```
/*write a java program to implement mergesort
sample output
enter the size of an array
7
enter the elements of an array
12  14  5  8  3  90  4
after sorting array elements are
3   4  5   8   12  14  90 */
import java.util.*;
public class Mergesort
{
void mergeSort(int[] array, int low, int high){
	if(low < high){
		int middle = (low + high) / 2;
		mergeSort(array, low, middle);
		mergeSort(array, middle+1, high);
		merge(array, low, middle, high);
	}
}

void merge(int[] array, int low, int middle, int high){
	int[] helper = new int[array.length];
	for (int i = low; i <= high; i++) {
		helper[i] = array[i];
	}

	int helperLeft = low;
	int helperRight = middle+1;
	int current = low;

	while (helperLeft <= middle && helperRight <=high) {
		if(helper[helperLeft] <= helper[helperRight]){
			array[current] = helper[helperLeft];
			helperLeft++;

		}else{
			array[current] = helper[helperRight];
			helperRight++;
		}
		current ++;
	}

	int remaining = middle - helperLeft;
	for (int i = 0; i <= remaining; i++) {
		array[current+i] = helper[helperLeft+ i];
	}
}
public void printArray(int arr[])
{
for(int i=0;i<arr.length;i++)
System.out.print(arr[i]+" ");
System.out.println();
```

```java
}
public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
Mergesort ob=new Mergesort();
System.out.println("enter the size of an array");
int n=sc.nextInt();
int arr[]=new int[n];
System.out.println("enter the elements of an array");
for(int i=0;i<n;i++)
arr[i]=sc.nextInt();
ob.mergeSort(arr,0,n-1);
System.out.println("after sorting array elements are");
ob.printArray(arr);
}
}
```

```java
/*write a java program to find inversion count of an array
Note: Inversion Count for an array indicates how far (or close) the array is from being sorted.
If array is already sorted then inversion count is 0.
If array is sorted in reverse order that inversion count is the maximum.
Formally speaking, two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j)
expected output
Enter number of integer elements
4
Enter 4 integer elements
3 7 1 9
output
Inversion count is 2
*/
import java.util.*;
class inverse
{
   // Merge two sorted subarrays arr[low .. mid] and arr[mid + 1 .. high]
   public int merge(int[] arr,int low, int mid, int high)
   {
      int[] aux = new int[arr.length];
      int k = low, i = low, j = mid + 1;
      int inversionCount = 0;
      // While there are elements in the left and right runs
      while (i <= mid && j <= high)
      {
         if (arr[i] <= arr[j])
         {
            aux[k++] = arr[i++];
         }
         else {
            aux[k++] = arr[j++];
            inversionCount += (mid - i + 1);    // NOTE
         }
      }
```

```
        // Copy remaining elements
        while (i <= mid)
            aux[k++] = arr[i++];
       while(j<=high)    //copy remaining elements of the second list at end of temp[]
                aux[k++]=arr[j++];

        // copy back to the original array to reflect sorted order
        for (i = low; i <= high; i++) {
            arr[i] = aux[i];
        }

        return inversionCount;
    }

    // Sort array arr [low..high] using auxiliary array aux[]
    public int mergeSort(int[] arr,int low, int high)
    {
        // Base case
        if (high == low) {    // if run size == 1
            return 0;
        }

        // find mid point
        int mid = (low+high)/2;
        int inversionCount = 0;

        // recursively split runs into two halves until run size == 1,
        // then merge them and return back up the call chain

        inversionCount += mergeSort(arr,low, mid);      // split / merge left  half
        inversionCount += mergeSort(arr,mid + 1, high); // split / merge right half
        inversionCount += merge(arr,low, mid, high);    // merge the two half runs

        return inversionCount;
    }
}
import java.util.*;
class invCount
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner( System.in );
        int n, i;
        /* Accept number of elements */
        System.out.println("Enter number of integer elements");
        n = scan.nextInt();
        /* Create array of n elements */
        int arr[] = new int[ n ];
        /* Accept elements */
        System.out.println("Enter "+ n +" integer elements");
        for (i = 0; i < n; i++)
            arr[i] = scan.nextInt();
```

```
        inverse ob=new inverse();
       // get inversion count by performing merge sort on arr
       System.out.println("Inversion count is "+ob.mergeSort(arr,0,arr.length-1));
    }
}
```

/*java program implement Quicksort
expected output
Enter number of integer elements
5
Enter 5 integer elements
30   20   10 40 70
Sorted array
10 20 30 40 70

