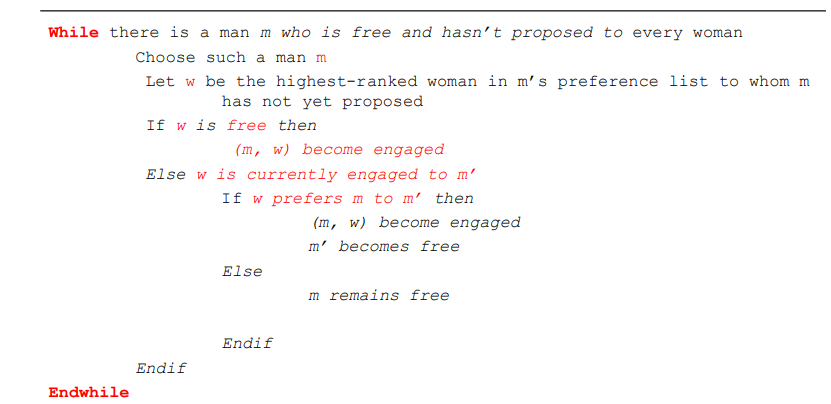
1. **G-S (Stable Matching Problem)**

Input :Reference list of men and women

Output : the set of pairs of men and women

Algorithm :

Program :

// Java program for stable marriage problem

import java.util.\*;

class GFG

{

// Number of Men or Women

static int N = 4;

// This function returns true if woman

// 'w' prefers man 'm1' over man 'm'

static boolean wPrefersM1OverM(int prefer[][], int w,

int m, int m1)

{

// Check if w prefers m over her current engagment m1

for (int i = 0; i < N; i++)

{

// If m1 comes before m in lisr of w,

// then w prefers her current engagement,

// don't do anything

if (prefer[w][i] == m1)

return true;

// If m cmes before m1 in w's list,

// then free her current engagement

// and engage her with m

if (prefer[w][i] == m)

return false;

}

return false;

}

// Prints stable matching for N boys and

// N girls. Boys are numbered as 0 to

// N-1. Girls are numbereed as N to 2N-1.

static void stableMarriage(int prefer[][])

{

// Stores partner of women. This is our

// output array that stores paing information.

// The value of wPartner[i] indicates the partner

// assigned to woman N+i. Note that the woman

// numbers between N and 2\*N-1. The value -1

// indicates that (N+i)'th woman is free

int wPartner[] = new int[N];

// An array to store availability of men.

// If mFree[i] is false, then man 'i' is

// free, otherwise engaged.

boolean mFree[] = new boolean[N];

// Initialize all men and women as free

Arrays.fill(wPartner, -1);

int freeCount = N;

// While there are free men

while (freeCount > 0)

{

// Pick the first free man

// (we could pick any)

int m;

for (m = 0; m < N; m++)

if (mFree[m] == false)

break;

// One by one go to all women

// according to m's preferences.

// Here m is the picked free man

for (int i = 0; i < N &&

mFree[m] == false; i++)

{

int w = prefer[m][i];

// The woman of preference is free,

// w and m become partners (Note that

// the partnership maybe changed later).

// So we can say they are engaged not married

if (wPartner[w - N] == -1)

{

wPartner[w - N] = m;

mFree[m] = true;

freeCount--;

}

else // If w is not free

{

// Find current engagement of w

int m1 = wPartner[w - N];

// If w prefers m over her current engagement m1,

// then break the engagement between w and m1 and

// engage m with w.

if (wPrefersM1OverM(prefer, w, m, m1) == false)

{

wPartner[w - N] = m;

mFree[m] = true;

mFree[m1] = false;

}

} // End of Else

} // End of the for loop that goes

// to all women in m's list

} // End of main while loop

// Print the solution

System.out.println("Woman Man");

for (int i = 0; i < N; i++)

{

System.out.print(" ");

System.out.println(i + N + " " +

wPartner[i]);

}

}

// Driver Code

public static void main(String[] args)

{

int prefer[][] = new int[][]{{4,5,6,7},

{5, 4, 6, 7},

{7,4,6,5},

{7,6,5,4},

{0,1,3,2},

{1,2,0,3},

{2,3,0,1},

{2,3,1,0}};

