Dear reader, welcome to the article on the problem named **‘**[**Shortest Path in Weights**](https://www.pepcoding.com/resources/online-java-foundation/graphs/shortest-path-in-weights-official/ojquestion)**’.** The algorithm, which we are going to learn today, is known as **Dijkstra’s Algorithm**.

Trust me, the algorithm itself is easier than the pronunciation. The algorithm is very intuitive, as it follows a ***Greedy Approach***. The algorithm is also known as ‘***Single Source Shortest Paths Algorithm***’.

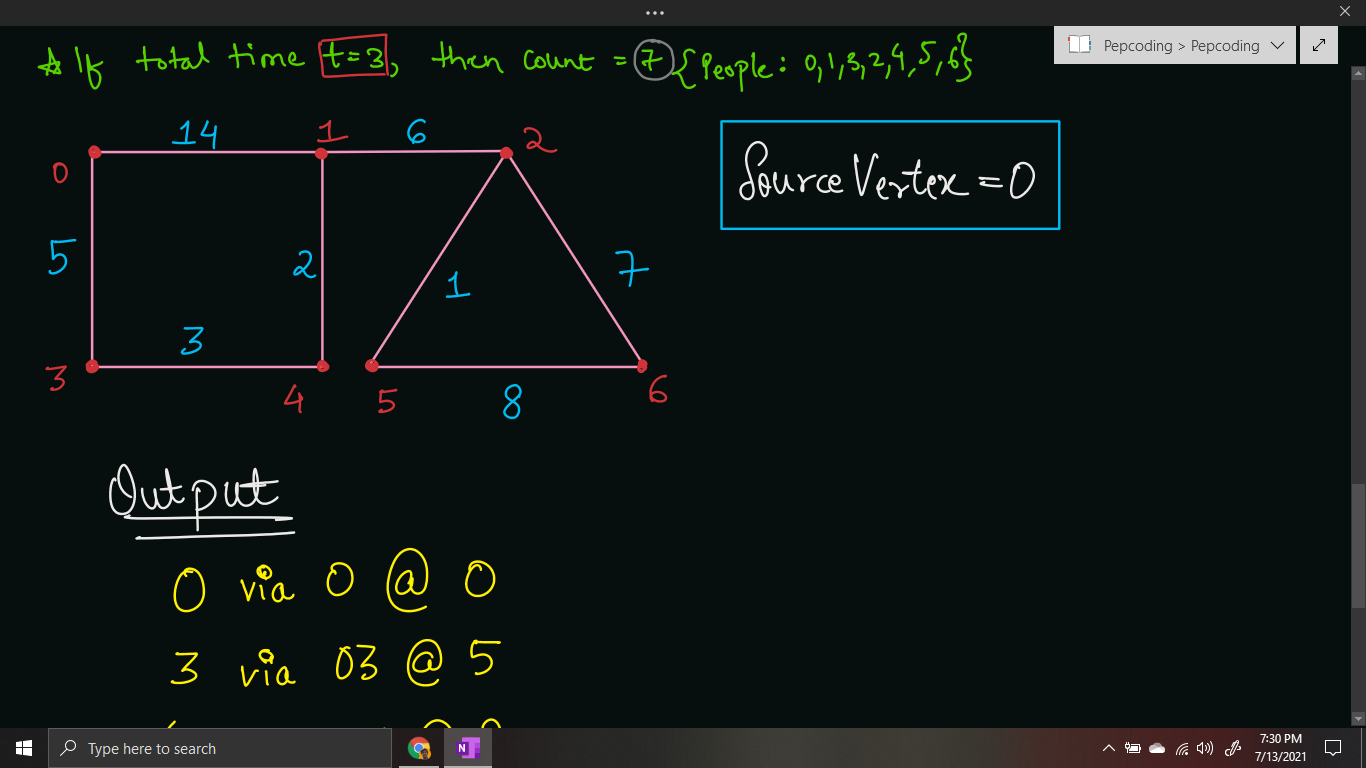
So, enough talk, let’s jump into the problem!