***/**

```
import java.util.*;
class Quicksort
{

 void swap (int a[], int left, int right)
{
	int temp;
	temp=a[left];
	a[left]=a[right];
	a[right]=temp;
}//end swap

void quicksort( int a[], int low, int high )
{
	int pivot;

	if ( high > low )
	{
		pivot = partition( a, low, high );
		// ...Student-to-write-code-here... to sort left-part of pivot item (use recursion)
		quicksort( a, low, pivot-1 );
		// ...Student-to-write-code-here... to sort right-part of pivot item (use recursion)
		quicksort( a, pivot+1, high );
	}
} //end quicksort

int partition( int a[], int low, int high )
{
	int left_index, right_index;
	int pivot_item;
	int pivot_index = left_index = low;
	pivot_item = a[pivot_index];
	right_index = high;
	while ( left_index < right_index )
	{
```

```
                while( a[left_index] <= pivot_item ) {left_index++;}          // Move left_index
forward while item < pivot

                while( a[right_index] > pivot_item ) {right_index--;}         // Move right_index
backward while item > pivot

                if ( left_index < right_index )     { swap(a,left_index,right_index); }
        }
        // right is final position for the pivot
        a[low] = a[right_index];
        a[right_index] = pivot_item;

        return right_index;
}
void printArray(int a[], int n)
{
        int i;
        for (i=0; i<n; i++)
                System.out.print(" "+a[i]);

}//end printarray
}
import java.util.*;
public class QuicksortTest
{
public static void main(String args[])
{

        Scanner scan = new Scanner( System.in );

      int n, i;
      /* Accept number of elements */
      System.out.println("Enter number of integer elements");
      n = scan.nextInt();
      /* Create array of n elements */
      int arr[] = new int[n];
      /* Accept elements */
      System.out.println("\nEnter "+ n +" integer elements");
      for (i = 0; i < n; i++)
         arr[i] = scan.nextInt();
      /* Call method sort */
      Quicksort ob = new Quicksort();
      ob.quicksort(arr, 0, arr.length-1);
      System.out.println("\nSorted array");
      ob.printArray(arr,n);


}

}
```

/*Write a program to sort an array a[] consisting of 0's,1's and 2's give an algorithm for sorting a[].
The algorithm should put all 0's first then all 1's and all 2's last (implement using count sort)
sample output
enter a size of an array
12
enter elements of an array
0 1 1 0 1 2 1 2 0 0 0 1
after sorting
0 0 0 0 0 1 1 1 1 1 2 2
***/**

```
import java.util.*;
public class CountingSort
{

        int[] c;

        public int[] sort(int[] a,int k)
        {
      int []b=new int[a.length];
                c=new int[k+1];
                for(int i=0;i<=k;i++){
                        c[i]=0;
                }
                for(int i=0;i<a.length;i++){
                        c[a[i]]=c[a[i]]+1;
                }
                for(int i=1;i<=k;i++){
                        c[i]=c[i]+c[i-1];
                }
                for(int i=a.length-1;i>=0;i--){
                        b[c[a[i]]-1]=a[i];
                        c[a[i]]=c[a[i]]-1;
                }

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

}
        public static void main(String[] args)
        {
  Scanner sc=new Scanner(System.in);
  System.out.println("enter a size of an array");
  int n=sc.nextInt();
        int arr[]=new int[n];
  System.out.println("enter elements of an array");
```

```
for(int i=0;i<n;i++)
        arr[i]=sc.nextInt();
            int max=arr[0];
    CountingSort in=new CountingSort();
    for(int i=0;i<n;i++)
    {
        if(max<arr[i])
             max=arr[i];
    }
            int result[]=in.sort(arr,max);
            System.out.println("after sorting");
    in.printArray(result);

            }
}
```

/*write a java program to implement simple union and simple find on the given disjoint sets

