Towers of Hanoi

```
public class TOH {
   public static void main(String[] args) {
      int N=4;
      TowerOfHanoi(N,'A','C','B');

   }
   public static void TowerOfHanoi(int N, char from_rod, char to_rod, char help_rod){

      //Base condition
      if(N==0){
         return;
      }
      TowerOfHanoi(N-1,from_rod,help_rod,to_rod);
      System.out.println("Move Disk"+N+" From Rod "+ from_rod+" To Rod "+to_rod);
      TowerOfHanoi(N-1,help_rod,to_rod,from_rod);
   }
}
```

GCD of Given Two Numbers

```
public class GCD_2 {
   static int GCD(int a, int b){
      if(b==0){
         return a;
      }
      return GCD(b,a%b);
   }

   public static void main(String[] args) {
      System.out.println("GCD is "+ GCD(98,56));
   }
}
```

Factorial (Iterative)-

```
public class Factorial {
  public static void main(String[] args) {

    int num = 5;
    long factorial = 1;
    for(int i = 1; i <= num; ++i)
    {
      // factorial = factorial * i;
      factorial *= i;
    }
    System.out.printf("Factorial of %d = %d", num, factorial);
  }
}
```

Factorial (Recursive)-

```
public class Fact2 {
  static int factorial(int n){
    if(n==1){
      return 1;
    }
    return n*factorial(n-1);
  }
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int num=sc.nextInt();
    int Fact = factorial(num);
    System.out.println("Factorial of number "+num+" is "+Fact);
  }
}
```

Job Scheduling

```
class Job {

    // Each job has a unique-id,profit and deadline
    char id;
    int deadline, profit;

    // Constructors
    public Job() {}

    public Job(char id, int deadline, int profit)
    {
        this.id = id;
        this.deadline = deadline;
        this.profit = profit;
    }

    // Function to schedule the jobs take 2 arguments
    // arraylist and no of jobs to schedule
    void printJobScheduling(ArrayList<Job> arr, int t)
    {
        // Length of array
        int n = arr.size();

        // Sort all jobs according to decreasing order of
        // profit
        Collections.sort(arr,
                (a, b) -> b.profit - a.profit);

        // To keep track of free time slots
        boolean result[] = new boolean[t];

        // To store result (Sequence of jobs)
        char job[] = new char[t];

        // Iterate through all given jobs
        for (int i = 0; i < n; i++) {
            // Find a free slot for this job (Note that we
            // start from the last possible slot)
            for (int j
                 = Math.min(t - 1, arr.get(i).deadline - 1);
                 j >= 0; j--) {
                // Free slot found
                if (result[j] == false) {
                    result[j] = true;
                    job[j] = arr.get(i).id;
                    break;
                }
            }
        }

        // Print the sequence
        for (char jb : job)
            System.out.print(jb + " ");
        System.out.println();
    }

    // Driver's code
    public static void main(String args[])
    {
        ArrayList<Job> arr = new ArrayList<Job>();
        arr.add(new Job('a', 2, 100));
        arr.add(new Job('b', 1, 19));
        arr.add(new Job('c', 2, 27));
        arr.add(new Job('d', 1, 25));
        arr.add(new Job('e', 3, 15));

        System.out.println(
                "Following is maximum profit sequence of jobs");

        Job job = new Job();

        // Function call
        job.printJobScheduling(arr, 3);
    }
}
```

Knapsack Problem

```
class Knapsack {

   // A utility function that returns
   // maximum of two integers
   static int max(int a, int b)
   {
      return Math.max(a, b);
   }

   // Returns the maximum value that
   // can be put in a knapsack of
   // capacity W
   static int knapSack(int W, int wt[], int val[], int n)
   {
      // Base Case
      if (n == 0 || W == 0)
         return 0;

      // If weight of the nth item is
      // more than Knapsack capacity W,
      // then this item cannot be included
      // in the optimal solution
      if (wt[n - 1] > W)
         return knapSack(W, wt, val, n - 1);

         // Return the maximum of two cases:
         // (1) nth item included
         // (2) not included
      else
         return max(val[n - 1]
                  + knapSack(W - wt[n - 1], wt,
                  val, n - 1),
              knapSack(W, wt, val, n - 1));
   }

   // Driver code
   public static void main(String args[])
   {
      int val[] = new int[] { 60, 100, 120 };
      int wt[] = new int[] { 10, 20, 30 };
      int W = 50;
      int n = val.length;
      int MAX_VALUE = (knapSack(W, wt, val, n));
      System.out.println("The Maximum Value is :" +MAX_VALUE);
   }
}
```