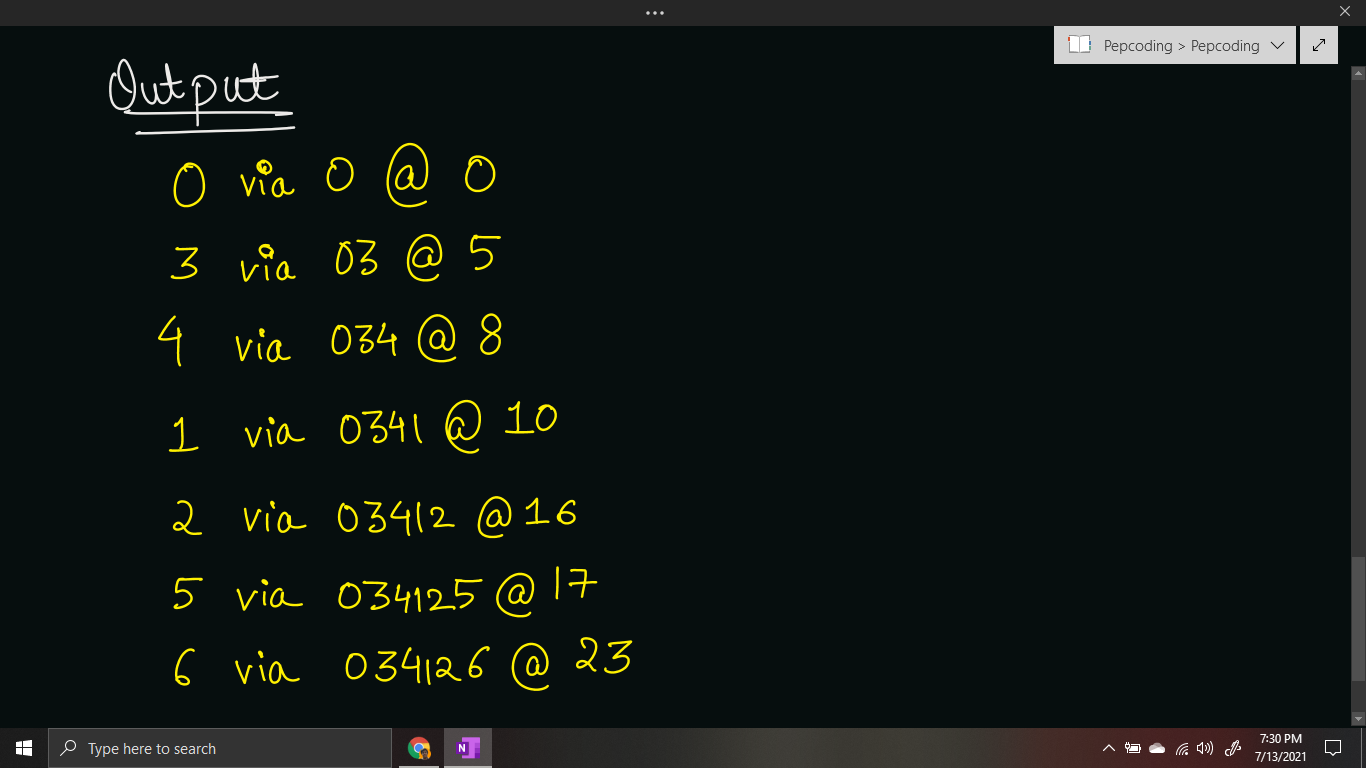
***Problem Statement:***

* You are given an undirected graph and a source vertex. The vertices represent cities and the edges represent distance in kms.
* The graph is weighted in nature. The weights of all edges are ***non-negative***.
* You are required to find the shortest path to each city (in terms of kms) from the source city along with the total distance on the path from source to destinations.

*Note*: Input is given in the form of adjacency list. Please go through the output format once, before learning the algorithm.

***Example:***

****

****

***Solution:***

Please note that this problem is different from Single Source Source Shortest Path in terms of count of edges (in unweighted graph).

For the shortest path in terms of edge count, we had used BFS traversal in the previous problem ‘[***Spread of Infection’***](https://www.pepcoding.com/resources/online-java-foundation/graphs/infection-spread-official/ojquestion).

Here, we need to find the shortest path in terms of ***weight***. The shortest path may (or may not) be longer in terms of *edge count*.

In the above example, the shortest weighted path from 0 to 1 is equal to 10 via “02341”, which is longer from the path “01”.

You will be amazed to read that, we just need a slight modification to our BFS algorithm, so that it can find the shortest path in terms of weight.

Can you guess where we need to tweak?

