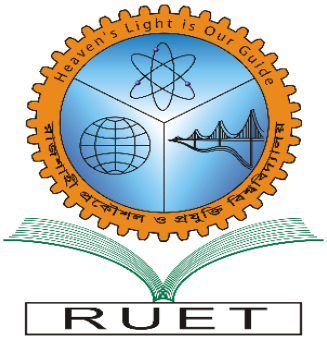


RAJSHAHI UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Course No: CSE 2201

Course Title: Sessional Based on CSE-2201

Submitted to:
Biprodip Pal

Assistant Professor ,
Department of Computer Science and Engineering
Rajshahi University of Engineering and Technology

Submitted by:
Md Al Amin Tokder Shoukhin ,

Roll: 1803078, **Section:** B
Department of Computer Science and Engineering,
Rajshahi University of Engineering and Technology

Problem Name: Dijkstra Algorithm

Source Code (Using BinaryHeap + Adjacency List) :

```
package algorithm;

import java.util.LinkedList;
import java.util.Scanner;

public class dijkstra_heap_1803078{

	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;
		}
	}

	static class HeapNode{

		int vertex;
		int distance;
	}

	static class Graph{

		int vertices;
		LinkedList<Edge>[] adjacencylist;

		Graph(int vertices){
			this.vertices=vertices;
			adjacencylist=new LinkedList[vertices];
			for(int i=0;i<vertices;i++){
				adjacencylist[i]=new LinkedList<>();
			}
		}

		public void addEdge(int source,int destination,int weight){
			Edge edge=new Edge(source,destination,weight);
			adjacencylist[source].addFirst(edge);
```