sample output
Intial sets
1 2 3 4 5
sets after union(4,3)
1 2 3 3 5
sets after union(2,1)
1 1 3 3 5
sets after union(1,3)
3 3 3 3 5 */

```
import java.util.*;

// class to represent a disjoint set
class Disjoint
{
    public static Map<Integer, Integer> parent = new HashMap();

    // perform MakeSet operation
    public void makeSet(int[] universe)
    {
        // create n disjoint sets (one for each item)
        for (int i : universe)
            parent.put(i, i);
    }

    // Find the root of the set in which element k belongs
    public static int Find(int k)
    {
        // if k is root
        if (parent.get(k) == k)
            return k;

        // recurse for parent until we find root
```

```
        return Find(parent.get(k));
    }

    // Perform Union of two subsets
    private void Union(int a, int b)
    {
        // find root of the sets in which elements
        // x and y belongs
        int x = Find(a);
        int y = Find(b);

        parent.put(x, y);
    }

    public void printSets(int[] universe)
    {
        for (int i : universe)
            System.out.print(Find(i) + " ");

        System.out.println();
    }


    public static void main(String[] args)
    {

        int[] universe = { 1, 2, 3, 4, 5 };


        Disjoint ds = new Disjoint();

        // create singleton set for each element of universe
        System.out.println("Intial sets");
        ds.makeSet(universe);
        ds.printSets(universe);

        System.out.println("sets after union(4,3)");
        ds.Union(4, 3); // 4 and 3 are in same set
        ds.printSets(universe);

        System.out.println("sets after union(2,1)");
        ds.Union(2, 1); // 1 and 2 are in same set
        ds.printSets(universe);

        System.out.println("sets after union(1,3)");
        ds.Union(1, 3); // 1, 2, 3, 4 are in same set
        ds.printSets(universe);
    }
}
```

```
/*write a java code for detecting cycles in the following graphs.you are given an input where
the first line is a pair(n,m). n is the number of vertices in the graph and m is the number of
edges.
After that m lines follow.Each line is a pair of vertices identifying each edge.
you will have to use the idea of disjoint sets and declare an output
"cycle" or "no cycle"
sample input1=
enter noof vertices and edges
5
5
enter the edges
2 3
4 1
4 5
4 2
3 5
cycle
sample input2=
enter noof vertices and edges
5
4
enter the edges
5 4
3 4
4 1
2 4
no cycle
*/
import java.util.*;
import java.lang.*;
import java.io.*;

class Graph
{
	int V, E;
	Edge edge[];

	class Edge
	{
		int src, dest;
	};


	Graph(int v,int e)
	{
		V = v;
		E = e;
		edge = new Edge[E];
		for (int i=0; i<e; ++i)
			edge[i] = new Edge();
	}
```

```
	int find(int parent[], int i)
	{
		if (parent[i] == -1)
				return i;
	    return find(parent, parent[i]);
	}


	void Union(int parent[],int x,int y)
	{
		int xset = find(parent, x);
int yset = find(parent, y);
parent[xset] = yset;
	}


	int isCycle( Graph graph)
	{
		// Allocate memory for creating V subsets
int parent[] = new int[graph.V];
// Initialize all subsets as single element sets
for (int i=0; i<graph.V; ++i)
parent[i]=-1;
// Iterate through all edges of graph, find subset of both
// vertices of every edge, if both subsets are same, then
// there is cycle in graph.
for (int i = 0; i < graph.E; ++i)
{
int x = graph.find(parent, graph.edge[i].src);
int y = graph.find(parent, graph.edge[i].dest);

if (x == y)
return 1;

graph.Union(parent, x, y);
}
return 0;
	}


	public static void main (String[] args)
	{

		int V, E;

		Scanner sc=new Scanner(System.in);
		System.out.println("enter noof vertices and edges");
		V=sc.nextInt();
		E=sc.nextInt();
      Graph graph = new Graph(V+1, E);
      System.out.println("enter the edges");
```

```
        for(int i=0;i<E;i++)
        {
                graph.edge[i].src=sc.nextInt();
                graph.edge[i].dest=sc.nextInt();
        }

                if (graph.isCycle(graph)==1)
                        System.out.println( "cycle" );
                else
                        System.out.println( "no cycle" );
         }
}
```

/*program to implement weigthed union on disjoint sets
sample output
Attach 2 to 1 Size of root(1) = 2
Attach 4 to 3 Size of root(3) = 2
Attach 5 to 3 Size of root(3) = 3
Attach 1 to 3 Size of root(3) = 5
Attach 0 to 3 Size of root(3) = 6
after updating sets
parent of 0 is=3
parent of 1 is=3
parent of 2 is=1
parent of 3 is=3
parent of 4 is=3
root of compressed tree
3
***/**

```
import java.util.Arrays;

    public class WeigthedUnion {
        int[] root;
        int[] sz;
        public WeigthedUnion(int N) {
            root = new int[N];
            sz = new int[N];
            for (int i = 0; i < N; ++i) {
                root[i] = i;
                sz[i] = 1;
            }
        }

         int rt(int i)
        {
            if(i == root[i])
             {
                return i;
             }
            root[i] = rt(root[i]);
            return root[i];
        }
```

```
    public void union(int p, int q)
   {
       int i = rt(p);
       int j = rt(q);
       if (i == j) {
           return;
       }
       if (sz[i] < sz[j]) {
           root[i] = j;
           sz[j] += sz[i];
           System.out.println("Attach "+i+" to "+j+" Size of"
                   + " root("+q+") = "+sz[j]);
       } else {
           root[j] = i;
           sz[i] += sz[j];
           System.out.println("Attach "+j+" to "+i+" Size of"
                   + " root("+p+") = "+sz[i]);
       }

    }

    public static void main(String[] args)
    {
       WeigthedUnion qf = new WeigthedUnion(6);
       qf.union(1, 2);
       qf.union(3, 4);
       qf.union(3, 5);
       qf.union(2, 3);
       qf.union(3, 0);
       System.out.println("after updating sets");
       for(int i=0;i<5;i++)
       System.out.println("parent of "+i+" is="+qf.root[i]);
       System.out.println("root of compressed tree");
       System.out.println(qf.rt(0));


    }
}
```

/*write a java program to find articulation points in a given graph
sample output:
Articulation points in first graph
0 3
Articulation points in Second graph
1 2
Articulation points in Third graph
1