In BFS traversal, we used a ***queue*** data structure. But since we can only maintain FIFO order in the queue, we cannot get the shortest path in terms of weight.

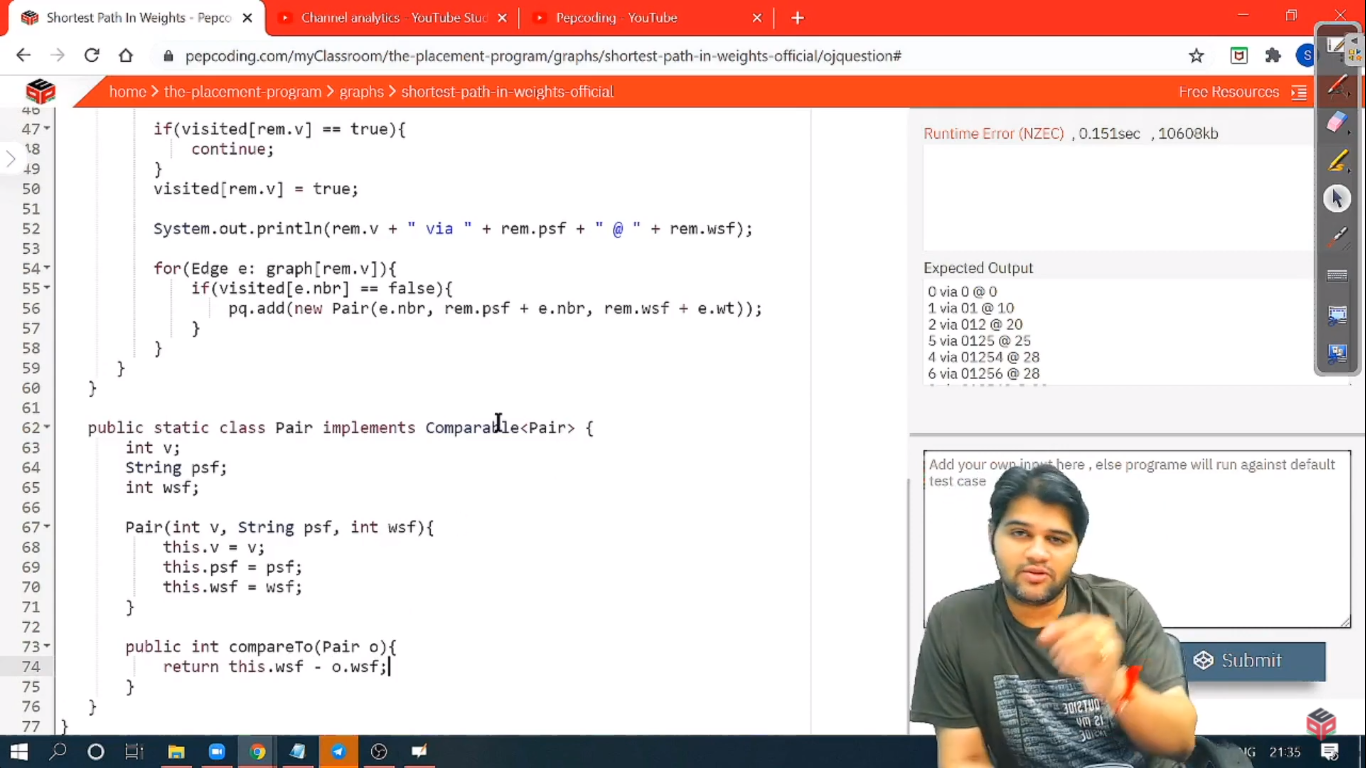
For Dijkstra’s algorithm, we just need to replace the queue with a ***priority queue***, and that’s it.

You should be able to analyze that the Priority Queue pops out the element with higher priority among all the candidates.

We should have a path with the shortest distance as the highest priority. Hence, we will declare a Pair class, which will have a weight so far, storing the maximum distance of that node from source vertex.

But, how will our priority queue be able to decide which Pair object has higher priority?

So, our Pair class must implement a comparable interface. Please refer to the '[***Comparable vs Comparator***](https://www.pepcoding.com/resources/online-java-foundation/hashmap-and-heap/comparator-comparable/video)’ video of ‘Priority Queue’ if you do not know how to define a custom comparator for user defined classes. We will define the compareTo function which compares the values of weight so far (wsf).



