Insertion Sort

```
public class InsertionSort {
  static int[] insertionSort(int[] array){
     for (int j = 1; j <array.length ; j++) {
       int key = array[j];
       int i = j-1;
       while(i>=0 && array[i]>key){
          array[i+1]= array[i];
          i = i-1;
       }
       array[i+1] = key ;
     }

       return array;
  }

    public static void main(String[] args) {
       int[] array = {20,4,5,7,1};
       insertionSort(array);
       for (int i = 0; i < array.length ; i++) {
          System.out.print(array[i]+" ");
       }
    }


}
```

/*
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:

Time Complexity: O(n*2)
Auxiliary Space: O(1)
Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order.
And it takes minimum time (Order of n) when elements are already sorted.
Algorithmic Paradigm: Incremental Approach
Sorting In Place: Yes

Stable: Yes
Online: Yes
Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted,
only few elements are misplaced in complete big array.
*/

Selection Sort

```
package soting;

import java.util.Arrays;

public class SelectionSort {
    void sort(int[]array) {
        for (int i = 0; i < array.length - 1; i++) {    // since no of pass will be n-1 n = array.length
            int min = i;
            for (int j = i + 1; j < array.length; j++) {
                if (array[j] < array[min]) {
                    min = j;
                }

            }
            int temp = array[min];  // swap arr(min , i)
            array[min] = array[i];
            array[i] = temp;

        }
    }

    public static void main(String[] args) {
        int[] a = {1,9,4,6,3,2};
        SelectionSort selectionSort = new SelectionSort();
        selectionSort.sort(a);
        System.out.println(Arrays.toString(a));
    }
    }
/*
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: O(n2) as there are two nested loops.

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
Stable = not stabe
in place = yes
*/

Bubble Sort

```
public class BubbleSort {

    void sort(int[]array){
        int n = array.length;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j <n-i-1 ; j++) {
                if(array[j]>array[j+1]){

                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }

            }

        }
    }
    void print(int[]array){
        for(int  i =0; i< array.length;i++){
            System.out.print(array[i]+" ");
        }
    }

    public static void main(String[] args) {
        int[] arr= {8,3,4,1,2,8,4};
        BubbleSort bubbleSort = new BubbleSort();
        bubbleSort.sort(arr);
        bubbleSort.print(arr);

    }
}
```

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

Merge Sort

```
public class MergeSort {
/*
The complexity of merge sort in all there cases is O(nlogn)
Like QuickSort, Merge Sort is a Divide and Conquer 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)
 */
       // Merges two subarrays of arr[].
       // First subarray is arr[l..m]
       // Second subarray is arr[m+1..r]
       void merge(int arr[], int l, int m, int r)
       {
          // Find sizes of two subarrays to be merged
          int n1 = m - l + 1;
          int n2 = r - m;

          /* Create temp arrays */
          int L[] = new int[n1];
          int R[] = new int[n2];

          /*Copy data to temp arrays*/
          for (int i = 0; i < n1; ++i)
              L[i] = arr[l + i];
          for (int j = 0; j < n2; ++j)
              R[j] = arr[m + 1 + j];

          /* Merge the temp arrays */

          // Initial indexes of first and second subarrays
          int i = 0, j = 0;
```

```
    // Initial index of merged subarry array
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of L[] if any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy remaining elements of R[] if any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// Main function that sorts arr[l..r] using
// merge()
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

/* A utility function to print array of size n */
```

```
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    // Driver code
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };

        System.out.println("Given Array");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length - 1);

        System.out.println("\nSorted array");
        printArray(arr);
    }
}
```

Count Sort

Linear sorting algo

/*

Time Complexity: O(n+k) where n is the number of elements in input array and k is the range of input.

Auxiliary Space: O(n+k)

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in the output sequence.

3) Output each object from the input sequence followed by decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place

```
  next data 1 at an index 1 smaller than this index.
 */
import java.util.Arrays;

public class CountSort {
    public static void main(String[] args) {
        int[] arr = {1,4,2,6,9,1,2};
        int maxElement = arr[0];
        for (int i = 0; i <arr.length ; i++) {
            if(arr[i]>maxElement){
                maxElement= arr[i];
            }
        }

       // System.out.println(maxElement);
        int[] countArray = new int[maxElement];
        for (int i = 0; i < arr.length; i++) {

            countArray[arr[i]-1]++;
        }
        System.out.println(Arrays.toString(countArray));
        int index = 0;
        for (int i = 0; i < countArray.length; i++) {


          if(countArray[i]!=0){//
              for (int j = 0; j <countArray[i] ; j++) {  //
                  arr[index]=i+1;
                   index++;
              }
          }
        }
        System.out.println(Arrays.toString(arr));