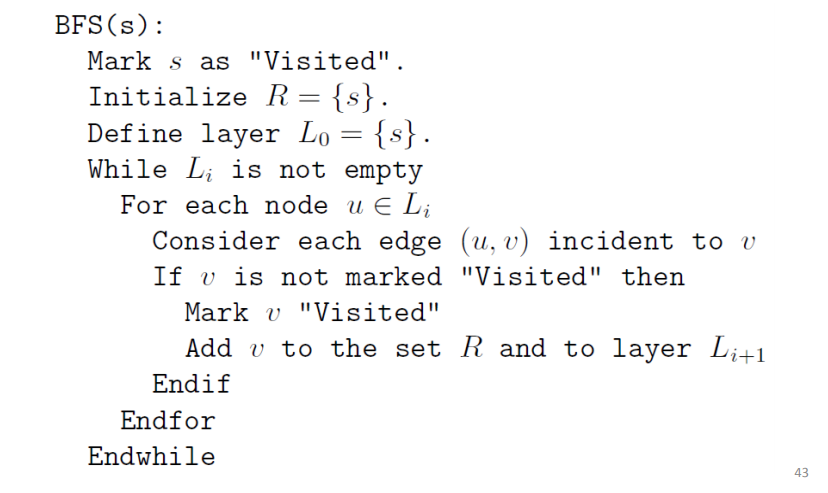
stableMarriage(prefer);

}

}

1. **BFS**

Algorithm:



Program :

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List; class Main{

static class Graph

{

int v;

int e;

int[][] adj;

Graph(int v, int e)

{

this.v = v;

this.e = e;

adj = new int[v][v];

for(int row = 0; row < v; row++)

Arrays.fill(adj[row], 0);

}

void addEdge(int start, int e)

{

// Considering a bidirectional edge

adj[start][e] = 1;

adj[e][start] = 1;

}

// Function to perform BFS on the graph

void BFS(int start)

{

// Visited vector to so that

// a vertex is not visited more than once

// Initializing the vector to false as no

// vertex is visited at the beginning

boolean[] visited = new boolean[v];

Arrays.fill(visited, false);

List<Integer> q = new ArrayList<>();

q.add(start);

// Set source as visited

visited[start] = true;

int vis;

while (!q.isEmpty())

{

vis = q.get(0);

// Print the current node

System.out.print(vis + " "); q.remove(q.get(0));

// For every adjacent vertex to

// the current vertex

for(int i = 0; i < v; i++)

{

if (adj[vis][i] == 1 && (!visited[i]))

{

// Push the adjacent node to

// the queue

q.add(i);

// Set

visited[i] = true;

}

}

}

}

}

// Driver code

public static void main(String[] args)

{

int v = 9, e = 10;

// Create the graph

Graph G = new Graph(v, e); G.addEdge(0, 1);

G.addEdge(0, 1);

G.addEdge(0,8);

G.addEdge(8,2);

G.addEdge(8,6);

G.addEdge(6,5);

G.addEdge(6,7);

G.addEdge(2,5);

G.addEdge(2,4 );

G.addEdge(2,3);

G.addEdge(4,7);

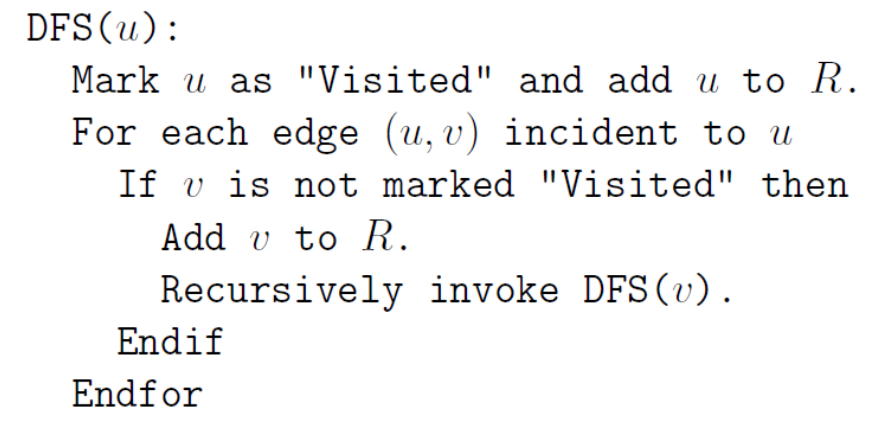
G.BFS(0);

}

}

**3. DFS**

Algorithm :



// Java program to print DFS

//mtraversal from a given given

// graph

import java.io.\*;

import java.util.\*;

// This class represents a

// directed graph using adjacency

// list representation

class Main {

private int V;

boolean visited[];// No. of vertices

// Array of lists for

// Adjacency List Representation

private LinkedList<Integer> adj[];

// Constructor

Main(int v)

{

V = v;

adj = new LinkedList[v];

visited=new boolean[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.

