1. **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.

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning**11** 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 **12** 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 **22** 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 **25** 64

**In computer science, selection sort is an in-place comparison sorting algorithm. It has an O(n2) time complexity, which makes it inefficient on large lists, and generally performs worse than the similar insertion sort.**

1. **Bubble Sort**

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. I.e index i and i+1 is compared and swapped if i+1 is smaller then i.

Number of pass are number of element. If you observe for each pass largest element goes to end.

Example:   
First Pass:   
( 5 1 6 2 8 ) –> ( 1 5 6 2 8 ), i=0 Here, algorithm compares the first two elements, and swaps since 5 > 1.   
( 1 5 6 2 8 ) –>  ( 1 5 6 2 8 ), i=1Don’t swap since 5 < 6    
( 1 5 6 2 8 ) –>  ( 1 5 2 6 8 ), i=2Swap since 6 > 2   
( 1 5 2 6 8 ) –> ( 1 5 2 6 8 ), i=3 Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second pass as below: from i=0 to n-2

( 1 5 2 6 8 ) –> ( 1 5 2 6 8), i=0 Don’t swap since 1 < 5    
( 1 5 2 6 8 ) –>  ( 1 2 5 6 8 ), i =1 Swap since 5 > 2   
( 1 2 5 6 8 ) –> ( 1 2 5 6 8 ), i=2 Now, since these elements are already in order (6 > 5), algorithm does not swap them.

And so on…

**Worst Time Complexity : O(n2)**

**Average Time Complexity : O(n2)**

**Worst Time Complexity : O(n2)**

**Space Complexity : 1**

1. **Recursive Bubble Sort**

[Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/)is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.  
Example:   
First Pass:   
( 5 1 4 2 8 ) –> ( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.   
( 1 5 4 2 8 ) –> ( 1 4 5 2 8 ), Swap since 5 > 4   
( 1 4 5 2 8 ) –> ( 1 4 2 5 8 ), Swap since 5 > 2   
( 1 4 2 5 8 ) –> ( 1 4 2 5 8 ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.  
Second Pass:   
( 1 4 2 5 8 ) –> ( 1 4 2 5 8 )   
( 1 4 2 5 8 ) –> ( 1 2 4 5 8 ), Swap since 4 > 2   
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )   
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )   
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.  
Third Pass:   
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )   
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )   
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )   
( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )  
Following is iterative Bubble sort algorithm :

// Iterative Bubble Sort

bubbleSort(arr[], n)

{

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

// Last i elements are already in place

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

{

if(arr[j] > arr[j+1])

swap(arr[j], arr[j+1]);

}

}

See [Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/)for more details.  
How to implement it recursively?   
Recursive Bubble Sort has no performance/implementation advantages, but can be a good question to check one’s understanding of Bubble Sort and recursion.  
If we take a closer look at Bubble Sort algorithm, we can notice that in first pass, we move largest element to end (Assuming sorting in increasing order). In second pass, we move second largest element to second last position and so on.   
Recursion Idea.

**Worst Time Complexity : O(n2)**

**Average Time Complexity : O(n2)**

**Worst Time Complexity : O(n2)**

**Space Complexity : 1**

1. **Insertion Sort**

[Bubble Sort](https://www.geeksforgeeks.org/bubble-sort/)is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.  
Algorithm   
To sort an array of size n in ascending order:   
1: Iterate from arr[1] to arr[n] over the array.   
2: Compare the current element (key) to its predecessor.   
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.  
Example:   
 

Another Example:   
12, 11, 13, 5, 6  
Let us loop for i = 1 (second element of the array) to 4 (last element of the array)  
i = 1. Since 11 is smaller than 12, move 12 and insert 11 before 12   
11, 12, 13, 5, 6  
i = 2. 13 will remain at its position as all elements in A[0..I-1] are smaller than 13   
11, 12, 13, 5, 6  
i = 3. 5 will move to the beginning and all other elements from 11 to 13 will move one position ahead of their current position.   
5, 11, 12, 13, 6  
i = 4. 6 will move to position after 5, and elements from 11 to 13 will move one position ahead of their current position.   
5, 6, 11, 12, 13

// C++ program for insertion sort

#include <bits/stdc++.h>

**using** **namespace** std;

/\* Function to sort an array using insertion sort\*/

**void** insertionSort(**int** arr[], **int** n)

{

**int** i, key, j;

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

    {

        key = arr[i];

        j = i - 1;

        /\* Move elements of arr[0..i-1], that are

        greater than key, to one position ahead

        of their current position \*/

**while** (j >= 0 && arr[j] > key)

        {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

// A utility function to print an array of size n

**void** printArray(**int** arr[], **int** n)

{

**int** i;

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

        cout << arr[i] << " ";

    cout << endl;

}

/\* Driver code \*/

**int** main()

{

**int** arr[] = { 12, 11, 13, 5, 6 };

**int** n = **sizeof**(arr) / **sizeof**(arr[0]);

    insertionSort(arr, n);

    printArray(arr, n);

**return** 0;

}

**Worst Time Complexity : O(n2)**

**Average Time Complexity : O(n2)**

**Worst Time Complexity : O(n2)**

**Space Complexity : 1**

1. **Merge Sort**

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides the input array into 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 a 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 the 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-l)/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)

The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg" \t "https://www.geeksforgeeks.org/merge-sort/_blank) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



**Worst Time Complexity : O(n log(n))**

**Average Time Complexity : O(n log(n))**

**Worst Time Complexity : O(n log(n))**

**Space Complexity : O(n)**

1. **Quick Sort**

Like [Merge Sort](https://www.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

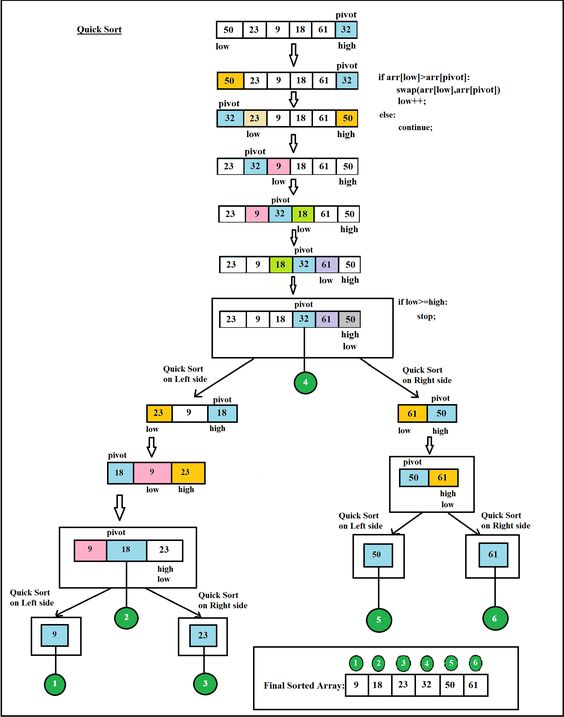
Always pick first element as pivot.

Always pick last element as pivot (implemented below)

Pick a random element as pivot.

Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



1. **Heap Sort:**

**Below are non comparison sort**

1. **Radix Sort:**

It is non - comparison sorting technique. Please go through below link for more information.

<https://brilliant.org/wiki/radix-sort/>

Below link shows the animation of how radix sort works. If you are not able to get anything from above link for sure you will be understand how radix sort works from animation.

<https://www.cs.usfca.edu/~galles/visualization/RadixSort.html>