    }
}
```

Quick Sort

Like 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.
Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

```
// Java program for implementation of QuickSort
class QuickSort
{
    /* This function takes last element as pivot,
       places the pivot element at its correct
       position in sorted array, and places all
       smaller (smaller than pivot) to left of
       pivot and all greater elements to right
       of pivot */
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<high; j++)
        {
            // If current element is smaller than or
            // equal to pivot
            if (arr[j] <= pivot)
            {
                i++;

                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
```

```
    // swap arr[i+1] and arr[high] (or pivot)
    int temp = arr[i+1];
    arr[i+1] = arr[high];
    arr[high] = temp;

    return i+1;
}


/* The main function that implements QuickSort()
  arr[] --> Array to be sorted,
  low  --> Starting index,
  high  --> Ending index */
void sort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is
          now at right place */
        int pi = partition(arr, low, high);

        // Recursively sort elements before
        // partition and after partition
        sort(arr, low, pi-1);
        sort(arr, pi+1, high);
    }
}

/* A utility function to print array of size n  */
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i]+" ");
    System.out.println();
}


// Driver program
public static void main(String args[])
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = arr.length;

    QuickSort ob = new QuickSort();
    ob.sort(arr, 0, n-1);

    System.out.println("sorted array");
```

```
        printArray(arr);
    }
```

Best case = o(nlogn)
Wrost case = O(n^2)
Stable = no

Prims and Krukal's

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Below are the steps for finding MST using Kruskal's algorithm

Sort all the edges in non-decreasing order of their weight.
Pick the smallest edge. Check if it forms a cycle with the spanning-tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
Repeat step#2 until there are (V-1) edges in the spanning tree.
Prim's algorithm for MST

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Below are the steps for finding MST using Prim's algorithm

Create a set mstSet that keeps track of vertices already included in MST.
Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE.
Assign key value as 0 for the first vertex so that it is picked first.
While mstSet doesn't include all vertices
Pick a vertex u which is not there in mstSet and has minimum key value.
Include u to mstSet.
Update the key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v
Both Prim's and Kruskal's algorithm finds the Minimum Spanning Tree and follow the Greedy approach of problem-solving, but there are few major differences between them.

Prim's Algorithm Kruskal's Algorithm
It starts to build the Minimum Spanning Tree from any vertex in the graph. It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance. It traverses one node only once.
Prim's algorithm has a time complexity of O(V2), V being the number of vertices and can be improved up to O(E + log V) using Fibonacci heaps. Kruskal's algorithm's time complexity is O(E log V), V being the number of vertices.
Prim's algorithm gives connected component as well as it works only on connected graph. Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.Kruskal's algorithm runs faster in sparse graphs.

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

```
MST-PRIM(G, w, r)
1  for each u ∈ G.V
2      u.key = ∞
3      u.π = NIL
4  r.key = 0
5  Q = G.V
6  while Q ≠ ∅
7      u = EXTRACT-MIN(Q)
8      for each v ∈ G.Adj[u]
9          if v ∈ Q and w(u, v) < v.key
10             v.π = u
11             v.key = w(u, v)
```

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

There are n items in the store

Weight of ith item wi>0
Profit for ith item pi>0 and

Capacity of the Knapsack is W

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction xi of ith item.

$0 \leqslant xi \leqslant 1$
The ith item contributes the weight xi.wi to the total weight in the knapsack and profit xi.pi to the total profit.

Hence, the objective of this algorithm is to

$\text{maximize} \sum_{n=1}^{n} (xi.pi)$
subject to constraint,

$\sum_{n=1}^{n} (xi.wi) \leqslant W$
It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$\sum_{n=1}^{n} (xi.wi) = W$
In this context, first we need to sort those items according to the value of piwi, so that $pi+1wi+1 \leq piwi$. Here, x is an array to store the fraction of items.

If the provided items are already sorted into a decreasing order of piwi, then the while loop takes a time in O(n); Therefore, the total time including the sort is in O(n logn).

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks

Algorithm
1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While sptSet doesn’t include all vertices
….a) Pick a vertex u which is not there in sptSet and has minimum distance value.
….b) Include u to sptSet.
….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

DIJKSTRA(G,w,s)

```
    INITIALIZE-SINGLE-SOURCE (G,s)
 S=  phi
Q= V[G]
while Q!=  phi
      do u= EXTRACT-MIN(Q)
           S = S  {u}
           for each vertex v Adj[u]
                 do RELAX (u,v,w)
```

Time Complexity of Dijkstra's Algorithm is O (V 2) but with min-priority queue it drops down to O (V + E l o g V) .

BELLMAN-FORD(G, w, s)

