Dear reader, welcome to the article on the problem named **‘**[**Minimum Wire Required To Connect All Pcs**](https://www.pepcoding.com/resources/online-java-foundation/graphs/minimum-wire-to-connect-all-pcs-official/ojquestion)**’.**

The problem is popularly known as ‘***Minimum Spanning Tree***’. The algorithm, which we are going to learn today, is known as **Prim’s Algorithm**.

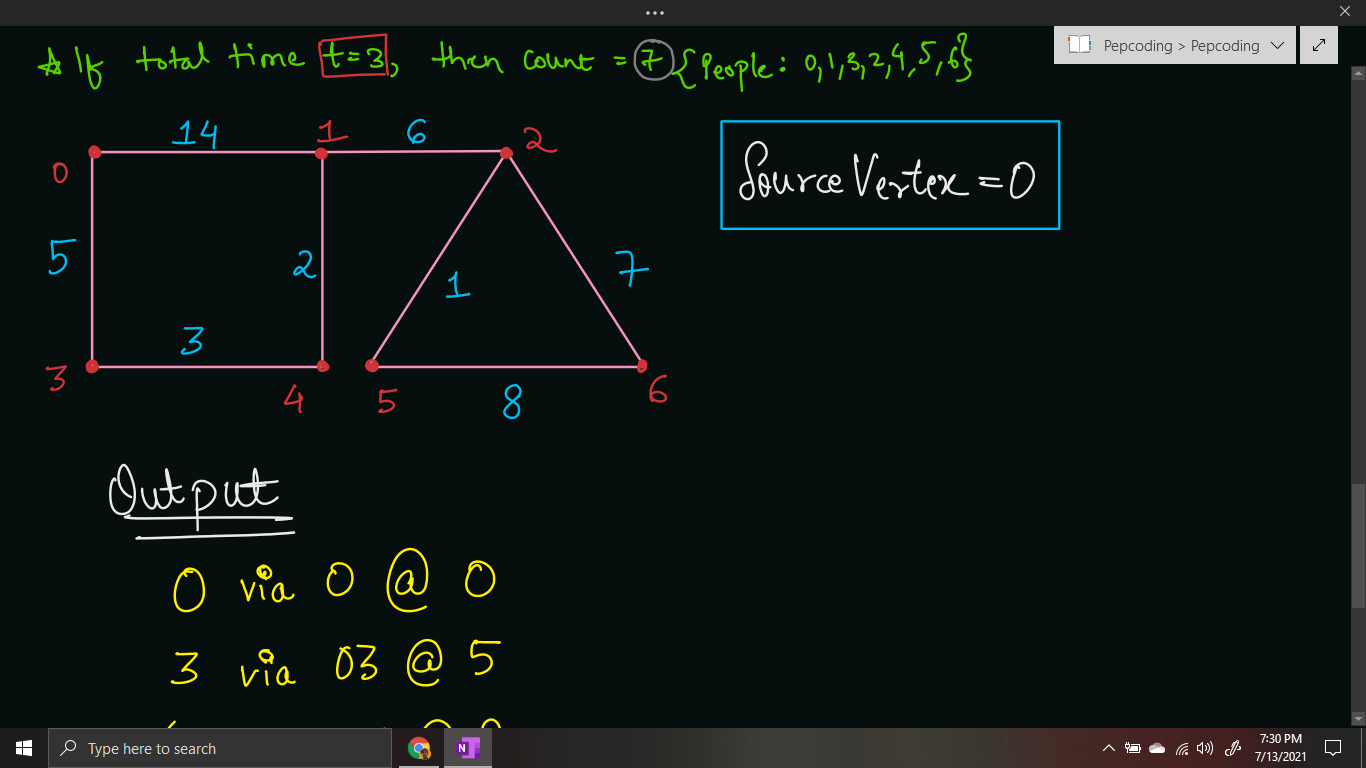
***Problem Statement:***

* You are given a graph and a source vertex. The vertices represent computers and the edges represent the length of LAN wire required to connect them.
* You are required to find the minimum length of wire required to connect all PCs over a network.
* Print the output in terms of which all PCs need to be connected, and the length of wire between them.