***/**

```
import java.io.*;
import java.util.*;
import java.util.LinkedList;

// This class represents an undirected graph using adjacency list
// representation
class Graph
{
    private int V;   // No. of vertices

    // Array  of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];
    int time = 0;
    static final int NIL = -1;

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  // Add w to v's list.
        adj[w].add(v);    //Add v to w's list
    }

    // A recursive function that find articulation points using DFS
    // u --> The vertex to be visited next
    // visited[] --> keeps tract of visited vertices
    // disc[] --> Stores discovery times of visited vertices
    // parent[] --> Stores parent vertices in DFS tree
    // ap[] --> Store articulation points
    void APUtil(int u, boolean visited[], int disc[],
                int low[], int parent[], boolean ap[])
    {

        // Count of children in DFS Tree
        int children = 0;

        // Mark the current node as visited
        visited[u] = true;

        // Initialize discovery time and low value
        disc[u] = low[u] = ++time;

        // Go through all vertices aadjacent to this
        Iterator<Integer> i = adj[u].iterator();
```

```
    while (i.hasNext())
    {
        int v = i.next();  // v is current adjacent of u

        // If v is not visited yet, then make it a child of u
        // in DFS tree and recur for it
        if (!visited[v])
        {
            children++;
            parent[v] = u;
            APUtil(v, visited, disc, low, parent, ap);

            // Check if the subtree rooted with v has a connection to
            // one of the ancestors of u
            low[u]  = Math.min(low[u], low[v]);

            // u is an articulation point in following cases

            // (1) u is root of DFS tree and has two or more chilren.
            if (parent[u] == NIL && children > 1)
                ap[u] = true;

            // (2) If u is not root and low value of one of its child
            // is more than discovery value of u.
            if (parent[u] != NIL && low[v] >= disc[u])
                ap[u] = true;
        }

        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u]  = Math.min(low[u], disc[v]);
    }
}

// The function to do DFS traversal. It uses recursive function APUtil()
void AP()
{
    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    int disc[] = new int[V];
    int low[] = new int[V];
    int parent[] = new int[V];
    boolean ap[] = new boolean[V]; // To store articulation points

    // Initialize parent and visited, and ap(articulation point)
    // arrays
    for (int i = 0; i < V; i++)
    {
        parent[i] = NIL;
        visited[i] = false;
        ap[i] = false;
    }
```

```
}

        // Call the recursive helper function to find articulation
        // points in DFS tree rooted with vertex 'i'
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                APUtil(i, visited, disc, low, parent, ap);

        // Now ap[] contains articulation points, print them
        for (int i = 0; i < V; i++)
            if (ap[i] == true)
                System.out.print(i+" ");
    }

    // Driver method
    public static void main(String args[])
    {
        // Create graphs given in above diagrams
        System.out.println("Articulation points in first graph ");
        Graph g1 = new Graph(5);
        g1.addEdge(1, 0);
        g1.addEdge(0, 2);
        g1.addEdge(2, 1);
        g1.addEdge(0, 3);
        g1.addEdge(3, 4);
        g1.AP();
        System.out.println();

        System.out.println("Articulation points in Second graph");
        Graph g2 = new Graph(4);
        g2.addEdge(0, 1);
        g2.addEdge(1, 2);
        g2.addEdge(2, 3);
        g2.AP();
        System.out.println();

        System.out.println("Articulation points in Third graph ");
        Graph g3 = new Graph(7);
        g3.addEdge(0, 1);
        g3.addEdge(1, 2);
        g3.addEdge(2, 0);
        g3.addEdge(1, 3);
        g3.addEdge(1, 4);
        g3.addEdge(1, 6);
        g3.addEdge(3, 5);
        g3.addEdge(4, 5);
        g3.AP();
    }
}
```

```
/*
Java program to solve 8 Queen Problem using backtracking
The 4 Queen is the problem of placing 8 chess queens on an 8×8 chessboard
so that no two queens attack each other.
sample output=
one of the solution for 8-queens problem is
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
*/
public class Nqueens
{
	final int N=8;

	/* 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
	placed 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 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 queen can not be place in any row in
		this colum 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[][] = new int[N][N];

         //intialize board[][] with zeros
          for(int i=0;i<N;i++)
              for(int j=0;j<N;j++)
                board[i][j]=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
class NqueensTest
{

        public static void main(String args[])
        {
                Nqueens obj= new Nqueens();
                System.out.println("one of the solution for 8-queens problem is");
                obj.solveNQ();
        }
}
```

/* Java program for solution of M Coloring problem using backtracking
sample output=
enter adjacency matrix of graph
1 1 0 1
1 1 1 1
0 1 1 1
1 1 1 1
enter number of colors
2
Solution does not exist
enter adjacency matrix of graph
1 1 0 1
1 1 1 1
0 1 1 1
1 1 1 1
enter number of colors
3