}

// A function used by DFS

void DFSUtil(int v)

{

// Mark the current node as visited and print it

visited[v] = true;

System.out.print(v + " ");

// Recur for all the vertices adjacent to this

// vertex

Iterator<Integer> i = adj[v].listIterator();

while (i.hasNext())

{

int n = i.next();

if (!visited[n])

DFSUtil(n);

}

}

// The function to do DFS traversal.

// It uses recursive

// DFSUtil()

// Driver Code

public static void main(String args[])

{

Main G = new Main(10);

G.addEdge(0, 1);

G.addEdge(0,8);

G.addEdge(8,2);

G.addEdge(8,6);

G.addEdge(6,5);

G.addEdge(6,7);

G.addEdge(2,5);

G.addEdge(2,4 );

G.addEdge(2,3);

G.addEdge(4,7);

System.out.println(

"Following is Depth First Traversal "

+ "(starting from vertex 2)");

G.DFSUtil(0);

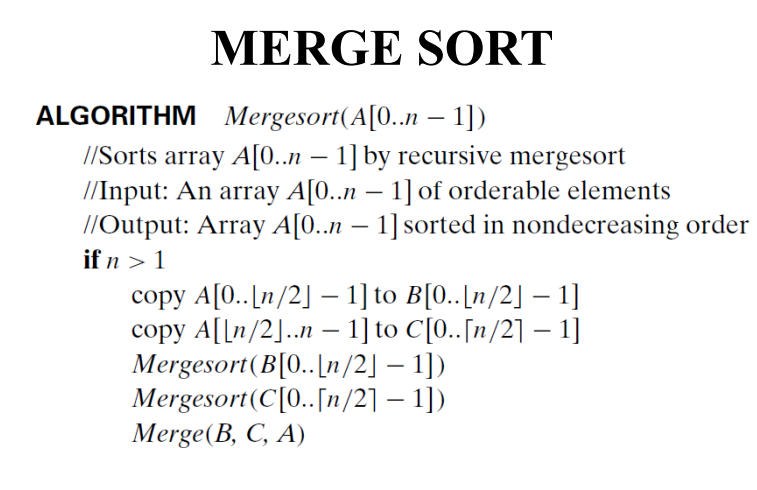
}

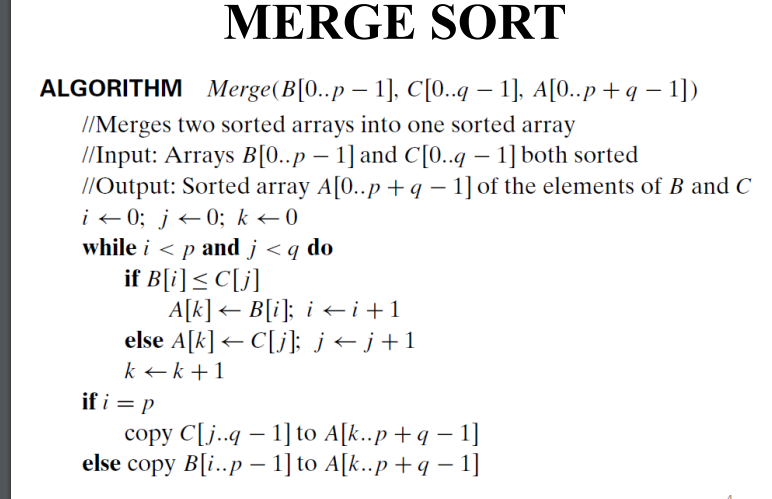
}

// This code is contributed by Aakash Hasija

**4. Merge Sort**

Algorithm:





Program:  
import java.util.Random;

import java.util.Scanner;

public class Merge\_sort

{

public static void main(String[] args)