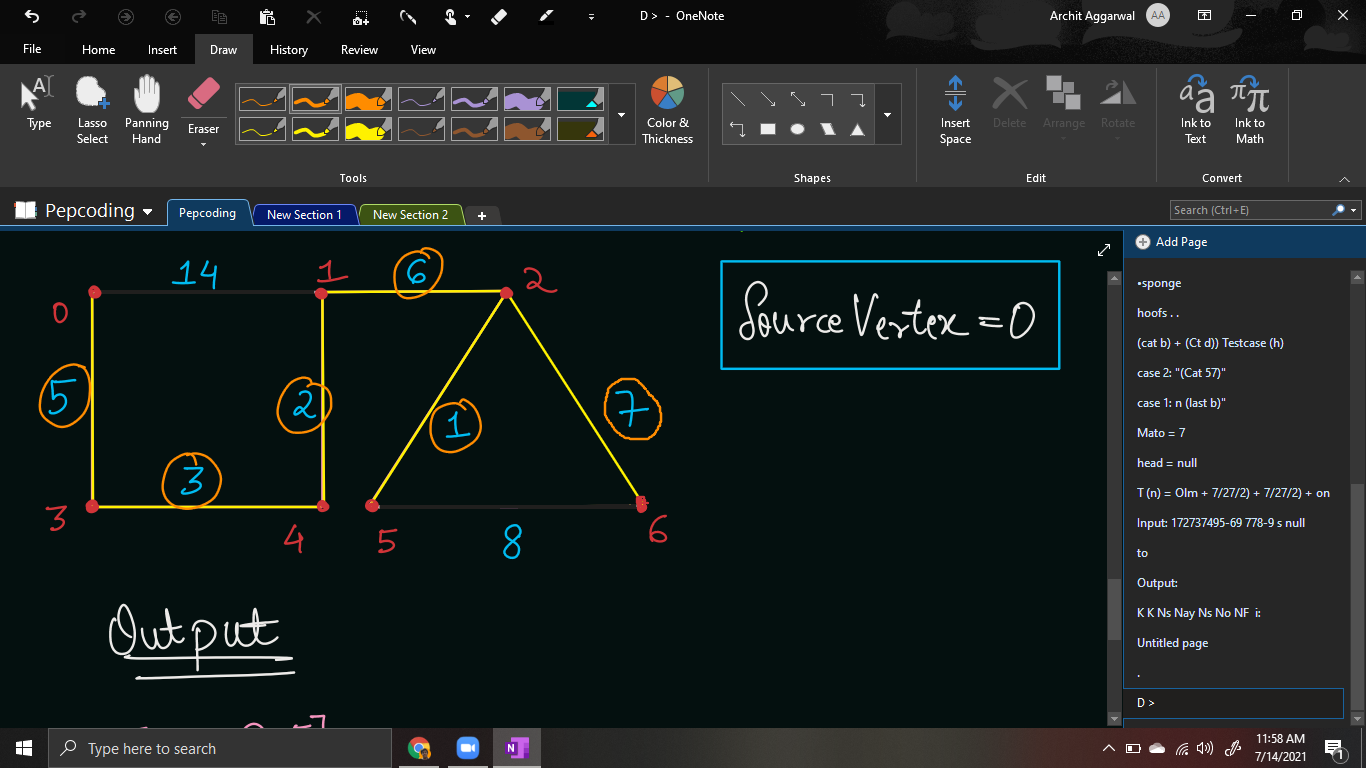
*Note*: Input is given in the form of adjacency list. Please go through the output format once, before moving ahead in the article.

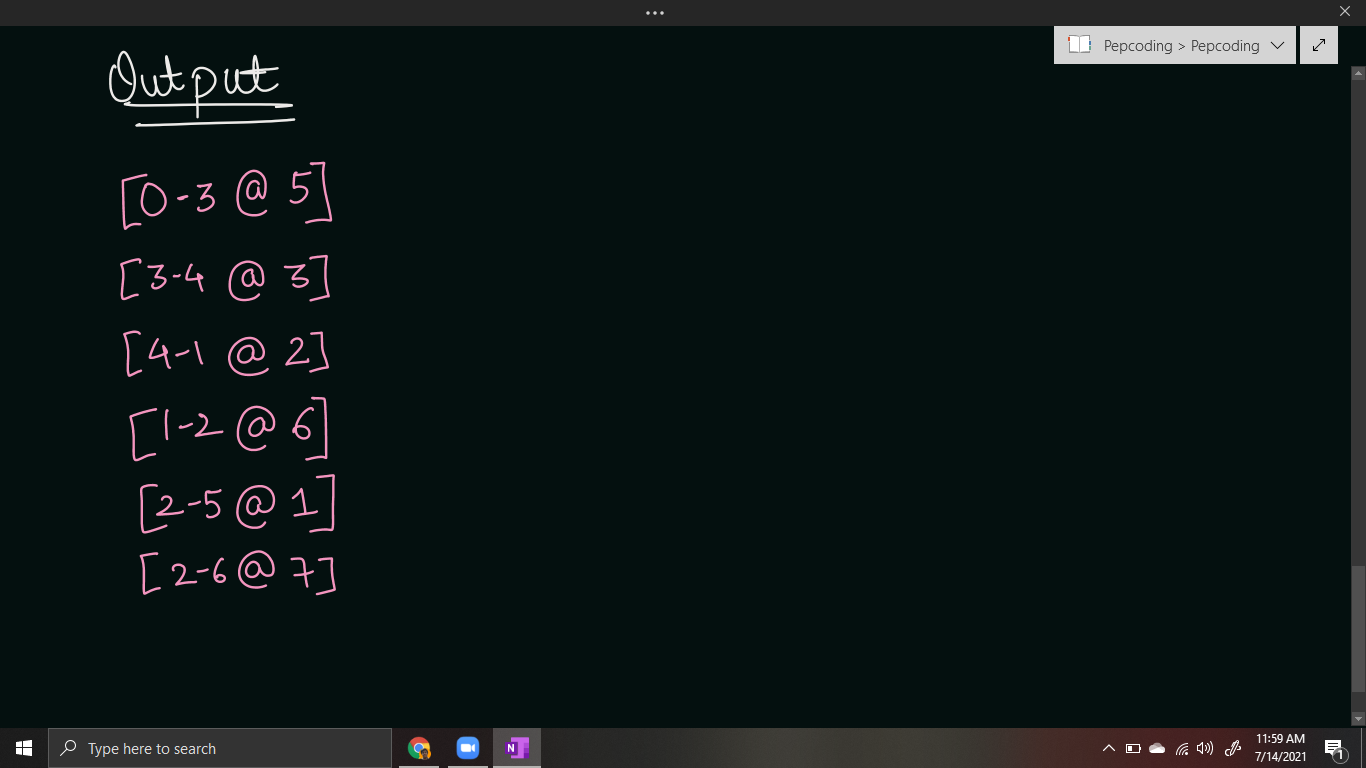
***Example:***

Input Graph:

****

Minimum Spanning Tree:

****

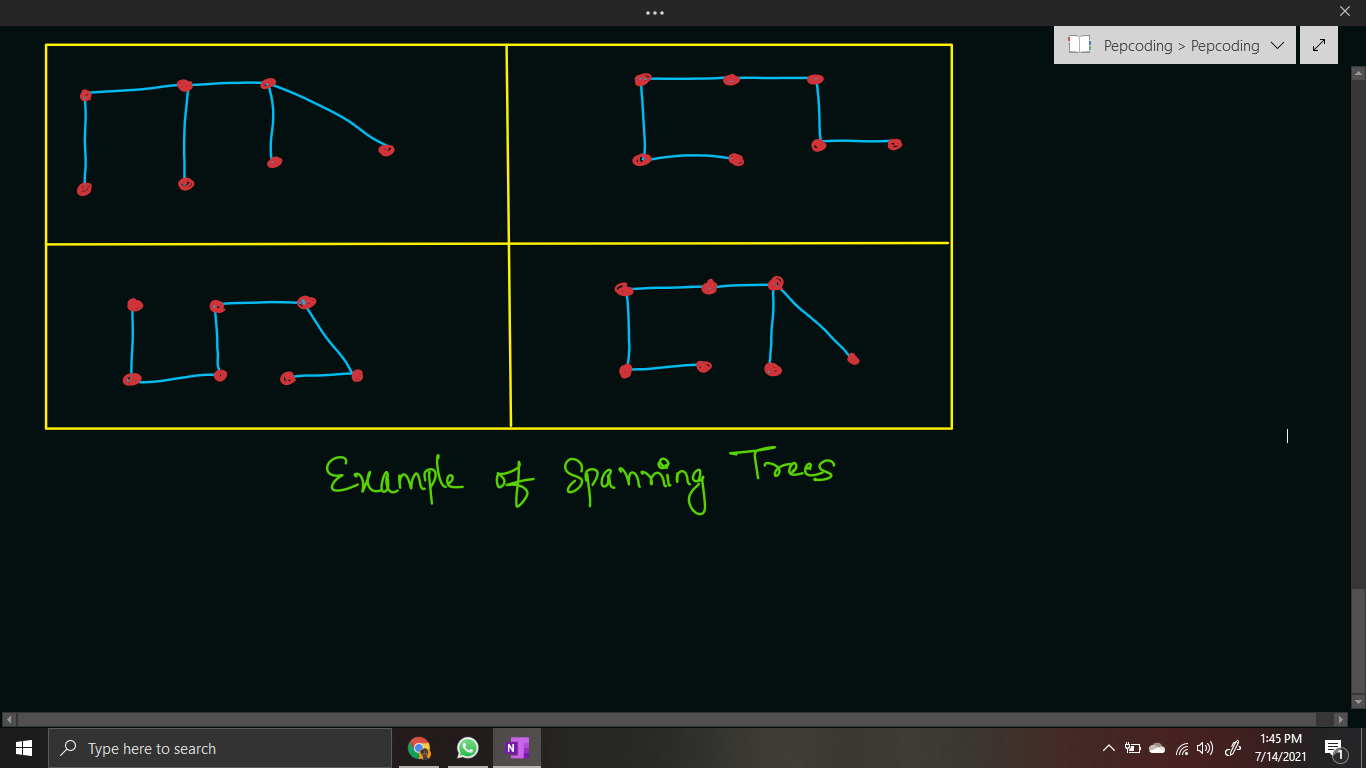


***Solution:***

First of all, let us see, what is a MINIMUM SPANNING TREE?

* **Tree**: A tree is an undirected ***acyclic*** and ***connected*** graph. If a tree has n nodes, then there will be n-1 edges present. Since the tree is connected, there cannot be more than 1 component.
* **Spanning Tree**: A spanning tree is a ***subgraph*** which ***covers all the vertices*** of the graph. Hence, if the graph has n vertices, then the spanning tree must contain all the n vertices in one connected component.
* **Minimum Spanning Tree**: There can be many spanning trees possible. The one whose ***sum of weights of edges is minimum***, is known as minimum spanning tree.

So, for the graph drawn above, there can be many spanning trees possible:



Prim’s algorithm is based on a ***greedy approach*** and is very similar to Dijkstra’s algorithm.

In Dijkstra’s algorithm, we were required to find the shortest path from source, hence we required the total sum of weights from source to destination to be minimum.

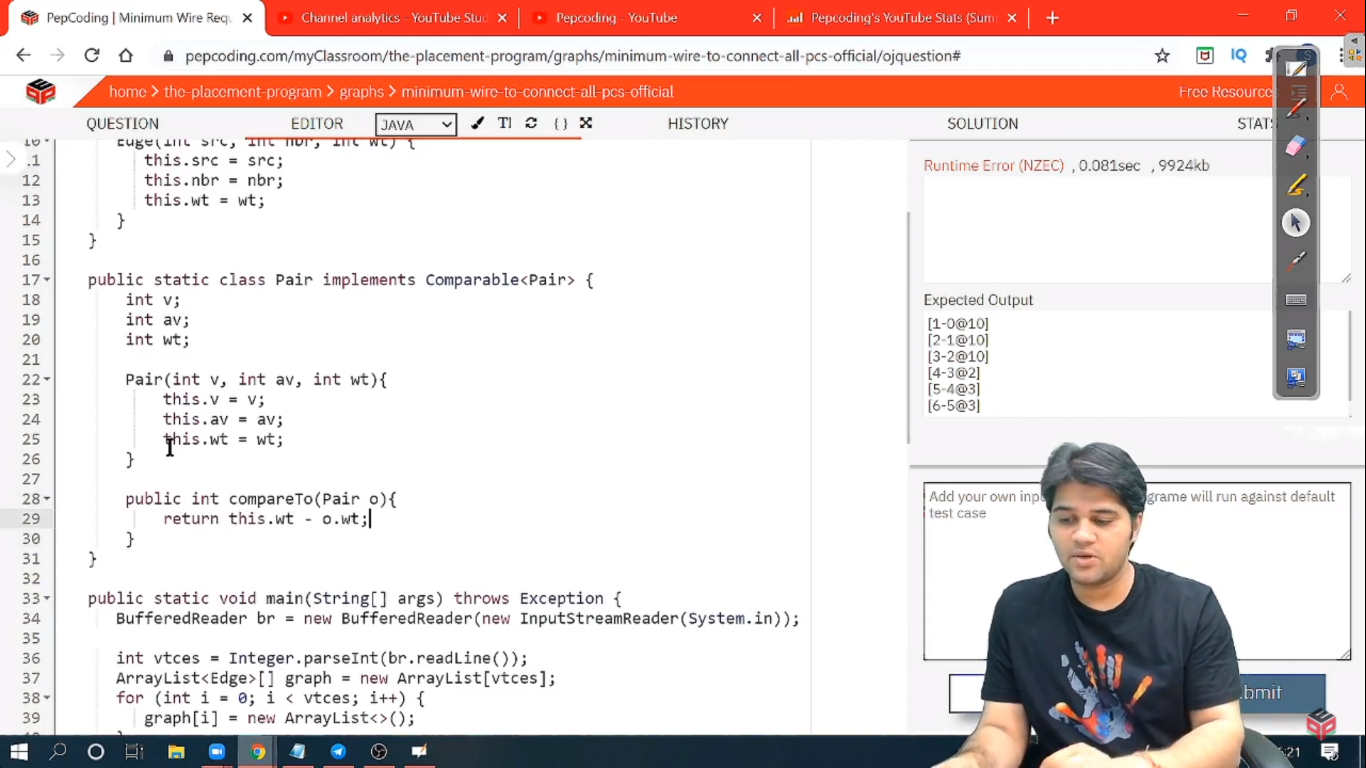
But in prim’s algorithm, we need the sum of all edges taken in spanning tree to be minimum. So, instead of adding the summation of paths, we will require to add the ***edge weights*** directly into the ***priority queue***.

Intuition behind the prim’s algorithm is simple, i.e. pick the edges with the minimum weights possible, such that all vertices are connected in a single component.

Hence, we can use priority queue to find the minimum edge weight among all the edges of a source node.

Hence, the Pair objects that will be pushed into the priority queue will not have weight so far, but instead have simply the edge weight.

Also, instead of having the path so far, they will store the information of the parent vertex, through which they are acquired (pushed into the priority queue).



***Pseudo Code/Algorithm***

* Create a priority queue of type Pair.
* Push a pair node having vertex 0, which is acquired via node -1 (imaginary or dummy node), and the weight is 0.
* Create a visited array of size n (n = number of vertices), initially containing all nulls (false).
* Run a while loop until the queue becomes empty.
  + **Remove** the top element of the priority queue and store it in a Pair object rem.
  + If the node is already visited (visited[rem.v] != null), then simply continue.
  + Else, visit the node, i.e. mark visited[rem.p] = rem.p
  + If the parent node is not -1 (dummy node), then print the edge weight by using the following statement:  ***System.out.println("[" + rem.v + "-" + rem.p + "@" + rem.wt + "]");***
  + Traverse through all the neighbours of the current node, and push the unvisited ones into the priority queue, with the parent node as the current node (rem.v) and the weight as the edge weight between rem.v and the neighbour’s node.

***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 = 0;

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

queue.add(new Pair(src, -1, 0));

Integer[] visited = new Integer[vtces];

while(queue.size() > 0){

Pair rem = queue.remove();

if(visited[rem.v] != null){

continue;

}

visited[rem.v] = rem.p;

if(rem.p != -1){

System.out.println("[" + rem.v + "-" +

rem.p + "@" + rem.wt + "]");

}

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

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

queue.add(new Pair(e.nbr, rem.v, e.wt));

}

}

}

}

static class Pair implements Comparable<Pair> {

Integer v;

Integer p;

int wt;

Pair(Integer v, Integer p, int wt){

this.v = v;

this.p = p;

this.wt = wt;

}

public int compareTo(Pair o){

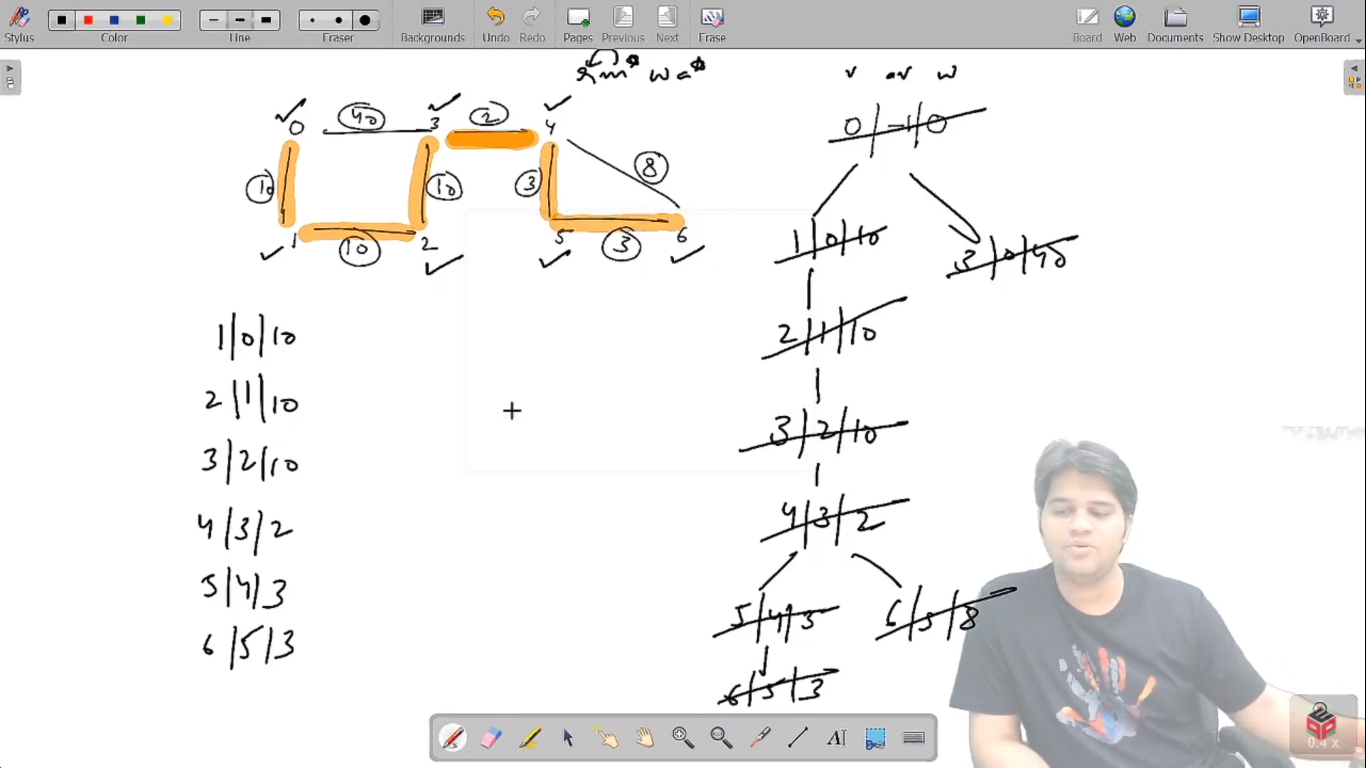
return this.wt - o.wt;

}

}

}

This code is written and explained by our team in the [solution video](https://www.youtube.com/watch?v=Vw-sktU1zmc&list=TLGGnnxbek9XREoxNDA3MjAyMQ). Do check it out to understand the concept completely.



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

Time Complexity is similar to Dijkstra’s algorithm, since we have just replaced the weight so far with the current edge weight. Hence, total time complexity will be ***O(E + VlogV)*** where E = number of edges and V = number of vertices.

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

We are using a priority queue, which will take ***O(N)*** space, where N is equal to the maximum pair objects pushed, and is equivalent to the number of vertices. Also, we have declared a visited array of size N.

**Follow Up:**

* Please remember that for a graph consisting of ***n nodes***, any spanning tree will always have ***n - 1 edges***. This is because spanning tree is a ‘tree’ and there can be no cycles present in a tree. Adding the nth edge in a tree will always result in a cycle formation.
* There is another algorithm to find the Minimum Spanning Tree of a Graph, which is known as ‘***Kruskal’s Algorithm***’ which uses Disjoint Set Union (DSU) data structure. (We will learn about this algorithm in the level 2 section of our course.)

Hope that you liked the article on ***’Minimum Wire To Connect All PCs’***.

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/)