Solution Exists: Following are the assigned colors
1 2 1 3
***/**

```
import java.util.*;
public class Mcoloring{

        int color[];
      final int V=4;
        /* A utility function to check if the current
        color assignment is safe for vertex v */
        boolean isSafe(int v, int graph[][], int color[],
                                int c)
        {
                for (int i = 0; i < V; i++)
                        if (graph[v][i] == 1 && c == color[i])
                                return false;
                return true;
        }

        /* A recursive utility function to solve m
        coloring problem */
        boolean graphColoringUtil(int graph[][], int m,
                                                int color[], int v)
        {

                /* base case: If all vertices are assigned
                a color then return true */
                if (v == V)
                        return true;

                /* Consider this vertex v and try different
                colors */
                for (int c = 1; c <= m; c++)
                {
                        /* Check if assignment of color c to v
                        is fine*/
                        if (isSafe(v, graph, color, c))
                        {
                                color[v] = c;

                                /* recur to assign colors to rest
                                of the vertices */
                                if (graphColoringUtil(graph, m,color, v + 1))
                                        return true;

                                /* If assigning color c doesn't lead
                                to a solution then remove it */
                                color[v] = 0;
                        }
                }
```

```
        /* If no color can be assigned to this vertex
        then return false */
        return false;
}

/* This function solves the m Coloring problem using
Backtracking. It mainly uses graphColoringUtil()
to solve the problem. It returns false if the m
colors cannot be assigned, otherwise return true
and prints assignments of colors to all vertices.
Please note that there may be more than one
solutions, this function prints one of the
feasible solutions.*/
boolean graphColoring(int graph[][], int m)
{
        // Initialize all color values as 0. This
        // initialization is needed correct functioning
        // of isSafe()

        color = new int[V];
        /*for (int i = 0; i < V; i++)
                color[i] = 0;*/

        // Call graphColoringUtil() for vertex 0
        if (!graphColoringUtil(graph, m, color, 0))
        {
                System.out.println("Solution does not exist");
                return false;
        }

        // Print the solution
        printSolution(color);
        return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
        System.out.println("Solution Exists: Following" +
                                        " are the assigned colors");
        for (int i = 0; i < V; i++)
                System.out.print(color[i] + " ");
        System.out.println();
}

// driver program to test above function
public static void main(String args[])
{
        Mcoloring Coloring = new Mcoloring();
        Scanner sc=new Scanner(System.in);
   //System.out.println("enter noof vertices");
   //int V=sc.nextInt();
```

```
                int graph[][] = new int[4][4];
            System.out.println("enter adjacency matrix of graph");
             for(int i=0;i<4;i++)
                 for(int j=0;j<4;j++)
                    graph[i][j]=sc.nextInt();
            System.out.println("enter number of colors");
                int m = sc.nextInt(); // Number of colors
                Coloring.graphColoring(graph, m);
        }
}
```