{

int a[]= new int[200000];

Scanner in = new Scanner(System.in);

long start, end;

System.out.println("MERGE SORT PROGRAM");

System.out.println("Enter the number of elements to be sorted");

int n = in.nextInt();

Random rand= new Random();

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

a[i]=rand.nextInt(100);

System.out.println("Array elements to be sorted are");

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

System.out.print(a[i]+" ");

start=System.nanoTime();

mergesort(a,0,n-1);

end=System.nanoTime();

System.out.println("\nThe sorted elements are");

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

System.out.print(a[i]+" ");

System.out.println("\nThe time taken to sort is "+(end-start)+"ns");

}

static void mergesort(int a[], int low, int high)

{

int mid;

if(low < high)

{

mid = (low+high)/2;

mergesort(a, low, mid);

mergesort(a, mid+1, high);

merge(a, low, mid, high);

}

}

static void merge(int a[], int low, int mid, int high)

{

int i, j, h, k, b[]= new int[100000];

h=low; i=low; j=mid+1;

while((h<=mid) && (j<=high))

{

if(a[h] < a[j])

{

b[i]=a[h];

h=h+1;

}

else

{

b[i] = a[j];

j=j+1;

}

i = i+1;

}

if(h > mid)

{

for(k=j; k<=high; k++)

{

b[i]=a[k];

i= i+1;

}

}

else

{

for(k=h;k<=mid;k++)

{

b[i]=a[k];

i= i+1;

}

}

for(k=low; k<= high; k++)

a[k] = b[k];

}

}

**5. Quick Sort**

**Algorithm :**

/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[pi] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

\* 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 \*/

partition (arr[], low, high)

{

// pivot (Element to be placed at right position)

pivot = arr[high];

i = (low - 1) // Index of smaller element and indicates the

// right position of pivot found so far

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

{

// If current element is smaller than the pivot

if (arr[j] < pivot)

{

i++; // increment index of smaller element

swap arr[i] and arr[j]

}

}

swap arr[i + 1] and arr[high])

return (i + 1)