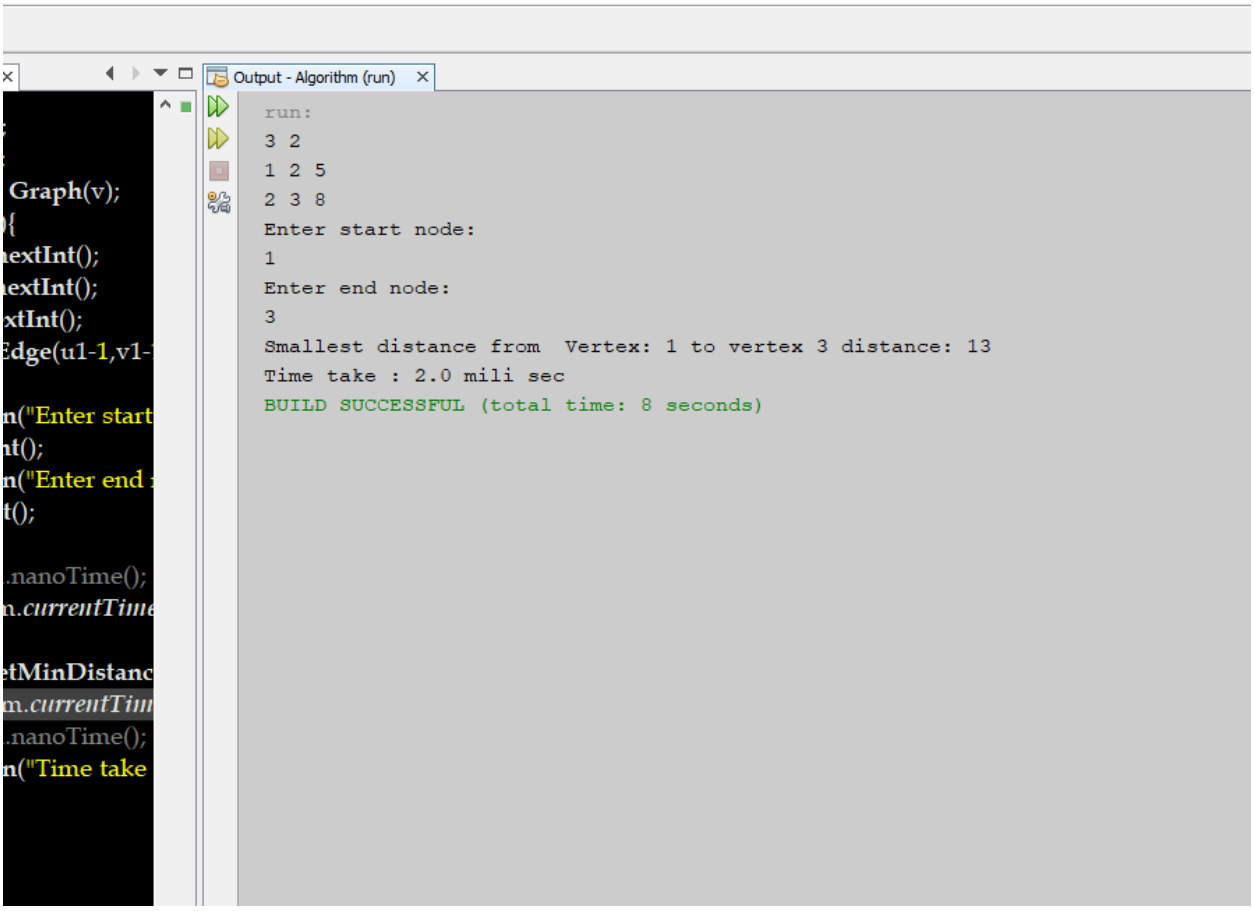
```
                edge=new Edge(destination,source,weight);
                adjacencylist[destination].addFirst(edge); //for undirected graph
        }

        public void dijkstra_GetMinDistances(int sourceVertex,int end){
                int INFINITY=Integer.MAX_VALUE;
                boolean[] SPT=new boolean[vertices];

                HeapNode[] heapNodes=new HeapNode[vertices];
                for(int i=0;i<vertices;i++){
                        heapNodes[i]=new HeapNode();
                        heapNodes[i].vertex=i;
                        heapNodes[i].distance=INFINITY;
                }

                heapNodes[sourceVertex].distance=0;

                MinHeap minHeap=new MinHeap(vertices);
                for(int i=0;i<vertices;i++){
                        minHeap.insert(heapNodes[i]);
                }
                while(!minHeap.isEmpty()){

                        HeapNode extractedNode=minHeap.extractMin();

                        int extractedVertex=extractedNode.vertex;
                        SPT[extractedVertex]=true;

                        LinkedList<Edge> list=adjacencylist[extractedVertex];
                        for(int i=0;i<list.size();i++){
                                Edge edge=list.get(i);
                                int destination=edge.destination;

                                if(SPT[destination]==false){

                                        int
newKey=heapNodes[extractedVertex].distance+edge.weight;
                                        int currentKey=heapNodes[destination].distance;
                                        if(currentKey>newKey){
                                                decreaseKey(minHeap,newKey,destination);
                                                heapNodes[destination].distance=newKey;
                                        }
                                }
                        }
                }

                printDijkstra(heapNodes,sourceVertex,end);
        }
```

```
		public void decreaseKey(MinHeap minHeap,int newKey,int vertex){

			int index=minHeap.indexes[vertex];

			HeapNode node=minHeap.mH[index];
			node.distance=newKey;
			minHeap.bubbleUp(index);
		}

		public void printDijkstra(HeapNode[] resultSet,int sourceVertex,int end){
			System.out.println("Smallest distance from  Vertex: "+(sourceVertex+1)+"
to vertex "+end+" distance: "+resultSet[end-1].distance);

		}
	}

	static class MinHeap{

		int capacity;
		int currentSize;
		HeapNode[] mH;
		int[] indexes;

		public MinHeap(int capacity){
			this.capacity=capacity;
			mH=new HeapNode[capacity+1];
			indexes=new int[capacity];
			mH[0]=new HeapNode();
			mH[0].distance=Integer.MIN_VALUE;
			mH[0].vertex=-1;
			currentSize=0;
		}

		public void insert(HeapNode x){
			currentSize++;
			int idx=currentSize;
			mH[idx]=x;
			indexes[x.vertex]=idx;
			bubbleUp(idx);
		}

		public void bubbleUp(int pos){
			int parentIdx=pos/2;
			int currentIdx=pos;
			while(currentIdx>0&&mH[parentIdx].distance>mH[currentIdx].distance){
				HeapNode currentNode=mH[currentIdx];
				HeapNode parentNode=mH[parentIdx];
```

```
                indexes[currentNode.vertex]=parentIdx;
                indexes[parentNode.vertex]=currentIdx;
                swap(currentIdx,parentIdx);
                currentIdx=parentIdx;
                parentIdx=parentIdx/2;
            }
        }

        public HeapNode extractMin(){
            HeapNode min=mH[1];
            HeapNode lastNode=mH[currentSize];

            indexes[lastNode.vertex]=1;
            mH[1]=lastNode;
            mH[currentSize]=null;
            sinkDown(1);
            currentSize--;
            return min;
        }

        public void sinkDown(int k){
            int smallest=k;
            int leftChildIdx=2*k;
            int rightChildIdx=2*k+1;

if(leftChildIdx<heapSize()&&mH[smallest].distance>mH[leftChildIdx].distance){
                smallest=leftChildIdx;
            }

if(rightChildIdx<heapSize()&&mH[smallest].distance>mH[rightChildIdx].distance){
                smallest=rightChildIdx;
            }
            if(smallest!=k){

                HeapNode smallestNode=mH[smallest];
                HeapNode kNode=mH[k];

                indexes[smallestNode.vertex]=k;
                indexes[kNode.vertex]=smallest;
                swap(k,smallest);
                sinkDown(smallest);
            }
        }

        public void swap(int a,int b){
            HeapNode temp=mH[a];
            mH[a]=mH[b];
```

```
                        mH[b]=temp;
                }

                public boolean isEmpty(){
                        return currentSize==0;
                }

                public int heapSize(){
                        return currentSize;
                }
        }

        public static void main(String[] args){
                Scanner ob=new Scanner(System.in);

                int v=ob.nextInt();
                int e=ob.nextInt();
                Graph graph=new Graph(v);
                for(int i=0;i<e;i++){
                        int u1=ob.nextInt();
                        int v1=ob.nextInt();
                        int c=ob.nextInt();
                        graph.addEdge(u1-1,v1-1,c);
                }
                System.out.println("Enter start node: ");
                int start=ob.nextInt();
                System.out.println("Enter end node: ");
                int end=ob.nextInt();

                //long t1=System.nanoTime();
                double t11=System.currentTimeMillis();

                graph.dijkstra_GetMinDistances(start-1,end);
                double  t22=System.currentTimeMillis();
                //long t2=System.nanoTime();
                System.out.println("Time take : "+(t22-t11)+" mili sec");

        }
}
```