/*Java program for solution of Hamiltonian Cycle problem using backtracking
expected output=
enter noof vertices
5
enter adjacency matrix
0 1 0 1 0
1 0 1 1 1
0 1 0 0 1
1 1 0 0 1
0 1 1 1 0
Solution Exists: Following is one Hamiltonian Cycle
1 2 3 5 4 1
***/**

```
import java.util.*;
class HamiltonianCycle
{
    int V;
        int path[];

    HamiltonianCycle(int n)
     {
            V= n;
     }


        /* A utility function to check if the vertex v can be
        added at index 'pos'in the Hamiltonian Cycle
        constructed so far (stored in 'path[]') */
        boolean isSafe(int v, int graph[][], int path[], int pos)
        {
                /* Check if this vertex is an adjacent vertex of
                the previously added vertex. */
                if (graph[path[pos - 1]][v] == 0)
                        return false;

                /* Check if the vertex has already been included.
                This step can be optimized by creating an array
                of size V */
                for (int i = 0; i < pos; i++)
                        if (path[i] == v)
                                return false;
```

```
        return true;
}

/* A recursive utility function to solve hamiltonian
cycle problem */
boolean hamCycleUtil(int graph[][], int path[], int pos)
{
        /* base case: If all vertices are included in
        Hamiltonian Cycle */
        if (pos == V)
        {
                // And if there is an edge from the last included
                // vertex to the first vertex
                if (graph[path[pos - 1]][path[0]] == 1)
                        return true;
                else
                        return false;
        }

        // Try different vertices as a next candidate in
        // Hamiltonian Cycle. We don't try for 0 as we
        // included 0 as starting point in in hamCycle()
        for (int v = 1; v < V; v++)
        {
                /* Check if this vertex can be added to Hamiltonian
                Cycle */
                if (isSafe(v, graph, path, pos))
                {
                        path[pos] = v;

                        /* recur to construct rest of the path */
                        if (hamCycleUtil(graph, path, pos + 1) == true)
                                return true;

                        /* If adding vertex v doesn't lead to a solution,
                        then remove it */
                        path[pos] = -1;
                }
        }

        /* If no vertex can be added to Hamiltonian Cycle
        constructed so far, then return false */
        return false;
}

/* This function solves the Hamiltonian Cycle problem using
Backtracking. It mainly uses hamCycleUtil() to solve the
problem. It returns false if there is no Hamiltonian Cycle
possible, otherwise return true and prints the path.
Please note that there may be more than one solutions,
this function prints one of the feasible solutions. */
int hamCycle(int graph[][])
```

```
	{
		path = new int[V];
		for (int i = 0; i < V; i++)
			path[i] = -1;

		/* Let us put vertex 0 as the first vertex in the path.
		If there is a Hamiltonian Cycle, then the path can be
		started from any point of the cycle as the graph is
		undirected */
		path[0] = 0;
		if (hamCycleUtil(graph, path, 1) == false)
		{
			System.out.println("\nSolution does not exist");
			return 0;
		}

		printSolution(path);
		return 1;
	}

	/* A utility function to print solution */
	void printSolution(int path[])
	{
		System.out.println("Solution Exists: Following" +
						" is one Hamiltonian Cycle");
		for (int i = 0; i < V; i++)
			System.out.print((path[i]+1)+" ");

		// Let us print the first vertex again to show the
		// complete cycle
		System.out.println(" " + (path[0]+1) + " ");
	}

	// driver program to test above function
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
	    System.out.println("enter noof vertices");
	    int V=sc.nextInt();
	    HamiltonianCycle obj=new HamiltonianCycle(V);
	    int graph[][]=new int[V][V];
	    System.out.println("enter adjacency matrix");
	    for(int i=0;i<V;i++)
	      for(int j=0;j<V;j++)
	         graph[i][j]=sc.nextInt();
	// Print the solution
		obj.hamCycle(graph);

	}
}
```

```
/*Write a Java program for Perfect Sum Problem (Print all subsets with given sum)
using sumofsubsets algorithm strategy.
Given an array of integers and a sum, the task is to print all subsets of given
array with sum equal to given sum.
sample output=
Enter the number of elements:6
Enter 6 Elements:
2 3 5 6 8 10
Enter the sum to be obtained:
10
solution set is:
1 1 1
solution set is:
1 0 0 0 1
solution set is:
0 0 0 0 0 1
*/
import java.util.Scanner;
class SumOfSubsets
{
    int[] w;
    int[] x;
    int sum,n;
    int total = 0;
    Scanner sc;
    SumOfSubsets()
    {
        sc = new Scanner(System.in);
        System.out.print("Enter the number of elements:");
        n = sc.nextInt();
        w = new int[n + 1];
        x = new int[n + 1];
    }
    boolean issafe()
     {
        System.out.println("Enter "+n+" Elements:");
        for (int i = 1; i < n + 1; i++) {
            w[i] = sc.nextInt();
            total += w[i];
        }
        System.out.println("Enter the sum to be obtained:");
        sum = sc.nextInt();
        if (total < sum) {
                    return false;
        }
        return true;
    }
    void sumofsubsetUtil(int s, int k, int r) {
        int i = 0;
        x[k] = 1;
        if (s + w[k] == sum) {
            System.out.println("solution set is:");
```

```
            for (i = 1; i <= k; i++) {
                System.out.print(x[i]+" ");
            }
        }
        else if ((s + w[k] + w[k + 1]) <= sum)
        {
            sumofsubsetUtil(s + w[k], k + 1, r - w[k]);
        }
        if ((s + r - w[k]) >= sum && (s + w[k + 1]) <= sum) {
            x[k] = 0;
            sumofsubsetUtil(s, k + 1, r - w[k]);
        }
    }
    void solveSum()
    {
        if(issafe()==false)
          System.out.println("Not possible to obtain the subset!!!");
        sumofsubsetUtil(0, 1, total);
    }

    public static void main(String[] args)
   {
        new SumOfSubsets().solveSum();
    }
}
```

/* Write a java program which implements Greedy Knapsack strategy to obtain the optimal solution.
sample output=
Enter the no. of objects:
4
enter item 1 weights and profit
40
280
enter item 2 weights and profit
10
100
enter item 3 weights and profit
20
120
enter item 4 weights and profit
24
120
Enter the capacity of knapsack:
60
After sorting by pi/wi ratio
p[1]=100.0 w[1]=10.0
p[2]=280.0 w[2]=40.0
p[3]=120.0 w[3]=20.0
p[4]=120.0 w[4]=24.0
The result vector is:
1.0