Travelling Salesman Problem

```
// import required classes and packages

import java.util.*;
import java.io.*;
import java.util.Scanner;

// create TSPExample class to implement TSP code in Java
class TSPExample
{
  // create findHamiltonianCycle() method to get minimum weighted cycle
  static int findHamiltonianCycle(int[][] distance, boolean[] visitCity, int currPos, int cities, int count,
int cost, int hamiltonianCycle)
  {

    if (count == cities && distance[currPos][0] > 0)
    {
      hamiltonianCycle = Math.min(hamiltonianCycle, cost + distance[currPos][0]);
      return hamiltonianCycle;
    }

    // BACKTRACKING STEP
    for (int i = 0; i < cities; i++)
    {
      if (visitCity[i] == false && distance[currPos][i] > 0)
      {

        // Mark as visited
        visitCity[i] = true;
        hamiltonianCycle = findHamiltonianCycle(distance, visitCity, i, cities, count + 1, cost +
distance[currPos][i], hamiltonianCycle);

        // Mark ith node as unvisited
        visitCity[i] = false;
      }
    }
    return hamiltonianCycle;
  }

  // main() method start
  public static void main(String[] args)
  {
    int cities;

    //create scanner class object to get input from user
    Scanner sc = new Scanner(System.in);
```

```
        // get total number of cities from the user
        System.out.println("Enter total number of cities ");
        cities = sc.nextInt();


        //get distance of cities from the user
        int distance[][] = new int[cities][cities];
        for( int i = 0; i < cities; i++){
          for( int j = 0; j < cities; j++){
            System.out.println("Distance from city"+ (i+1) +" to city"+ (j+1) +": ");
            distance[i][j] = sc.nextInt();
          }
        }

        // create an array of type boolean to check if a node has been visited or not
        boolean[] visitCity = new boolean[cities];

        // by default, we make the first city visited
        visitCity[0] = true;


        int hamiltonianCycle = Integer.MAX_VALUE;

        // call findHamiltonianCycle() method that returns the minimum weight Hamiltonian Cycle
        hamiltonianCycle = findHamiltonianCycle(distance, visitCity, 0, cities, 1, 0, hamiltonianCycle);

        // print the minimum weighted Hamiltonian Cycle
        System.out.println(hamiltonianCycle);
    }
}
```

Sorting- 1) Merge Sort

```
public class MergeSort {

    public static void Conquer(int[] arr, int s, int mid, int e){
        int[] merged = new int[e-s+1];
        int i=s;
        int j=mid+1;
        int x=0;

        while (i<=mid && j<=e){
            if(arr[i]<=arr[j]){
                merged[x] = arr[i];
                x++;
                i++;
            }else {
                merged[x] = arr[j];
                x++;
                j++;
            }
        }
        while (i<=mid){
            merged[x] = arr[i];
            x++;
            i++;
        }
        while (j<=e){
            merged[x] = arr[j];
            x++;
            j++;
        }

        for(int k=0, l=s; k<merged.length; k++, l++){
            arr[l]=merged[k];
        }
    }
    public static void divide(int[] arr, int s, int e){
        if(s>=e){
            return;
        }
        int mid = s+(e-s)/2;
        divide(arr,s,mid);
        divide(arr,mid+1,e);

        Conquer(arr,s,mid,e);

    }
    public static void main(String[] args) {
        int[] arr = {6,3,9,5,2,8};
        int n= arr.length;

        divide(arr, 0, n-1);
        for (int i = 0; i < n; i++) {
            System.out.println(arr[i]+" ");
        }
        System.out.println();
    }
}
```

2) Quick Sort-

```
public class QuickSort {
    // A utility function to swap two elements
    static void swap(int[] arr, int i, int j)
    {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /* 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 */
    static int partition(int[] arr, int low, int high)
    {

        // pivot
        int pivot = arr[high];

        // Index of smaller element indicates the right position of pivot found so far
        int i = (low - 1);

        for (int j = low; j <= high - 1; j++) {

            // If current element is smaller
            // than the pivot
            if (arr[j] < pivot) {

                // Increment index of
                // smaller element
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }

    /* The main function that implements QuickSort
              arr[] --> Array to be sorted,
              low --> Starting index,
              high --> Ending index
     */
    static void quickSort(int[] arr, int low, int high)
    {
        if (low < high) {

            // pi is partitioning index, arr[p]
            // is now at right place
            int pi = partition(arr, low, high);

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

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

        quickSort(arr, 0, n - 1);

        //Print
        for (int i = 0; i < n; i++) {
            System.out.println(arr[i]+" ");
        }
        System.out.println();
    }
}
```

