Dijkstra: Shortest Reach 2

by pranav9413

Given a graph consisting N nodes (labelled 1 to N) where a specific given node S represents the starting position S and an edge between two nodes is of a given length, which may or may not be equal to other lengths in the graph.

It is required to calculate the shortest distance from the start position (Node S) to all of the other nodes in the graph.

Note: If a node is unreachable , the distance is assumed as -1.

Input Format

The first line contains T, denoting the number of test cases.
First line of each test case has two integers N, denoting the number of nodes in the graph and M, denoting the number of edges in the graph.

The next M lines each consist of three space-separated integers $x\ y\ r$, where x and y denote the two nodes between which the **undirected** edge exists, r denotes the length of edge between these corresponding nodes.

The last line has an integer S, denoting the starting position.

Constraints
$1 \leq T \leq 10$
$2 \leq N \leq 3000$
$1 \leq M \leq \frac{N \times (N-1))}{2}$
$1 \leq x, y, S \leq N$
$1 \leq r \leq 10^5$

If there are edges between the same pair of nodes with different weights, they are to be considered as is, like multiple edges.

Output Format

For each of the T test cases, print a single line consisting $N-1$ space separated integers denoting the shortest distance of $N-1$ nodes other than S from starting position S in increasing order of their labels.

For unreachable nodes, print -1.

Sample Input

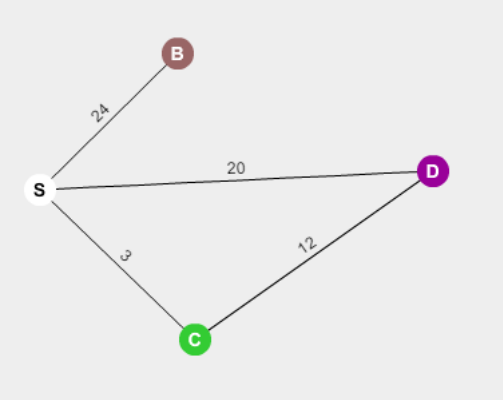
```
1
4 4
1 2 24
1 4 20
3 1 3
4 3 12
1
```

Sample Output

```
24 3 15
```

Explanation

The graph given in the test case is shown as :



- The straight line is a weighted edge, denoting length of edge between the corresponding nodes.
- The nodes S,B,C and D denote the obvious node 1,2,3 and 4 in the test case.

The shortest paths followed for the three nodes B,C and D are as follows :

S->B - Shortest Path Value : **24**

S->C - Shortest Path Value : **3**

S->C->D - Shortest Path Value : **15**

Submissions: 13871
Max Score: 60
Difficulty: Hard

Rate This Challenge:

More

Current Buffer (saved locally, editable) Java 8

```
import java.io.*;
import java.util.*;
import java.text.*;
import java.math.*;
import java.util.regex.*;


    class BinaryMinHeap{

        private List<Node> allNodes = new ArrayList<>();
        private Map<Integer,Integer> nodePosition = new HashMap<>();

        public BinaryMinHeap(){

        }


        public boolean containsData(int key){
            return nodePosition.containsKey(key);
        }

        public void add(int weight, int key){

            Node node = new Node();

            node.weight = weight;
            node.key = key;
            allNodes.add(node);

```

```
            int nodeSize = allNodes.size();
            int currentPosition = nodeSize - 1;

            int parentIndex = (currentPosition - 1) / 2;
            nodePosition.put(node.key, currentPosition);

            while(parentIndex >= 0){

                Node parentNode = allNodes.get(parentIndex);
                Node currentNode = allNodes.get(currentPosition);

                if(parentNode.weight > currentNode.weight){
                    swap(parentNode,currentNode);
                    updatePositionMap(parentNode.key,currentNode.key,parentIndex,currentPosition);
                    currentPosition = parentIndex;
                    parentIndex = (parentIndex-1)/2;                }
                else{
                    break;
                }

            }
        }

        public int min(){
            return allNodes.get(0).key;
        }

        public boolean empty(){
            return allNodes.size() == 0;
        }

        public void decrease(int data, int newWeight){
                Integer position = nodePosition.get(data);
                allNodes.get(position).weight = newWeight;
                int parent = (position -1 )/2;
                while(parent >= 0){
                    if(allNodes.get(parent).weight > allNodes.get(position).weight){
                        swap(allNodes.get(parent), allNodes.get(position));
                        updatePositionMap(allNodes.get(parent).key,allNodes.get(position).key,parent,position);
                        position = parent;
                        parent = (parent-1)/2;
                    }else{
                        break;
                    }
                }
            }

        public Integer getWeight(int key) {
            Integer position = nodePosition.get(key);
            if( position == null ) {
                return null;
            } else {
                return allNodes.get(position).weight;
            }
        }


        public Node extractMinNode() {
            int size = allNodes.size() -1;
            Node minNode = new Node();
            minNode.key = allNodes.get(0).key;
            minNode.weight = allNodes.get(0).weight;

            int lastNodeWeight = allNodes.get(size).weight;
            allNodes.get(0).weight = lastNodeWeight;
            allNodes.get(0).key = allNodes.get(size).key;
            nodePosition.remove(minNode.key);
            nodePosition.remove(allNodes.get(0));
            nodePosition.put(allNodes.get(0).key, 0);
            allNodes.remove(size);

            int currentIndex = 0;
            size--;
```

```
            while(true){
                int left = 2*currentIndex + 1;
                int right = 2*currentIndex + 2;
                if(left > size){
                    break;
                }
                if(right > size){
                    right = left;
                }
                int smallerIndex = allNodes.get(left).weight <= allNodes.get(right).weight ? left : right;
                if(allNodes.get(currentIndex).weight > allNodes.get(smallerIndex).weight){
                    swap(allNodes.get(currentIndex), allNodes.get(smallerIndex));

updatePositionMap(allNodes.get(currentIndex).key,allNodes.get(smallerIndex).key,currentIndex,smallerIndex);
                    currentIndex = smallerIndex;
                }else{
                    break;
                }
            }
            return minNode;
        }


        public int extractMin(){
            Node node = extractMinNode();
            return node.key;
        }


        private void swap(Node node1,Node node2){

            int weight = node1.weight;
            int data = node1.key;

            node1.key = node2.key;
            node1.weight = node2.weight;

            node2.key = data;
            node2.weight = weight;
    }


        private void updatePositionMap(int data1, int data2, int pos1, int pos2){
            nodePosition.remove(data1);
            nodePosition.remove(data2);
            nodePosition.put(data1, pos1);
            nodePosition.put(data2, pos2);
        }
    }


    class Node{

        int weight;
        int key;

        public Node(){

        }

    }


    class Graph{

        private int V;
        private LinkedList<Map<Integer,Integer>> adj[];

        public Graph(int V){
            this.V = V;
            adj = new LinkedList[V];

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

