shirts until they discover the correct place. If you consider the wardrobe as an array and shirts as a component, you may discover that we ought to move existing components to discover the correct place for the unused element.



Real life example 2:

Imagine that you are playing a game of cards. You're holding the cards in sorted order. Let's assume that the merchant hands you a new card. You may discover the position of the new card and embed it within the correct position. The existing cards more prominent than the unused card would be moved right by one place.

The same logic is used in insertion sort. You loop over the Array starting from the first index. Each new position is analogous to a new card, & you need to insert it in the correct place in the sorted subarray to your left.

Topic : Insertion Sort Code:

```
#include <bits/stdc++.h>
using namespace std;
// 0-based indexing used here

void insertionSort(vector<int> &a){
    int n = a.size();
    for (int i = 1; i < n; i++) {
        int j = i;
        // Insert a[i] into sorted left part 0..i-1
        while (j > 0 && a[j] < a[j - 1]) {
           // Swap a[j] and a[j-1]
            swap(a[j], a[j - 1]);
            // Decrement j by 1
            j --;
        }
    }
}

int main(){
    int n;
    cout<<"Enter the size of array"<<endl;
    cin>>n;
    vector<int>v(n);
```

Output :

```
Enter the size of the array
5
Enter the elements
2 5 1 4 6
Array after sorting
1 2 4 5 6
```

Insertion Sort Time Complexity:

The time complexity of the following algorithm will be $O(n^2)$ as for every particular element we find its correct position and insert it at that particular position. This requires O(n) time complexity for one element. We are doing this for every element so $O(n^2)$ time complexity in total.

Insertion Sort Space Complexity:

It does not require any extra space so space complexity is O(1).

Is insertion sort stable?

Here we just pick an element and place it in its correct place and in the logic we are only swapping the elements if the element is larger than the key, i.e. we are not swapping the element with the key when it holds equality condition, so insertion sort is stable sort.

Insertion Sort Applications

The insertion sort is used when:

- the array is has a small number of elements
- there are only a few elements left to be sorted