1.0
0.5
0.0
Maximum profit is:440.0
***/**

```
import java.util.*;
public class Knapsack
{
void knapsack(int n, double weight[], double profit[], double capacity)
{
double x[]=new double[20], tp = 0;
int i, j;
double u =capacity;
for (i = 0; i < n; i++)
x[i] = 0.0;
for (i = 0; i < n; i++) {
if (weight[i] > u)
break;
else {
x[i] = 1.0;
tp = tp + profit[i];
u = u - weight[i];
}
}
if (i < n)
x[i] = u / weight[i];
tp = tp + (x[i] * profit[i]);
System.out.println("The result vector is:");
for (i = 0; i < n; i++)
System.out.println(x[i]+"");
System.out.println("Maximum profit is:"+tp);
}
public static void main(String[] args)
{
double weight[]=new double[20];
double profit[]=new double[20];
int num, i, j;
double capacity;
double ratio[]=new double[20],temp;
Scanner sc=new Scanner(System.in);
System.out.println("Enter the no. of objects:");
num=sc.nextInt();
Knapsack obj=new Knapsack();
for (i = 0; i < num; i++) {
System.out.println("enter item "+(i+1)+" weights and profit");
weight[i]=sc.nextDouble();
profit[i]=sc.nextDouble();
}
System.out.println("Enter the capacity of knapsack:");
capacity=sc.nextDouble();
for (i = 0; i < num; i++) {
ratio[i] = profit[i] / weight[i];
```

```
}
for (i = 0; i < num; i++) {
for (j = i + 1; j < num; j++) {
if (ratio[i] < ratio[j]) {
temp = ratio[j];
ratio[j] = ratio[i];
ratio[i] = temp;

temp = weight[j];
weight[j] = weight[i];
weight[i] = temp;

temp = profit[j];
profit[j] = profit[i];
profit[i] = temp;
}
}
}
System.out.println("After sorting by pi/wi ratio");
for(i=0;i<num;i++)
System.out.println("p["+(i+1)+"]="+profit[i]+" "+"w["+(i+1)+"]="+weight[i]);
obj.knapsack(num, weight, profit, capacity);
}
}

/*
write a java program to implement job seequencing with deadlines
using greedy
sample output=
Enter the no of objects:
4
Enter the deadlines and profits:
2 70
1 12
2 18
1 35
After sorting profits and deadlines are
p[1]=70 d[1]=2
p[2]=35 d[2]=1
p[3]=18 d[3]=2
p[4]=12 d[4]=1
the optimal solution:105
*/
import java.util.*;
class Job
{
int k,i;
Scanner sc=new Scanner(System.in);
int jobseq(int n,int d[],int p[])
{
int r,t;
int j[]=new int[n+1];
```

```
int profit=0;
d[0]=j[0]=0;
k=1;
j[1]=1;
profit+=p[1];
for(int i=2;i<=n;i++)
{
r=k;
while(d[j[r]]>d[i] && d[j[r]]!=r)
r--;
if((d[j[r]] <=d[i]) && (d[i]>r))
{
for(t=k;t>=r+1;t--)
j[t+1]=j[t];
profit+=p[i];
j[r+1]=i;
k++;
}
}
return profit;
}
}
class Jobtest
{
public static void main(String args[])
{
Job obj=new Job();
int optsoln;
int n,i,j,temp;
int p[],d[];
Scanner sc=new Scanner(System.in);
System.out.println("Enter the no of objects:");
n=sc.nextInt();
p=new int[n+1];
d= new int[n+1];
System.out.println("Enter the deadlines and profits:");
for(i=1;i<=n;i++)
{
d[i]=sc.nextInt();
p[i]=sc.nextInt();
}
for (i =1;i<=n;i++) {
for (j = i + 1; j <=n; j++) {
if (p[i] < p[j]) {
temp = p[j];
p[j] =p[i];
p[i] = temp;

temp = d[j];
d[j] =d[i];
d[i] = temp;
}
```

```
}
}
System.out.println("After sorting profits and deadlines are");
for(i=1;i<=n;i++)
System.out.println("p["+i+"]="+p[i]+" "+"d["+i+"]="+d[i]);
optsoln=obj.jobseq(n,d,p);
System.out.println("the optimal solution:"+optsoln);
}
}
```

/*write a java program to find MST using kruskal's methodology
sample output=
enter noof vertices
4
enter noof Edges
5
enter each edge source,destination,weight
0 1 10
0 2 6
0 3 5
1 3 15
2 3 4
Minimum Spanning Tree:
Edge-1 source: 2 destination: 3 weight: 4
Edge-2 source: 0 destination: 3 weight: 5
Edge-3 source: 0 destination: 1 weight: 10
***/**

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.*;

public class KruskalMST
{
    public static class Edge
    {
        int source;
        int destination;
        int weight;

        public Edge(int source, int destination, int weight)
        {
            this.source = source;
            this.destination = destination;
            this.weight = weight;
        }
    }

    public static class Graph {

        int vertices;
        ArrayList<Edge> allEdges = new ArrayList<>();
```

```
    Graph(int vertices)
    {
        this.vertices = vertices;
    }

    public void addEgde(int source, int destination, int weight)
    {
        Edge edge = new Edge(source, destination, weight);
        allEdges.add(edge); //add to total edges
    }

    public void kruskalMST()
    {
        PriorityQueue<Edge> pq = new PriorityQueue<>(allEdges.size(),
Comparator.comparingInt(o -> o.weight));

        //add all the edges to priority queue, //sort the edges on weights
        for (int i = 0; i <allEdges.size() ; i++) {
            pq.add(allEdges.get(i));
        }

        //create a parent []
        int [] parent = new int[vertices];

        //makeset
        makeSet(parent);

        ArrayList<Edge> mst = new ArrayList<>();

        //process vertices - 1 edges
        int index = 0;
        while(index<vertices-1){
            Edge edge = pq.remove();
            //check if adding this edge creates a cycle
            int x_set = find(parent, edge.source);
            int y_set = find(parent, edge.destination);

            if(x_set==y_set){
                //ignore, will create cycle
            }else {
                //add it to our final result
                mst.add(edge);
                index++;
                union(parent,x_set,y_set);
            }
        }
        //print MST
        System.out.println("Minimum Spanning Tree: ");
        printGraph(mst);
    }
```