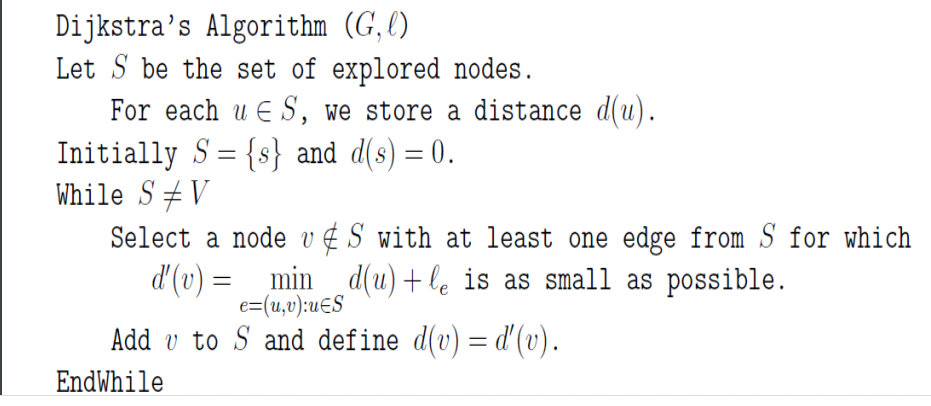
}

**Program:**

|  |
| --- |
| public class q5\_quicksort { |
|  |  |
|  | public static int partition(int arr[],int left,int right){ |
|  | int i=left-1; |
|  | int j=0; |
|  | int pivot=arr[right-1],temp; |
|  |  |
|  | for(j=left; j<right-1; ++j){ |
|  | if(arr[j]<pivot){ |
|  | i++; |
|  | temp=arr[i]; |
|  | arr[i]=arr[j]; |
|  | arr[j]=temp; |
|  |  |
|  | } |
|  | } |
|  |  |
|  | for(j=right-1; j>i+1; --j) |
|  | { |
|  | arr[j]=arr[j-1]; |
|  | } |
|  | arr[i+1] = pivot; |
|  | return i+1; |
|  | } |
|  | public static void quicksort(int[] arr,int left,int right) |
|  | { |
|  | if(right-left<=1) |
|  | { |
|  | return; |
|  | } |
|  |  |
|  | int pivot = partition(arr, left, right); |
|  | //System.out.println(pivot); |
|  |  |
|  | //left half |
|  | quicksort(arr, left, pivot); |
|  | //right half |
|  | quicksort(arr, pivot+1, right); |
|  |  |
|  | } |
|  | public static void main(String args[]){ |
|  | int length = 10; |
|  | int arr[] = new int[length]; |
|  | System.out.println("Before Sorting"); |
|  | for(int i=0; i<length; ++i) |
|  | { |
|  | arr[i] = (int)(Math.random()\*10); |
|  | System.out.print(arr[i]+" "); |
|  | } |
|  | System.out.println(); |
|  | quicksort(arr,0,length); |
|  | System.out.println("\nAfter Sorting"); |
|  | for(int i=0; i<length; ++i){ |
|  | System.out.print(arr[i]+" "); |
|  | } |
|  | System.out.println(); |
|  | } |
|  | } |
|  | /// contributed by sahil |

**6.** **Dijkstra**

**Algorithm:**

****

**Program:**

// A Java program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Main {

// A utility function to find the vertex with minimum distance value,

// from the set of vertices not yet included in shortest path tree

static final int V = 5;

int minDistance(int dist[], Boolean sptSet[])

{

// Initialize min value

int min = Integer.MAX\_VALUE, min\_index = -1;

for (int v = 0; v < V; v++)

if (sptSet[v] == false && dist[v] <= min) {

min = dist[v];

min\_index = v;

}

return min\_index;

}

// A utility function to print the constructed distance array

void printSolution(int dist[], int n)

{

System.out.println("Vertex Distance from Source");

for (int i = 0; i < V; i++)

System.out.println(i + " tt " + dist[i]);

}

// Function that implements Dijkstra's single source shortest path

// algorithm for a graph represented using adjacency matrix

// representation

void dijkstra(int graph[][], int src)

{

int dist[] = new int[V]; // The output array. dist[i] will hold

// the shortest distance from src to i

// sptSet[i] will true if vertex i is included in shortest

// path tree or shortest distance from src to i is finalized

Boolean sptSet[] = new Boolean[V];

// Initialize all distances as INFINITE and stpSet[] as false

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

dist[i] = Integer.MAX\_VALUE;

sptSet[i] = false;

}

// Distance of source vertex from itself is always 0

dist[src] = 0;

// Find shortest path for all vertices

for (int count = 0; count < V - 1; count++) {

// Pick the minimum distance vertex from the set of vertices

// not yet processed. u is always equal to src in first

// iteration.

int u = minDistance(dist, sptSet);

// Mark the picked vertex as processed

sptSet[u] = true;

// Update dist value of the adjacent vertices of the

// picked vertex.

for (int v = 0; v < V; v++)

// Update dist[v] only if is not in sptSet, there is an

// edge from u to v, and total weight of path from src to

// v through u is smaller than current value of dist[v]

if (!sptSet[v] && graph[u][v] != 0 &&

dist[u] != Integer.MAX\_VALUE && dist[u] + graph[u][v] < dist[v])

dist[v] = dist[u] + graph[u][v];

}

// print the constructed distance array

printSolution(dist, V);

}

// Driver method

public static void main(String[] args)

{

/\* Let us create the example graph discussed above \*/

int graph[][] = new int[][] {{0,10,0,0,100},

{10,0,50,0,0},

{0,50,0,20,10},

{0,0,20,0,60},

{100,0,10,60,0} };

Main t = new Main();

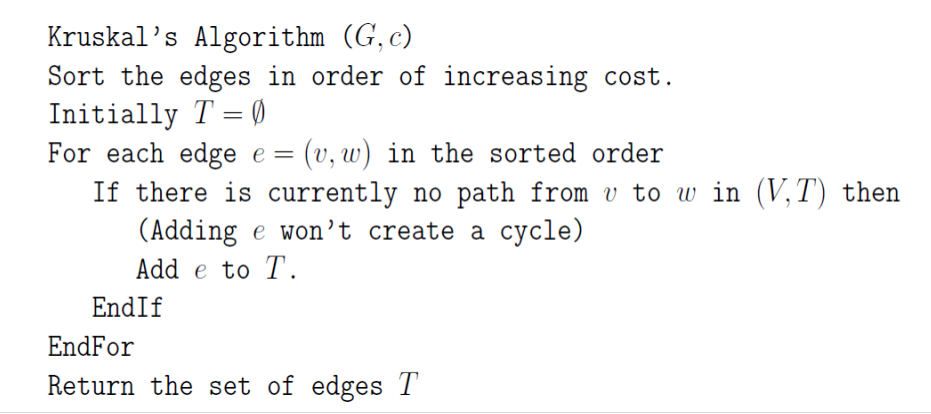
t.dijkstra(graph, 0);