```
    INITIALIZE-SINGLE-SOURCE(G, s)
    for i =1 to |V[G]| - 1
         do for each edge (u, v)  E[G]
              do RELAX(u, v, w)
    for each edge (u, v)  E[G]
          do if d[v] > d[u] + w(u, v)
              then return FALSE
     return TRUE
```

Algorithm
Following are the detailed steps.

Input: Graph and a source vertex src
Output: Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
…..a) Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then update dist[v]
…….dist[v] = dist[u] + weight of edge uv

3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

BFS DFS

1	Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
2	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
3	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
4	Suitablity for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
5	Speed	BFS is slower than DFS.	DFS is faster than BFS.
6	Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

BFS pusedocode

```
BFS(G,s)
1. for each vertex u in V[G] - {s}
2      do color[u] ← white
3           d[u] ← ∝
4           π[u] ← nil
5  color[s] ← gray
6  d[s] ← 0
7   π[s] ← nil
8  Q ← Φ
9  enqueue(Q,s)
10 while Q ≠ Φ
11     do u ← dequeue(Q)
12        for each v in Adj[u]
13          do if color[v] = white
14             then color[v] ← gray
15                  d[v] ← d[u] + 1
16                  π[v] ← u
17                  enqueue(Q,v)
18        color[u] ← black
```

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1              // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]        // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK              // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

The stability of a sorting algorithm is concerned with how the algorithm treats equal (or repeated) elements.
Stable sorting algorithms preserve the relative order of equal elements, while unstable sorting algorithms don't.
In other words, stable sorting maintains the position of two equals elements relative to one another.
Several common sorting algorithms are stable by nature, such as Merge Sort, Timsort, Counting Sort, Insertion Sort, and Bubble Sort.
Others such as Quicksort, Heapsort and Selection Sort are unstable.

An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'.
However, a small constant extra space used for variables is allowed.
In Place : Bubble sort, Selection Sort, Insertion Sort, Heapsort.

Not In-Place : Merge Sort. Note that merge sort requires O(n) extra space.

0/1 Knapsack

STEP 1: Draw a table say 'T' with (n+1) number of rows and (W+1) number of columns. Fill all the boxes of 0th row and 0th column with zeroes. Here, n is the number of items, whereas W represents the knapsack capacity.

STEP 2: Start filling the table row wise top to bottom from left to right. Use the following formula:

T (i , j) = max { T (i-1 , j) , profiti + T(i-1 , j – weighti) }

Here, T(i , j) = maximum profit of the selected items if we can take items 1 to i and have weight restrictions of j.

If j < weighti
T (i , j) = T (i-1 , j)

This step leads to completely filling the table. Then, profit of the last box represents the maximum possible profit that can be put into the knapsack.

```
// profits (stored in array v)
// Weights (stored in array w)
// Number of distinct items (n)
// Knapsack capacity (W)
Begin
  for i from 0 to n do:
    m[i, 0] := 0
  for j from 0 to W do:
    m[0, j] := 0
  for i from 1 to n do:
    for j from 0 to W do:
      if w[i] > j then:
        m[i, j] := m[i-1, j]
      else:
        m[i, j] := max(m[i-1, j], m[i-1, j-w[i-1]] + v[i-1])
End
```

Time complexity
Each entry of the table requires constant time θ(1) for its computation.
It takes θ(nW) time to fill (n+1)(W+1) table entries.
It takes θ(n) time for tracing the solution since tracing process traces the n rows.
Thus, overall θ(nW) time is taken to solve 0/1 knapsack problem using dynamic programming.

N Queen Problem
Algorithm
isValid(board, row, col)

Input: The chess board, row and the column of the board.
Output − True when placing a queen in row and place position is a valid or not.

```
Begin
  if there is a queen at the left of current col, then
    return false
  if there is a queen at the left upper diagonal, then
    return false
  if there is a queen at the left lower diagonal, then
    return false;
  return true //otherwise it is valid place
```

End
solveNQueen(board, col)

Input − The chess board, the col where the queen is trying to be placed.

Output − The position matrix where queens are placed.

```
Begin
  if all columns are filled, then
    return true
  for each row of the board, do
    if isValid(board, i, col), then
      set queen at place (i, col) in the board
      if solveNQueen(board, col+1) = true, then
        return true
      otherwise remove queen from place (i, col) from board.
  done
  return false
End
```