```
    public void makeSet(int [] parent){
        //Make set- creating a new element with a parent pointer to itself.
        for (int i = 0; i <vertices ; i++) {
            parent[i] = i;
        }
    }

    public int find(int [] parent, int vertex){
        //chain of parent pointers from x upwards through the tree
        // until an element is reached whose parent is itself
        if(parent[vertex]!=vertex)
            return find(parent, parent[vertex]);;
        return vertex;
    }

    public void union(int [] parent, int x, int y){
        int x_set_parent = find(parent, x);
        int y_set_parent = find(parent, y);
        //make x as parent of y
        parent[y_set_parent] = x_set_parent;
    }

    public void printGraph(ArrayList<Edge> edgeList){
        for (int i = 0; i <edgeList.size() ; i++) {
            Edge edge = edgeList.get(i);
            System.out.println("Edge-" + (i+1)+ " source: " + edge.source +
                    " destination: " + edge.destination +
                    " weight: " + edge.weight);
        }
    }
}
public static void main(String[] args) {

        Scanner sc=new Scanner(System.in);
        int source,dest,weight;
        System.out.println("enter noof vertices");
        int vertices = sc.nextInt();
        Graph graph = new Graph(vertices);
        System.out.println("enter noof  Edges");
         int edges=sc.nextInt();
         System.out.println("enter each edge  source,destination,weight");
        for(int i=0;i<edges;i++)
       {
         source=sc.nextInt();
         dest=sc.nextInt();
         weight=sc.nextInt();
         graph.addEgde(source, dest, weight);
           }

        graph.kruskalMST();
    }
}
```

```
/* A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
sample output=
enter noof vertices
4
enter adjacency matrix
0 1 5 4
1 0 3 2
4 3 0 7
5 2 7 0
spanning tree Edges and  Weights
1 - 2 1
2 - 3 3
2 - 4 2
total cost of MST=6
*/
import java.util.*;
import java.lang.*;
import java.io.*;

class MSTPrim
{
	int V;
	  MSTPrim(int v)
	{
	 V=v;
	}
	  // A utility function to find the vertex with minimum key
	  // value, from the set of vertices not yet included in MST
	  int minKey(int key[], Boolean mstSet[])
	  {
		// Initialize min value
		int min = Integer.MAX_VALUE, min_index=-1;

		for (int v = 0; v < V; v++)
			if (mstSet[v] == false && key[v] < min)
			{
				min = key[v];
				min_index = v;
			}

		return min_index;
	  }

	  // A utility function to print the constructed MST stored in
	  // parent[]
	  void printMST(int parent[], int n, int graph[][])
	  {
	     int tcost=0;
		System.out.println("spanning tree Edges and  Weights");
		for (int i = 1; i < V; i++)
		{
```

```
                System.out.println((parent[i]+1)+" - "+(i+1)+" "+graph[i][parent[i]]);
            tcost+=graph[i][parent[i]];
        }
      System.out.println("total cost of MST="+tcost);

    }

    // Function to construct and print MST for a graph represented
    // using adjacency matrix representation
    void primMST(int graph[][])
    {
            // Array to store constructed MST
            int parent[] = new int[V];

            // Key values used to pick minimum weight edge in cut
            int key[] = new int [V];

            // To represent set of vertices not yet included in MST
            Boolean mstSet[] = new Boolean[V];

            // Initialize all keys as INFINITE
            for (int i = 0; i < V; i++)
            {
                    key[i] = Integer.MAX_VALUE;
                    mstSet[i] = false;
            }

            // Always include first 1st vertex in MST.
            key[0] = 0;      // Make key 0 so that this vertex is picked as first vertex
            parent[0] = -1; // First node is always root of MST

            // The MST will have V vertices
            for (int count = 0; count < V-1; count++)
            {
                    // Pick thd minimum key vertex from the set of vertices
                    // not yet included in MST
                    int u = minKey(key, mstSet);

                    // Add the picked vertex to the MST Set
                    mstSet[u] = true;

                    // Update key value and parent index of the adjacent
                    // vertices of the picked vertex. Consider only those
                    // vertices which are not yet included in MST
                    for (int v = 0; v < V; v++)

                            // graph[u][v] is non zero only for adjacent vertices of m
                            // mstSet[v] is false for vertices not yet included in MST
                            // Update the key only if graph[u][v] is smaller than key[v]
                            if (graph[u][v]!=0 && mstSet[v] == false &&
                                    graph[u][v] < key[v])
                            {
```

```
                        parent[v] = u;
                        key[v] = graph[u][v];
                    }
            }

            // print the constructed MST
            printMST(parent, V, graph);
        }

        public static void main (String[] args)
        {

                    int i,j;
          Scanner sc=new Scanner(System.in);
          System.out.println("enter noof vertices");
          int V=sc.nextInt();
          int graph[][] = new int[V][V];
          MSTPrim t = new MSTPrim(V);
          System.out.println("enter adjacency matrix");
                   for(i=0;i<V;i++)
           for(j=0;j<V;j++)
            graph[i][j]=sc.nextInt();
              // Print the solution
              t.primMST(graph);
        }
}
```