}

}

7 . Kruskal’s

**Alorithm :**

****

**Program:**

// Simple Java implementation for Kruskal's

// algorithm

import java.util.\*;

class Main

{

static int V = 5;

static int[] parent = new int[V];

static int INF = Integer.MAX\_VALUE;

// Find set of vertex i

static int find(int i)

{

while (parent[i] != i)

i = parent[i];

return i;

}

// Does union of i and j. It returns

// false if i and j are already in same

// set.

static void union1(int i, int j)

{

int a = find(i);

int b = find(j);

parent[a] = b;

}

// Finds MST using Kruskal's algorithm

static void kruskalMST(int cost[][])

{

int mincost = 0; // Cost of min MST.

// Initialize sets of disjoint sets.

for (int i = 0; i < V; i++)

parent[i] = i;

// Include minimum weight edges one by one

int edge\_count = 0;

while (edge\_count < V - 1)

{

int min = INF, a = -1, b = -1;

for (int i = 0; i < V; i++)

{

for (int j = 0; j < V; j++)

{

if (find(i) != find(j) && cost[i][j] < min)

{

min = cost[i][j];

a = i;

b = j;

}

}

}

union1(a, b);

System.out.printf("Edge %d:(%d, %d) cost:%d \n",

edge\_count++, a, b, min);

mincost += min;

}

System.out.printf("\n Minimum cost= %d \n", mincost);

}

// Driver code

public static void main(String[] args)

{

int cost[][] = {

{ INF, 10, INF,INF, 5 },

{ 10, INF, 1, 6, INF},

{ INF, 1, INF, 2, 7 },

{ INF, 6, 2, INF, 3 },

{ 5, INF, 7, 3, INF },

};

// Print the solution

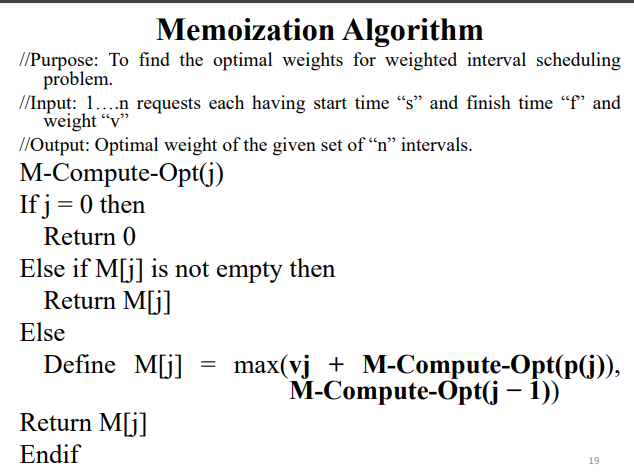
kruskalMST(cost);

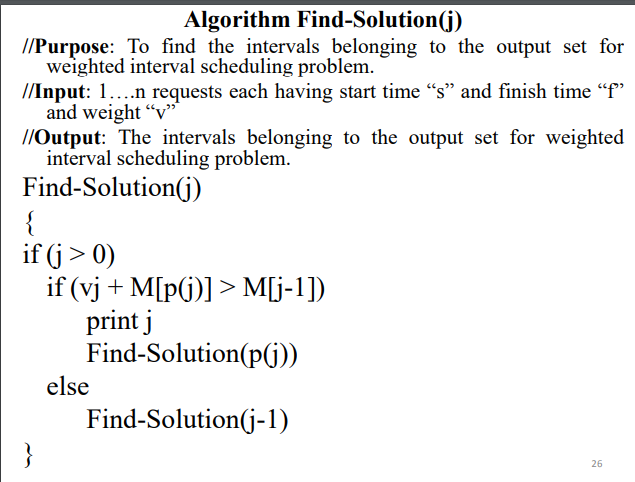
}

}

8**. Weighted Interval Scheduling** (using memoization)