Sample Input and Output:

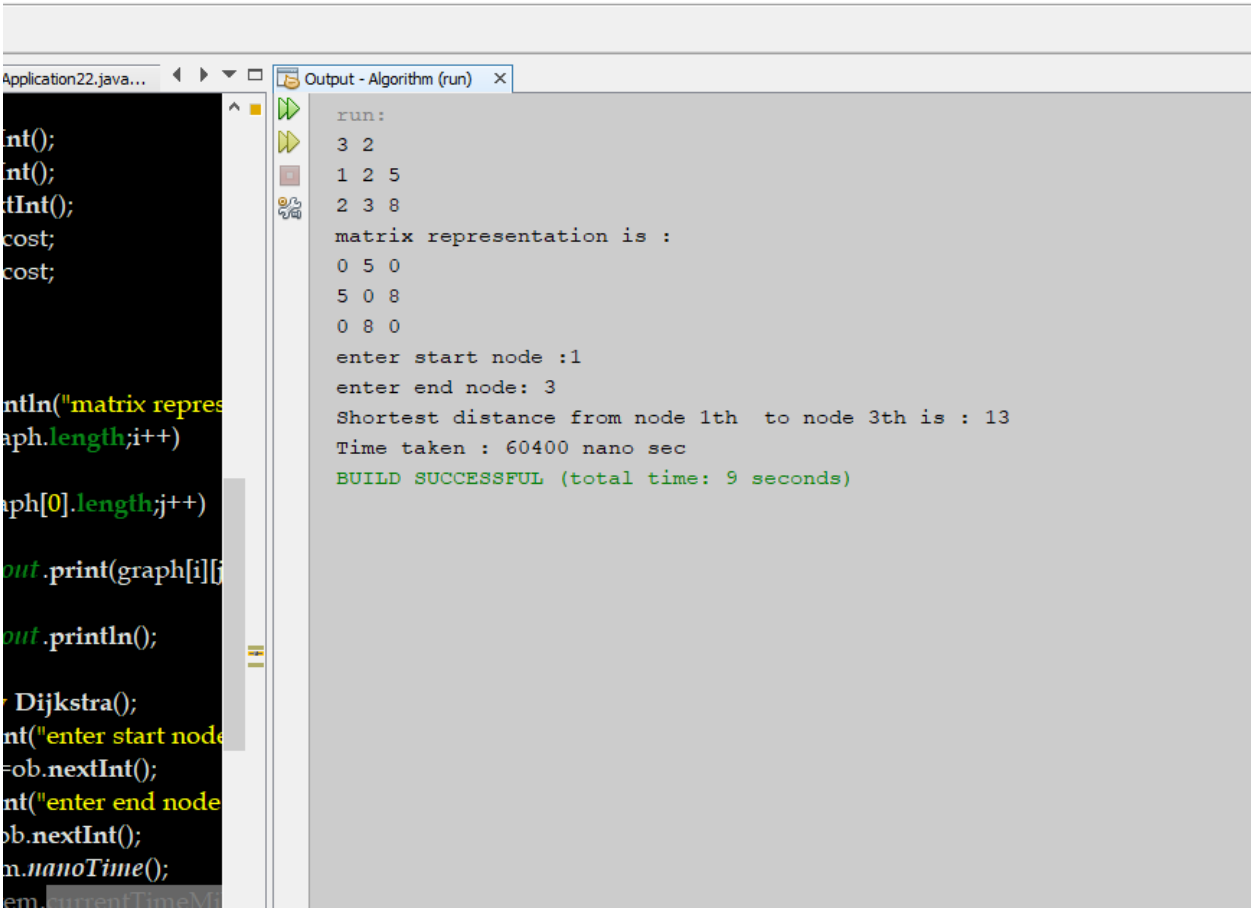


Source Code (Using Priority Queue + Adjacency Matrix) :

```
package algorithm;

import java.util.Scanner;

public class Dijkstra{

	public static void dijkstra(int[][] graph,int source,int start_node,int end_node){
		int count=graph.length;
		boolean[] visitedVertex=new boolean[count+1+1];
		int[] distance=new int[count];
		for(int i=1;i<count;i++){
			visitedVertex[i]=false;
			distance[i]=Integer.MAX_VALUE;
		}


		distance[source]=0;
		for(int i=1;i<count;i++){
			int u=findMinDistance(distance,visitedVertex);
			visitedVertex[u]=true;
			for(int v=1;v<count;v++){

	if(!visitedVertex[v]&&graph[u][v]!=0&&(distance[u]+graph[u][v]<distance[v])){
						distance[v]=distance[u]+graph[u][v];
				}
			}
		}
		System.out.println("Shortest distance from node "+start_node+"th  to node "+
end_node+"th is : "+distance[end_node]);

	}

	private static int findMinDistance(int[] distance,boolean[] visitedVertex){
		int minDistance=Integer.MAX_VALUE;
		int minDistanceVertex=-1;
		for(int i=1;i<distance.length;i++){
			if(!visitedVertex[i]&&distance[i]<minDistance){
				minDistance=distance[i];
				minDistanceVertex=i;
			}
		}
		return minDistanceVertex;
	}

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

```
                Scanner ob=new Scanner (System.in);
                int v=ob.nextInt();