```
                adj[i] = new LinkedList<Map<Integer,Integer>>();
            }

        }

        public void addEdge(int srcVertex, int destVertex, int weight){

            Map<Integer,Integer> srcTodest = new HashMap<Integer,Integer>();
            Map<Integer,Integer> destTosrc = new HashMap<Integer,Integer>();

            srcTodest.put(destVertex,weight);
            destTosrc.put(srcVertex,weight);

            adj[srcVertex].add(srcTodest);
            adj[destVertex].add(destTosrc);

        }

        public Map<Integer,Integer> doBFS(int source){

            Map<Integer,Integer> output = new HashMap<Integer,Integer>();

            BinaryMinHeap minHeap = new BinaryMinHeap();

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

                if(i != source){
                    minHeap.add(20000000,i);
                    output.put(i,20000000);
                }
                else{
                    minHeap.add(0,i);
                    output.put(i,0);
                }
            }



            while(!minHeap.empty()){

                Node node = minHeap.extractMinNode();

                Iterator<Map<Integer,Integer>> i = adj[node.key].listIterator();

                while(i.hasNext()){

                    Map<Integer,Integer> val = i.next();

                    List<Integer> l = new ArrayList<Integer>(val.keySet());

                    if(minHeap.containsData(l.get(0))){

                        if(minHeap.getWeight(l.get(0)) > node.weight + val.get(l.get(0))){
                            minHeap.decrease(l.get(0),node.weight + val.get(l.get(0)));
                            output.put(l.get(0),node.weight + val.get(l.get(0)));
                        }

                    }

                }

            }

            return output;

        }


    }


public class Solution {

```

```
    public static void main(String[] args) throws IOException{
        //1
        //4 4
        //1 2 24
        //1 4 20
        //3 1 3
        //4 3 12
        //1

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int t = Integer.parseInt(br.readLine());

        for(int a0 = 0; a0 < t; a0++){

            String line = br.readLine();
            String[] numbers = line.split("\\s");

            int n = Integer.parseInt(numbers[0]);
            int m = Integer.parseInt(numbers[1]);

            Graph graph = new Graph(n);

            HashMap<String,Integer> myMap = new HashMap<String,Integer>();

            for(int a1 = 0; a1 < m; a1++){

                line = br.readLine();
                numbers = line.split("\\s");

                int x = Integer.parseInt(numbers[0]);
                int y = Integer.parseInt(numbers[1]);
                int r = Integer.parseInt(numbers[2]);

                if(myMap.containsKey(x + "-" + y)){
                    int mapVal = myMap.get(x + "-" + y);
                    if(r < mapVal){
                        graph.addEdge((x - 1),(y - 1),r);
                    }
                }
                else{
                    myMap.put(x + "-" + y,r);
                    myMap.put(y + "-" + x,r);
                        graph.addEdge((x - 1),(y - 1),r);
            }



            }
            int s = Integer.parseInt(br.readLine());

           Map<Integer,Integer> map = graph.doBFS(s - 1);

            for(int o : map.values()){



                if(o == 20000000){
                   System.out.print("-1 ");
                }
                else if(o == 0){
                    continue;
                }
                else{
                    System.out.print(o + " ");
                }

            }

            System.out.println("");

        }
    }
}
```

```
321
```

Line: 1 Col: 1

Upload Code as File | Test against custom input | Run Code | Submit Code