**Algorithm :**

****

**2**

**Program :**

import java.util.\*;

class Request{

int s,f,v;

Request(int s, int f, int v){

this.s = s; // start time

this.f = f; // finish time

this.v = v; // profit or value

}

}

class SortByFinishTime implements Comparator<Request>{

public int compare(Request r1, Request r2) {

return r1.f-r2.f;

}

}

public class WeightedIntervalScheduling

{

static int N;

static ArrayList<Request> arr = new ArrayList<>();

static int[] pred;

static int[] sol;

static void findPred() {

pred = new int[N+1];

for(int i=1;i<N;i++) {

for(int j=i-1;j>=0;j--) {

if(arr.get(j).f<=arr.get(i).s) {

pred[i+1] = j+1;

break;

}

}

}

}

static int solve(int i) {

if(i==0) {

sol[i] = 0;

return 0;

}

if(sol[i]!=-1) return sol[i];

sol[i] = Math.max(arr.get(i-1).v + solve(pred[i]), solve(i-1));

return sol[i];

}

static void findRequests(int i) {

if(i>0) {

if((arr.get(i-1).v + sol[pred[i]]) > sol[i-1]) {

System.out.print(i + " ");

findRequests(pred[i]);

}

else findRequests(i-1);

}

}

public static void main(String[] args)

{

Scanner sc = new Scanner(System.in);

System.out.println("Enter the no. of requests : ");

N = sc.nextInt();

System.out.println("Enter the request no. , start time, finish time and profit : ");

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

int j = sc.nextInt();

int s = sc.nextInt();

int f = sc.nextInt();

int v = sc.nextInt();

arr.add(j-1,new Request(s,f,v));

}

Collections.sort(arr, new SortByFinishTime());

findPred();

sol = new int[N+1];

for(int i=0;i<N+1;i++) sol[i] = -1;

int maxProf = solve(N);

System.out.println("The maximum profit is : " + maxProf);

System.out.println("The requests included in the optimal solution are : ");

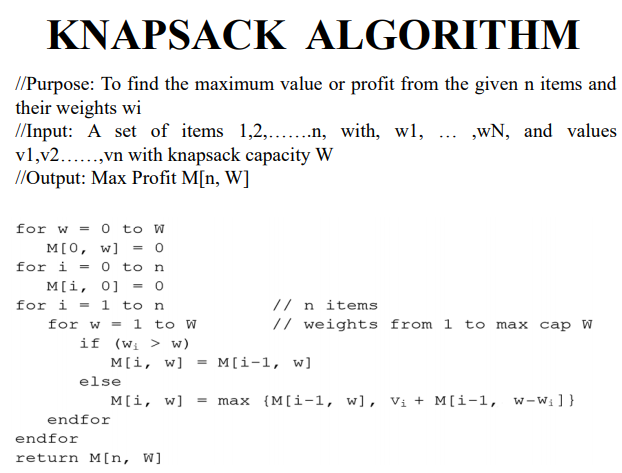
findRequests(N);

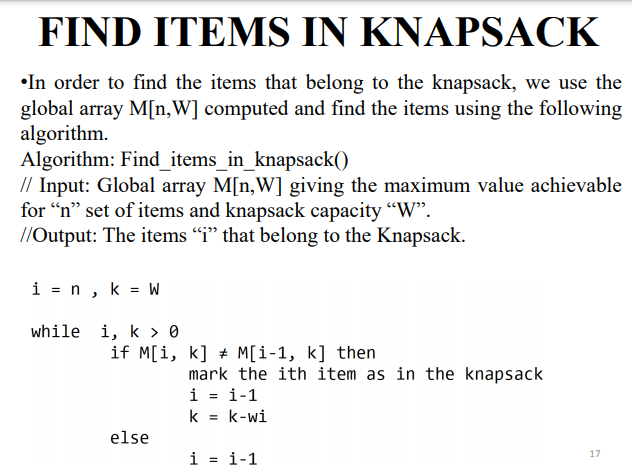
}

}

**9. KnapSack problem**

**Algorithm :**

****

****

**Program:**

// A Dynamic Programming based solution for 0-1 Knapsack problem

class Main {

// A utility function that returns maximum of two integers

static int max(int a, int b)