                int e=ob.nextInt();
        int graph[][]=new int[v+1][v+1];


                for(int j=0;j<e;j++)
                {
                int v1=ob.nextInt();
                int v2=ob.nextInt();
                int cost=ob.nextInt();
                graph[v1][v2]=cost;
                graph[v2][v1]=cost;

                }

                System.out.println("matrix representation is : ");
                for(int i=1;i<graph.length;i++)
                {
                for(int j=1;j<graph[0].length;j++)
                {
                        System.out.print(graph[i][j]+" ");
                }
                        System.out.println();
                }
                Dijkstra T=new Dijkstra();
                System.out.print("enter start node :");
                int start_node =ob.nextInt();
                System.out.print("enter end node: ");
                int end_node=ob.nextInt();
                long t11=System.nanoTime();
                //long t1=System.currentTimeMillis();
                T.dijkstra(graph,start_node,start_node,end_node);
                //long t2=System.currentTimeMillis();
                long t22=System.nanoTime();
                System.out.println("Time taken : "+(t22-t11)+" nano sec");
        }
}
```

Sample output:



Source Code (Using Priority Queue + Adjacency List) :

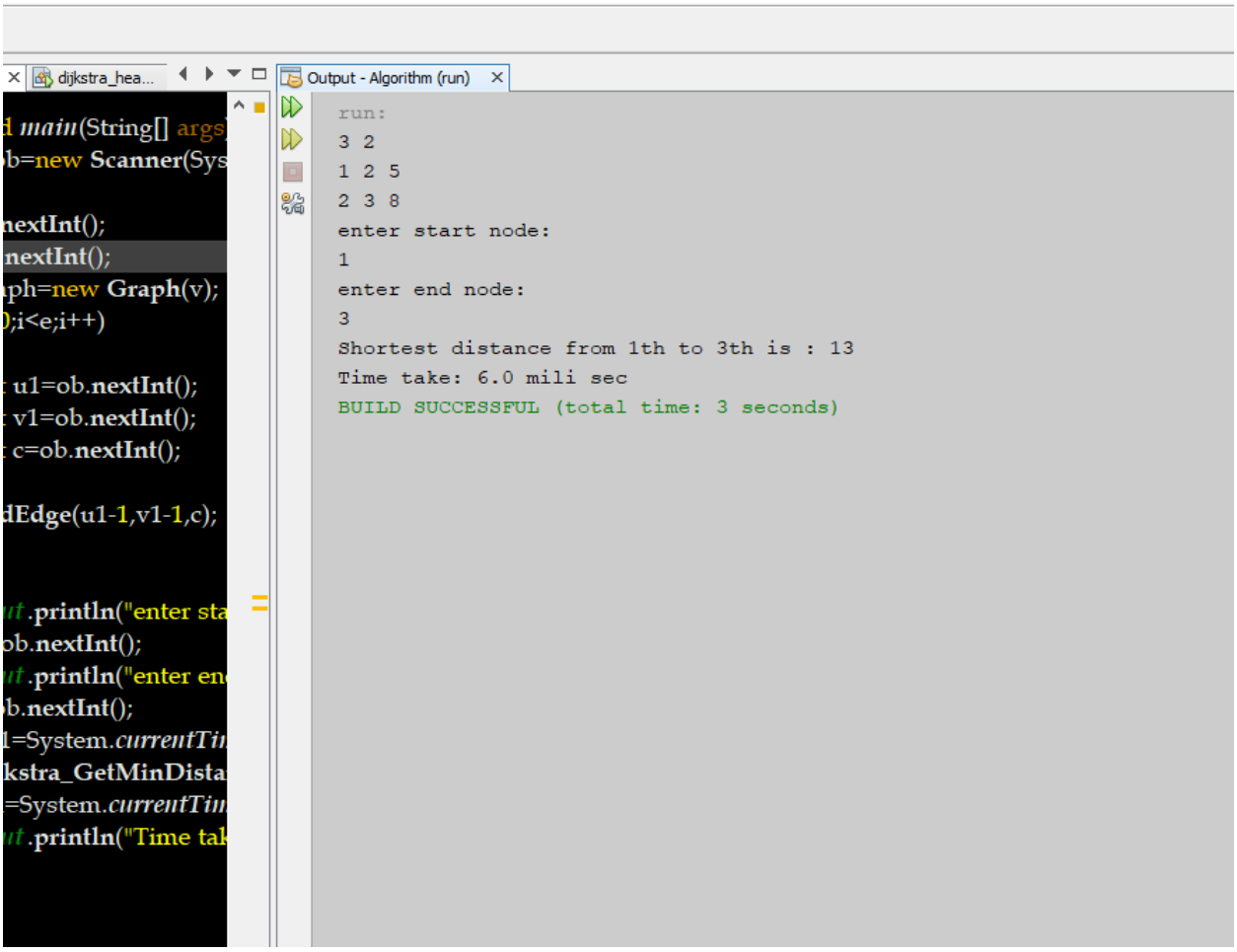
```
package algorithm;