***Algorithm***

* Make a boolean array ***visited*** of size n (number of vertices), which will store whether the node is visited or not. Initially all values will be false (no node is visited).
* Initialize an empty priority queue ***pq*** which stores objects of type ***Pair***, which have three data members: ***v*** (node’s value), ***psf*** (path so far), and ***wsf*** (weight so far).
* Insert a Pair(src, src + “”, 0) into the priority queue. It means that we are starting the algorithm with source vertex src, and weight so far as 0.
* Run the algorithm until there are some nodes present in the priority queue, i.e. until ***pq.size() > 0***.
  + Pop the top element of the priority queue and store it in a Pair object *rem*.
  + If the top element is already visited (we reached the current node from a shorter path before), then simply continue.
  + Else, mark the top element as visited, i.e. *visited[rem.v] = true*.
  + Print the node’s value along with the path so far and the shortest distance using the statement: *System.out.println(rem.v + " via " + rem.psf + " @ " + rem.wsf);*
  + Traverse through all the neighbours of the current vertex.
    - If the neighbour vertex is not visited (*visited[e.nbr] == false*), then add the node in the priority queue by adding the node’s value in path so far and distance in weight so far.

***Implementation***

*Note*: Before reading the Code, we recommend that you must try to come up with the solution on your own. Now, hoping that you have tried by yourself, here is the Java code.

import java.io.\*;

import java.util.\*;

public class Main {

static class Edge {

int src;

int nbr;

int wt;

Edge(int src, int nbr, int wt) {

this.src = src;

this.nbr = nbr;

this.wt = wt;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int vtces = Integer.parseInt(br.readLine());

ArrayList<Edge>[] graph = new ArrayList[vtces];

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

graph[i] = new ArrayList<>();

}

int edges = Integer.parseInt(br.readLine());

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

String[] parts = br.readLine().split(" ");

int v1 = Integer.parseInt(parts[0]);

int v2 = Integer.parseInt(parts[1]);

int wt = Integer.parseInt(parts[2]);

graph[v1].add(new Edge(v1, v2, wt));

graph[v2].add(new Edge(v2, v1, wt));

}

int src = Integer.parseInt(br.readLine());

boolean[] visited = new boolean[vtces];

PriorityQueue<Pair> queue = new PriorityQueue<>();

queue.add(new Pair(src, src + "", 0));

while(queue.size() > 0){

Pair rem = queue.remove();

if(visited[rem.v] == true){

continue;

}

visited[rem.v] = true;

System.out.println(rem.v + " via " + rem.psf

+ " @ " + rem.wsf);

for (Edge e : graph[rem.v]) {

if (visited[e.nbr] == false) {

queue.add(new Pair(e.nbr,

rem.psf + e.nbr, rem.wsf + e.wt));

}

}

}

}

static class Pair implements Comparable<Pair> {

int v;

String psf;

int wsf;

Pair(int v, String psf, int wsf){

this.v = v;

this.psf = psf;

this.wsf = wsf;