{ return (a > b) ? 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)

{

int K[][] = new int[n + 1][W + 1];

// builds k[][] according to algorithm we learnt.

for(int w =0;w<=W;w++)

K[0][w]=0;

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

K[i][0]=0;

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

{

for(int w=1;w<=W;w++)// val[i-1] and wt [i-1] are -1 coz of array indexing

{

if(wt[i-1]>w)

K[i][w]=K[i-1][w];

else

K[i][w]=max(K[i-1][w],val[i-1]+K[i-1][w-wt[i-1]]);

}

}

// find items in knapSack

System.out.println("Items in knapSack : \n");

int i=n,k=W;

while(i>0 && k>0)

{

if(K[i][k]!=K[i-1][k])

{

System.out.println(val[i-1]+" "+wt[i-1]);

k=k-wt[i-1];

i=i-1;

}

else

i=i-1;

}

return K[n][W];

}

// Driver program to test above function

public static void main(String args[])

{

int val[] = new int[] { 20,10,30 };

int wt[] = new int[] { 2,2,3 };

int W = 5;

int n = val.length;

System.out.println(knapSack(W, wt, val, n));

}

}

**10. BELLMAN FORD**

**Algorithm :**

**Program :**

// A Java program for Bellman-Ford's single source shortest path

// algorithm.

import java.util.\*;

import java.lang.\*;

import java.io.\*;

class Main{// A class to represent a connected, directed and weighted graph

static class Graph {

// A class to represent a weighted edge in graph

class Edge {

int src, dest, weight;

Edge()

{

src = dest = weight = 0;

}

};

int V, E;

Edge edge[];

// Creates a graph with V vertices and E edges

Graph(int v, int e)

{

V = v;

E = e;

edge = new Edge[e];

for (int i = 0; i < e; ++i)

edge[i] = new Edge();

}

// The main function that finds shortest distances from src

// to all other vertices using Bellman-Ford algorithm. The

// function also detects negative weight cycle

void BellmanFord(Graph graph, int src)

{

int V = graph.V, E = graph.E;

int dist[] = new int[V];

// Step 1: Initialize distances from src to all other

// vertices as INFINITE

for (int i = 0; i < V; ++i)

dist[i] = Integer.MAX\_VALUE;

dist[src] = 0;

// Step 2: Relax all edges |V| - 1 times. A simple

// shortest path from src to any other vertex can

// have at-most |V| - 1 edges

for (int i = 1; i < V; ++i) {

for (int j = 0; j < E; ++j) {

int u = graph.edge[j].src;

int v = graph.edge[j].dest;

int weight = graph.edge[j].weight;

if (dist[u] != Integer.MAX\_VALUE && dist[u] + weight <

dist[v])

dist[v] = dist[u] + weight;

}

}

// Step 3: check for negative-weight cycles. The above

// step guarantees shortest distances if graph doesn't

// contain negative weight cycle. If we get a shorter

// path, then there is a cycle.

for (int j = 0; j < E; ++j) {

int u = graph.edge[j].src;

int v = graph.edge[j].dest;

int weight = graph.edge[j].weight;

if (dist[u] != Integer.MAX\_VALUE && dist[u] + weight < dist[v]) {

System.out.println("Graph contains negative weight cycle");

return;

}

}

printArr(dist, V);

}

// A utility function used to print the solution

void printArr(int dist[], int V)

{

System.out.println("Vertex Distance from Source");

for (int i = 0; i < V; ++i)

System.out.println(i + "\t\t" + dist[i]);

}}

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

System.out.println("Enter the number of vertices: ");

int V = scan.nextInt(); // Number of vertices in graph

System.out.println("Enter the number of edges: ");

int E = scan.nextInt(); // Number of edges in graph

System.out.println("Enter the number of source,dest,weight of eachedge: ");

Graph graph = new Graph(V, E);

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

graph.edge[i].src = scan.nextInt();

graph.edge[i].dest = scan.nextInt();

graph.edge[i].weight = scan.nextInt();

}

long start = System.currentTimeMillis();

graph.BellmanFord(graph, 0);

long finish = System.currentTimeMillis();

long timeElapsed = finish - start;

System.out.println("Time Elapsed: "+timeElapsed+"ms");

}

}