import javafx.util.Pair;
import java.util.Comparator;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Scanner;

public class Dijksta_PQ{

	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;
		}
	}

	static class Graph{

		int vertices;
		static LinkedList<Edge>[] adjacencylist;

		Graph(int vertices){
			this.vertices=vertices;
			adjacencylist=new LinkedList[vertices];

			for(int i=0;i<vertices;i++){
				adjacencylist[i]=new LinkedList<>();
			}

		}

		public void addEdge(int source,int destination,int weight){
			Edge edge=new Edge(source,destination,weight);
			adjacencylist[source].addFirst(edge);

			edge=new Edge(destination,source,weight);
```

```
                adjacencylist[destination].addFirst(edge);
        }



        public void dijkstra_GetMinDistances(int sourceVertex,int start,int end){

                boolean[] SPT=new boolean[vertices];

                int[] distance=new int[vertices];


                for(int i=0;i<vertices;i++){
                        distance[i]=Integer.MAX_VALUE;
                }

                PriorityQueue<Pair<Integer,Integer>> pq=new
PriorityQueue<>(vertices,new Comparator<Pair<Integer,Integer>>(){

                        public int compare(Pair<Integer,Integer> p1,Pair<Integer,Integer>
p2){

                                int key1=p1.getKey();
                                int key2=p2.getKey();
                                return key1-key2;
                        }
                });

                distance[0]=0;
                Pair<Integer,Integer> p0=new Pair<>(distance[0],0);

                pq.offer(p0);

                while(!pq.isEmpty()){

                        Pair<Integer,Integer> extractedPair=pq.poll();

                        int extractedVertex=extractedPair.getValue();
                        if(SPT[extractedVertex]==false){
                                SPT[extractedVertex]=true;

                                LinkedList<Edge> list=adjacencylist[extractedVertex];
                                for(int i=0;i<list.size();i++){
                                        Edge edge=list.get(i);
                                        int destination=edge.destination;

                                        if(SPT[destination]==false){
```

```
                                                int
newKey=distance[extractedVertex]+edge.weight;
                                                int currentKey=distance[destination];
                                                if(currentKey>newKey){
                                                        Pair<Integer,Integer> p=new
Pair<>(newKey,destination);
                                                        pq.offer(p);
                                                        distance[destination]=newKey;
                                                }
                                        }
                                }
                        }
                }
                printDijkstra(distance,sourceVertex,start,end);
        }

        public void printDijkstra(int[] distance,int sourceVertex,int start,int end){

                System.out.println("Shortest distance from "+start+"th to "+end+"th is :
"+distance[end-1]);

        }

        public static void main(String[] args){
                Scanner ob=new Scanner(System.in);

                int v=ob.nextInt();
                int  e=ob.nextInt();
                Graph graph=new Graph(v);
                for(int i=0;i<e;i++)
                {
                        int u1=ob.nextInt();
                        int v1=ob.nextInt();
                        int c=ob.nextInt();

                graph.addEdge(u1-1,v1-1,c);
                }

                System.out.println("enter start node:");
                int start=ob.nextInt();
                System.out.println("enter end node:");
                int end=ob.nextInt();
                double  t1=System.currentTimeMillis();
                graph.dijkstra_GetMinDistances(start-1,start,end);
                double t2=System.currentTimeMillis();
                System.out.println("Time take: "+(t2-t1)+" mili sec ");

