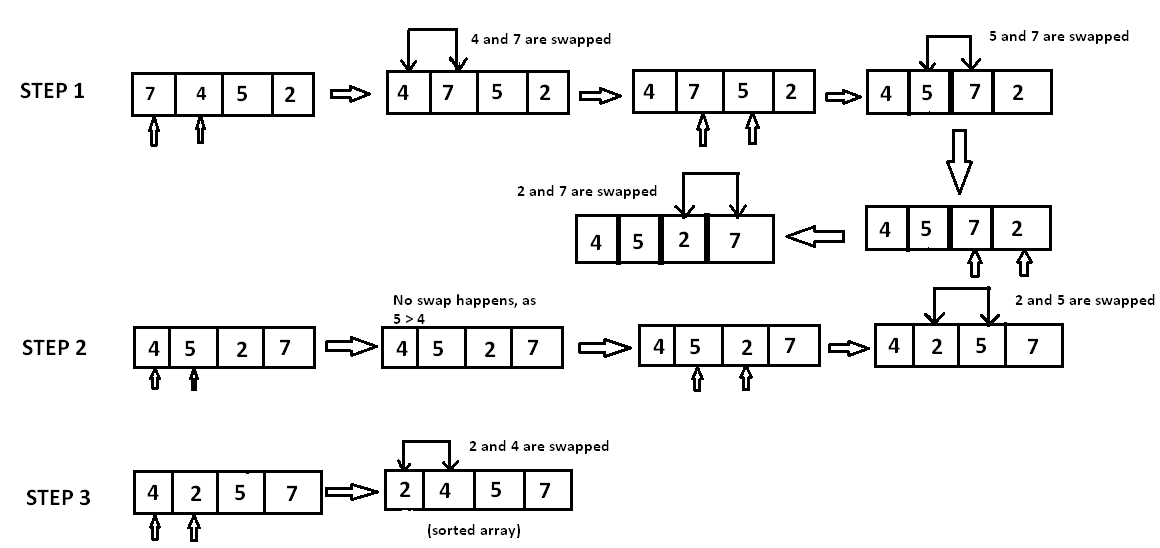
**Bubble Sort:**

Sorting Algorithms are concepts that every competitive programmer must know. Sorting algorithms can be used for collections of numbers, strings, characters, or a structure of any of these types.

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

Let’s try to understand the pseudo code with an example: A [ ] = { 7, 4, 5, 2}



**Complexity:**

**The complexity of bubble sort is O(n2) in both worst and average cases, because the entire array needs to be iterated for every element.**

**Pseudocode**

void bubble\_sort( int A[ ], int n ) {

int temp;

for(int k = 0; k< n-1; k++) {

// (n-k-1) is for ignoring comparisons of elements which have already been compared in earlier iterations

for(int i = 0; i < n-k-1; i++) {

if(A[ i ] > A[ i+1] ) {

// here swapping of positions is being done.

temp = A[ i ];

A[ i ] = A[ i+1 ];

A[ i + 1] = temp;

}

}

}

}

**Selection Sort**

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.



Time Complexity: Time Complexity:

To find the minimum element from the array of N elements, N-1 comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to N-2 and then, comparisons are required to find the minimum in the unsorted array.

Therefore, N -1 + N-2 + N-3 + … + 1 = (N(N-1))/2 comparisons and N swaps result in the overall complexity of O(N2).Auxiliary Space: O (1)

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

void selection\_sort (int A[ ], int n) {

// temporary variable to store the position of minimum element

int minimum;

// reduces the effective size of the array by one in each iteration.

for(int i = 0; i < n-1 ; i++) {

// assuming the first element to be the minimum of the unsorted array .

minimum = i ;

// gives the effective size of the unsorted array .

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

if(A[ j ] < A[ minimum ]) { //finds the minimum element

minimum = j ;

}

}

// putting minimum element on its proper position.

swap ( A[ minimum ], A[ i ]) ;

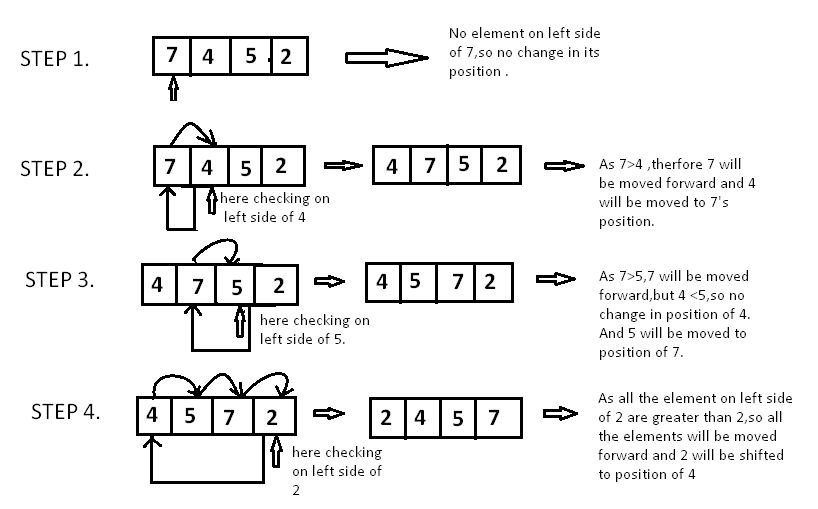
}

}

**Insertion sort**

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e, the position to which it belongs in a sorted array.

It iterates the input elements by growing the sorted array at each iteration. It compares the current element with the largest value in the sorted array. If the current element is greater, then it leaves the element in its place and moves on to the next element else it finds its correct position in the sorted array and moves it to that position. This is done by shifting all the elements, which are larger than the current element, in the sorted array to one position ahead



**Time Complexity:**

In worst case, each element is compared with all the other elements in the sorted array. For N elements, there will be N2 comparisons. Therefore, the time complexity is O(N2)

Implementation

void insertion\_sort ( int A[ ] , int n)

{

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

/\*storing current element whose left side is checked for its

correct position .\*/

int temp = A[ i ];

int j = i;

/\* check whether the adjacent element in left side is greater or

less than the current element. \*/

while( j > 0 && temp < A[ j -1]) {

// moving the left side element to one position forward.

A[ j ] = A[ j-1];

j= j - 1;

}

// moving current element to its correct position.

A[ j ] = temp;

}

}

**Merge Sort**

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

middle m = (l+r)/2

2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