}

public int compareTo(Pair o){

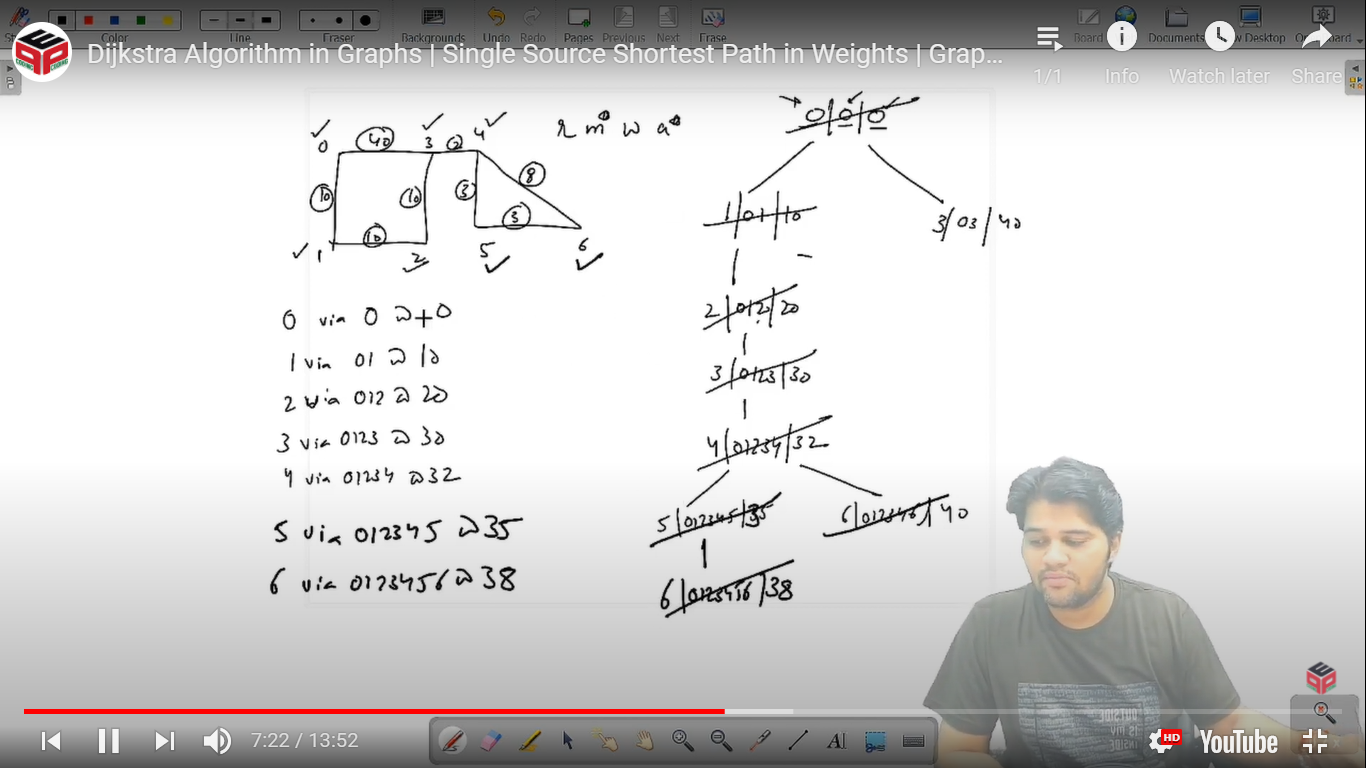
return this.wsf - o.wsf;

}

}

}

This code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=sD0lLYlGCJE&list=TLGG7ri8hA2DOAcxMzA3MjAyMQ). Do check it out to understand the concept completely.



* What is the ***time complexity*** of the above code?

This is an interesting analysis. The time complexity of Dijkstra’s algorithm will be O(E + V logV) where V = number of vertices and E = number of edges.

This is because, we are iterating over all the edges once during the entire run of the algorithm

In each iteration, we are popping one node and pushing the unvisited neighbour nodes. Since the priority queue can contain all the vertices, the push or pop operation will be O(log V).

Hence the total time complexity will be O(E) + O(V) \* O(log V) = O(E + V log V).

Note: You can argue that we might be having multiple Pairs having the same node’s value. So, the maximum size of the priority queue will be not O(N) but O(E).

But, even if you replace log V with Log E (cost of one push/pop operation), then there will be no difference in the time complexity as:

O(E + VlogE) = O(E + Vlog(V^2)) = O(E + 2V logV) = O(E + V logV) only.

* What is the ***space complexity*** of the above code ?

We are taking a priority queue of Pair nodes. Hence, the space complexity will be O(N) where N = maximum Pair nodes in the queue, which is equivalent to O(V).

**Follow Up:**

* We applied the dijkstra’s algorithm on an undirected weighted graph. Can you apply it on a directed weighted graph?
* Dijkstra’s algorithm gives the shortest path of all destination nodes from a ***single*** source node. To find the shortest path between ***all the pairs of vertices***, we will have to run Dijkstra's algorithm with source nodes as {0, 1, 2, … n-1} individually.
* Please remember that the dijkstra’s algorithm will not give correct results if there exists a ***negative weight cycle*** in the graph. Hence, either the graph should contain only positive edges or there must be no cycles with negative weight edges present for the dijkstra’s algorithm to work.

Hope that you liked the article on ***’Shortest Path in Weights’***.

Subscribe to Pepcoding’s youtube channel for more such amazing video content on Data Structures & Algorithms.

You can suggest any improvements to the article on our telegram channel, or on the youtube channel’s comment section.

Article Contributed by: [Archit Aggarwal](https://www.linkedin.com/in/archit-aggarwal-6a7716189/)