        }
```

```
        }
}
```

Sample output:

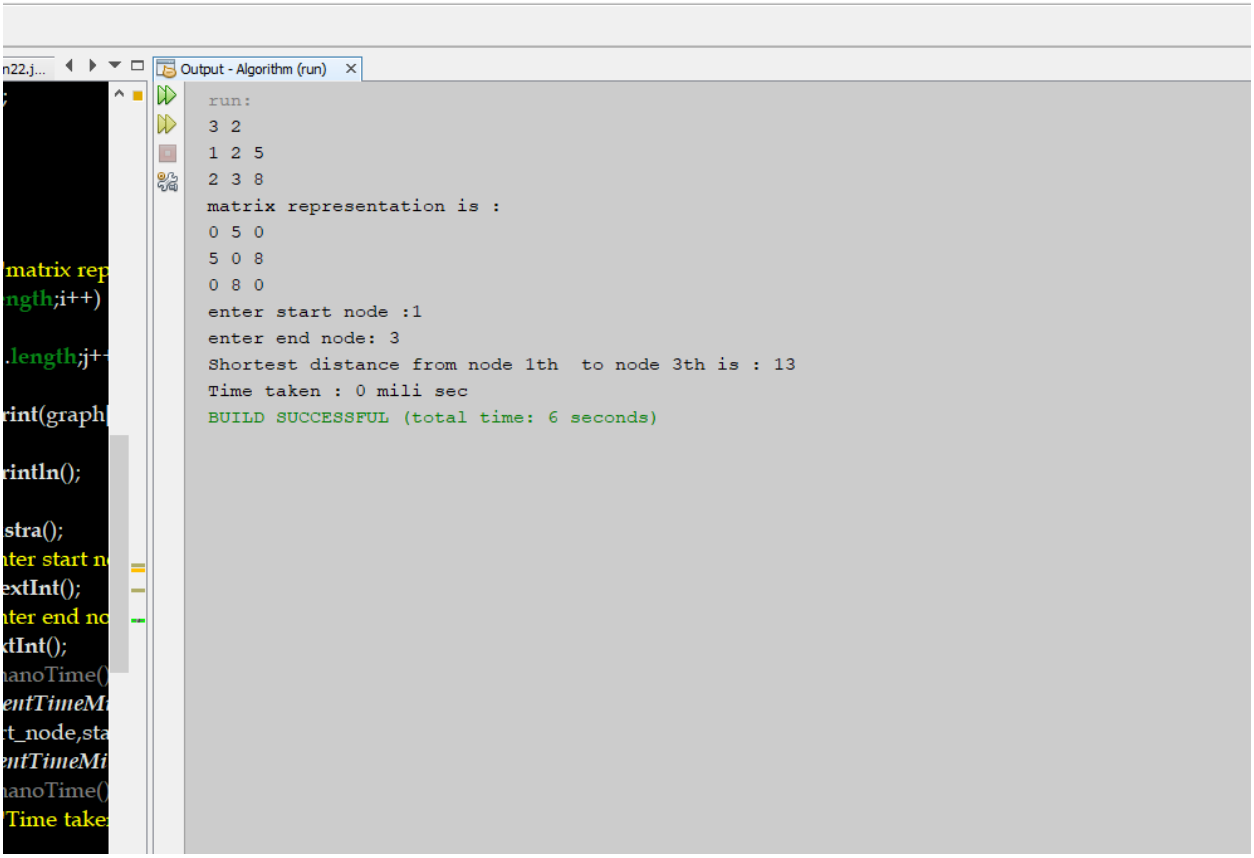


Source Code (Using Unsorted + Adjacency Matrix) :

```
package algorithm;

import java.util.Scanner;

public class Dijkstra{

	public static void dijkstra(int[][] graph,int source,int start_node,int end_node){
		int count=graph.length;
		boolean[] visitedVertex=new boolean[count+1+1];
		int[] distance=new int[count];
		for(int i=1;i<count;i++){
			visitedVertex[i]=false;
			distance[i]=Integer.MAX_VALUE;
		}


		distance[source]=0;
		for(int i=1;i<count;i++){
			int u=findMinDistance(distance,visitedVertex);
			visitedVertex[u]=true;
			for(int v=1;v<count;v++){

	if(!visitedVertex[v]&&graph[u][v]!=0&&(distance[u]+graph[u][v]<distance[v])){
						distance[v]=distance[u]+graph[u][v];
				}
			}
		}
		System.out.println("Shortest distance from node "+start_node+"th  to node "+
end_node+"th is : "+distance[end_node]);