Multiplication of Matrix using threads

1.Worker Thread: -

```
public class WorkerThread extends Thread{
   private int row;
   private int col;
   private int [][] A;
   private int [][] B;
   private int [][] C;

   public WorkerThread(int row, int col, int[][] A,
                int[][] B, int[][] C) {
      this.row = row;
      this.col = col;
      this.A = A;
      this.B = B;
      this.C = C;
   }

   public void run() {
      C[row][col] = (A[row][0] * B[0][col])+ (A[row][1]*B[1][col]) ;
   }
}
```

2.Matrix Multiplication

```
public class MatrixMultiplication {

   //initializes variables for dimensions
   public static int M = 3;
   public static int K = 2;
   public static int N = 3;

   //Declares Arrays A,B,C, and an Array or WorkerThreads
   public static int [][] A = {{1,4}, {2,5}, {3,6}}; //Initializes A
   public static int [][] B = {{8,7,6}, {5,4,3}};    //Initializes B
   public static int [][] C = new int [M][N];
   public static WorkerThread [][] Threads = new WorkerThread[3][3];

   public static void main(String[] args) {
      //creates 9 Worker threads. Each thread Calculates a Matrix Value and sets it to C matrix
      for (int i = 0; i<M; i++){
         for (int j=0; j<N; j++){
            Threads[i][j] = new WorkerThread(i,j,A,B,C);
            Threads[i][j].start();
         }
      }
      //Outputs the Values of Matrix C
      System.out.println("Elements of Matrix C:");
      for (int i = 0; i<M; i++){
         for (int j=0; j<N; j++){
            System.out.println("["+i+","+j+"] = "+C[i][j]);
         }
      }
   }
}
```

Fibonacci

```
public class Fib {
      static int fib(int n)
      {
         if (n <= 1)
            return n;
         return fib(n - 1) + fib(n - 2);
      }

      public static void main(String args[])
      {
         int n = 9;
         System.out.println(fib(n));
      }

   /* This code is contributed by Rajat Mishra */

}
```

N queens Problem:

```
/* Java program to solve N Queen Problem using
backtracking */
public class NQueenProblem {
        final int N = 4;

        /* A utility function to print solution */
        void printSolution(int board[][])
        {
                for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++)
                                System.out.print(" " + board[i][j]
                                                                + " ");
                        System.out.println();
                }
        }

        /* A utility function to check if a queen can
        be placed on board[row][col]. Note that this
        function is called when "col" queens are already
        placeed in columns from 0 to col -1. So we need
        to check only left side for attacking queens */
        boolean isSafe(int board[][], int row, int col)
        {
                int i, j;

                /* Check this row on left side */
                for (i = 0; i < col; i++)
                        if (board[row][i] == 1)
                                return false;

                /* Check upper diagonal on left side */
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
            if (board[i][j] == 1)
                return false;

        /* Check lower diagonal on left side */
        for (i = row, j = col; j >= 0 && i < N; i++, j--)
            if (board[i][j] == 1)
                return false;

        return true;
    }

    /* A recursive utility function to solve N
    Queen problem */
    boolean solveNQUtil(int board[][], int col)
    {
        /* base case: If all queens are placed
        then return true */
        if (col >= N)
            return true;

        /* Consider this column and try placing
        this queen in all rows one by one */
        for (int i = 0; i < N; i++) {
            /* Check if the queen can be placed on
            board[i][col] */
            if (isSafe(board, i, col)) {
                /* Place this queen in board[i][col] */
                board[i][col] = 1;

                /* recur to place rest of the queens */
                if (solveNQUtil(board, col + 1) == true)
```

```
                                        return true;

                                /* If placing queen in board[i][col]
                                doesn't lead to a solution then
                                remove queen from board[i][col] */
                                board[i][col] = 0; // BACKTRACK
                        }
                }

                /* If the queen can not be placed in any row in
                this column col, then return false */
                return false;
        }

        /* This function solves the N Queen problem using
        Backtracking. It mainly uses solveNQUtil () to
        solve the problem. It returns false if queens
        cannot be placed, otherwise, return true and
        prints placement of queens in the form of 1s.
        Please note that there may be more than one
        solutions, this function prints one of the
        feasible solutions.*/
        boolean solveNQ()
        {
                int board[][] = { { 0, 0, 0, 0 },
                                                { 0, 0, 0, 0 },
                                                { 0, 0, 0, 0 },
                                                { 0, 0, 0, 0 } };

                if (solveNQUtil(board, 0) == false) {
                        System.out.print("Solution does not exist");
                        return false;
                }


                printSolution(board);
                return true;
        }


        // driver program to test above function
        public static void main(String args[])
        {
                NQueenProblem Queen = new NQueenProblem();
                Queen.solveNQ();
        }
}
// This code is contributed by Abhishek Shankhadhar
```