	}

	private static int findMinDistance(int[] distance,boolean[] visitedVertex){
		int minDistance=Integer.MAX_VALUE;
		int minDistanceVertex=-1;
		for(int i=1;i<distance.length;i++){
			if(!visitedVertex[i]&&distance[i]<minDistance){
				minDistance=distance[i];
				minDistanceVertex=i;
			}
		}
		return minDistanceVertex;
```

```
        }

        public static void main(String[] args){
                Scanner ob=new Scanner (System.in);
                int v=ob.nextInt();
                int e=ob.nextInt();
        int graph[][]=new int[v+1][v+1];


                for(int j=0;j<e;j++)
                {
                int v1=ob.nextInt();
                int v2=ob.nextInt();
                int cost=ob.nextInt();
                graph[v1][v2]=cost;
                graph[v2][v1]=cost;

                }

                System.out.println("matrix representation is : ");
                for(int i=1;i<graph.length;i++)
                {
                for(int j=1;j<graph[0].length;j++)
                {
                        System.out.print(graph[i][j]+" ");
                }
                        System.out.println();
                }
                Dijkstra T=new Dijkstra();
                System.out.print("enter start node :");
                int start_node =ob.nextInt();
                System.out.print("enter end node: ");
                int end_node=ob.nextInt();
                //long t11=System.nanoTime();
               long t1=System.currentTimeMillis();
                T.dijkstra(graph,start_node,start_node,end_node);
                long t2=System.currentTimeMillis();
                //long t22=System.nanoTime();
                System.out.println("Time taken : "+(t2-t1)+" mili sec");
        }
}
```

Sample output:



Performance table:(for vertex=100)

Priority Queue Data Structure	Adjacency list	Adjacency matrix
Binary Heap	0.12 mili sec	2.103 mili sec
Unsorted Array	1.001 mili sec	2.989 mili sec

Problem : Kruskals Algorithm

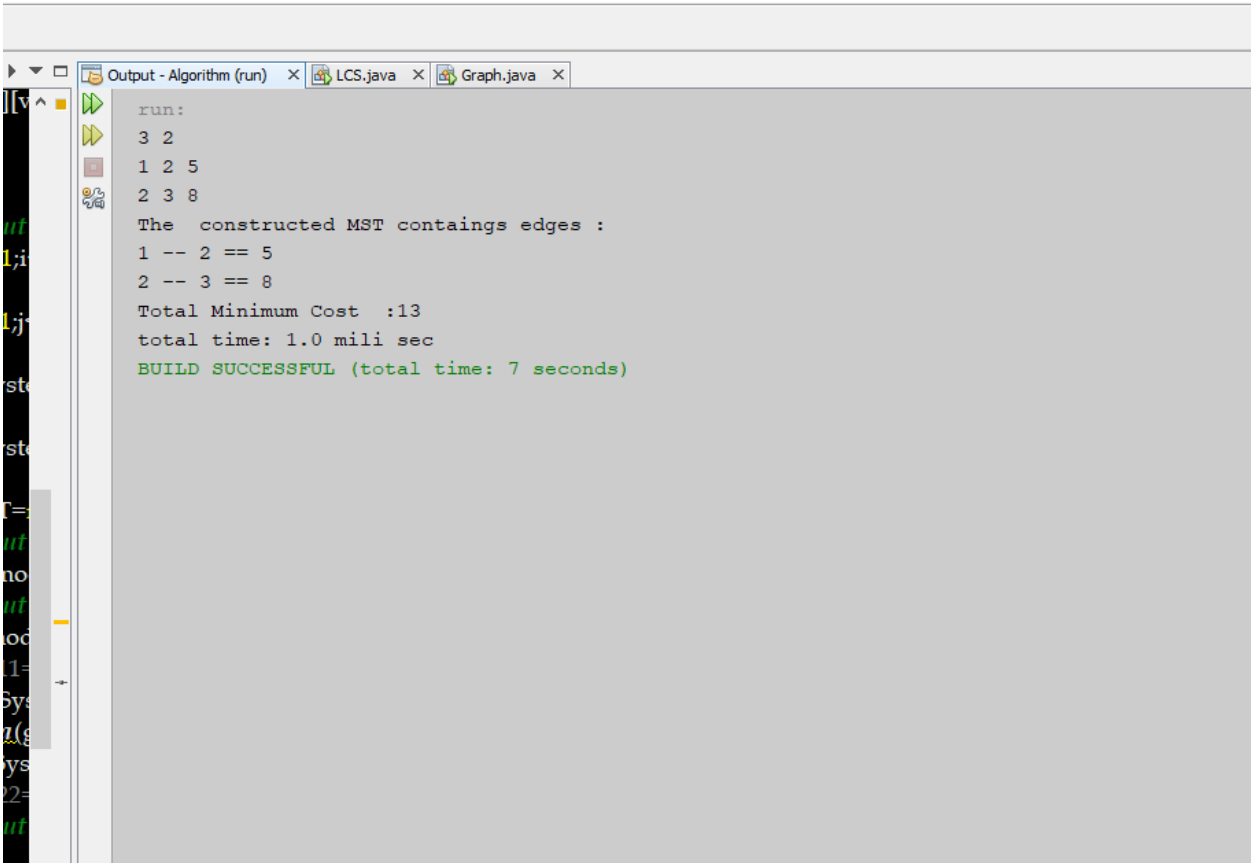
Source Code:

```
package algorithm;


import java.util.*;
import java.lang.*;

import java.io.*;

class Graph{

	class Edge implements Comparable<Edge>{

		int src, dest, weight;

		public int compareTo(Edge compareEdge){
			return this.weight-compareEdge.weight;
		}
	};

	class subset{

		int parent, rank;
	};
	int V, E;
	Edge edge[];

	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(subset subsets[],int i){

		if(subsets[i].parent!=i){
			subsets[i].parent=find(subsets,subsets[i].parent);
```

```
        }
        return subsets[i].parent;
    }

    void Union(subset subsets[],int x,int y){
        int xroot=find(subsets,x);
        int yroot=find(subsets,y);

        if(subsets[xroot].rank<subsets[yroot].rank){
            subsets[xroot].parent=yroot;
        } else if(subsets[xroot].rank>subsets[yroot].rank){
            subsets[yroot].parent=xroot;
        } else{
            subsets[yroot].parent=xroot;
            subsets[xroot].rank++;
        }
    }

    void KruskalMST(){
        Edge result[]=new Edge[V];
        int e=0;
        int i=0;
        for(i=0;i<V;++i){
            result[i]=new Edge();
        }

        Arrays.sort(edge);

        subset subsets[]=new subset[V];
        for(i=0;i<V;++i){
            subsets[i]=new subset();
        }

        for(int v=0;v<V;++v){
            subsets[v].parent=v;
            subsets[v].rank=0;
        }
        i=0;
        while(e<V-1){

            Edge next_edge=new Edge();
            next_edge=edge[i++];
            int x=find(subsets,next_edge.src);
            int y=find(subsets,next_edge.dest);

            if(x!=y){
                result[e++]=next_edge;
                Union(subsets,x,y);
```

```
                        }

                }
                int minimumCost=0;
                System.out.println("The  constructed MST contaings edges : ");
                for(i=0;i<e;++i){
                        System.out.println((result[i].src+1)+" -- "+(result[i].dest+1)+" ==
"+result[i].weight);
                        minimumCost+=result[i].weight;
                }
                System.out.println("Total Minimum Cost  :"+minimumCost);
        }

        public static void main(String[] args){

                Scanner ob=new Scanner(System.in);
                int V=ob.nextInt();
                int E=ob.nextInt();
                Graph graph=new Graph(V,E);
                for(int i=0;i<E;i++){
                        int u1=ob.nextInt();
                        int v1=ob.nextInt();
                        int c=ob.nextInt();
                        graph.edge[i].src=u1-1;
                        graph.edge[i].dest=v1-1;
                        graph.edge[i].weight=c;

                }

                double t1=System.currentTimeMillis();
                graph.KruskalMST();
                double t2=System.currentTimeMillis();
                System.out.println("total time: "+(t2-t1)+" mili sec");
        }
}
```

Output:


Output - Algorithm (run)
LCS.java
Graph.java
run:
3 2
1 2 5
2 3 8
The constructed MST contaings edges :
1 -- 2 == 5
2 -- 3 == 8
Total Minimum Cost :13
total time: 1.0 mili sec
BUILD SUCCESSFUL (total time: 7 seconds)

Performance table:(For Vertex >=100)

V	E	Time (path compression)	Time(Without path compression)	Solution(total cost)
100	100	3.580	2.106	4430
100	80	7.604	3.341	3589
100	50	6.616	5.565	2711

Problem: Longest Common Subsequence (Dynamic programming)

Source Code:

```
package algorithm;

import java.util.Scanner;

public class LCS_1803078 {
	public static void main(String[] args){
		Scanner ob=new Scanner(System.in);
		String A=ob.next();
		String B=ob.next();
		int m=A.length(), n=B.length();
		int[][] ans=new int[m+1][n+1];
		for(int i=0;i<m;i++){
			ans[i][0]=0;

		}
		for(int j=0;j<n;j++){
			ans[0][j]=0;

		}
		for(int i=1;i<=m;i++){
			for(int j=1;j<=n;j++){
				if(A.charAt(i-1)==B.charAt(j-1)){
					ans[i][j]=ans[i-1][j-1]+1;
				}else{
					ans[i][j]=Math.max(ans[i-1][j],ans[i][j-1]);
				}
			}
		}
```

```
                System.out.println(ans[m][n]);
        }
}
```

Output: