**Minimum Spanning Trees**

Minimum Meaning: The least

Spanning Meaning: extend from side to side

"the stream was spanned by a narrow bridge"

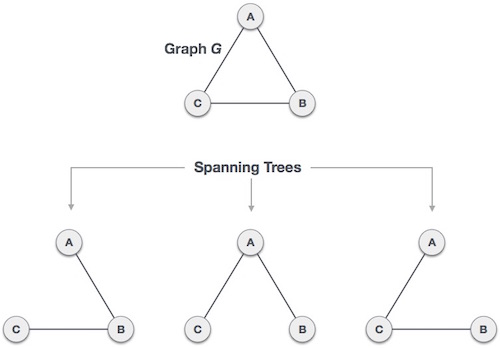
Trees Meaning (In programming) : Connected Acyclic graph.

Spanning Tree of G: Subset of edges of G that for a tree and hit all of vertices of G.

Minimum Spanning Tree: Given a Graph G(V,E) & edge weights W: E-> Real numbers, Find a spanning tree of minimum total length. (undirected graphs)

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected...

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum **nn-2** number of spanning trees, where **n** is the number of nodes. In the above addressed example, **33−2 = 3** spanning trees are possible.

**General Properties of Spanning Tree**

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G −

* A connected graph G can have more than one spanning tree.
* All possible spanning trees of graph G, have the same number of edges and vertices.
* The spanning tree does not have any cycle (loops).
* Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
* Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

## Mathematical Properties of Spanning Tree

* Spanning tree has **n-1** edges, where **n** is the number of nodes (vertices).
* From a complete graph, by removing maximum **e - n + 1** edges, we can construct a spanning tree.
* A complete graph can have maximum **nn-2** number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

## Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

* **Civil Network Planning**
* **Computer Network Routing Protocol**
* **Cluster Analysis**

## Minimum Spanning Tree (MST)

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a [connected](https://en.wikipedia.org/wiki/Connected_graph), edge-weighted (un)directed graph that connects all the [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) together, without any cycles and with the minimum possible total edge weight. That is, it is a [spanning tree](https://en.wikipedia.org/wiki/Spanning_tree) whose sum of edge weights is as small as possible. More generally, any edge-weighted undirected graph (not necessarily connected) has a **minimum spanning forest**, which is a union of the minimum spanning trees for its [connected components](https://en.wikipedia.org/wiki/Connected_component_(graph_theory)).

**Assumptions.**

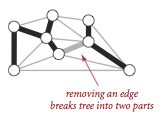
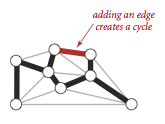
 We adopt the following conventions:

* *The graph is connected.* The spanning-tree condition in our definition implies that the graph must be connected for an MST to exist. If a graph is not connected, we can adapt our algorithms to compute the MSTs of each of its connected components, collectively known as a *minimum spanning forest*.
* *The edge weights are not necessarily distances.* Geometric intuition is sometimes beneficial, but the edge weights can be arbitrary.
* *The edge weights may be zero or negative.* If the edge weights are all positive, it suffices to define the MST as the subgraph with minimal total weight that connects all the vertices.
* *The edge weights are all different.* If edges can have equal weights, the minimum spanning tree may not be unique. Making this assumption simplifies some of our proofs, but all of our algorithms work properly even in the presence of equal weights.

**Underlying principles.**

 We recall two of the defining properties of a tree:

* Adding an edge that connects two vertices in a tree creates a unique cycle.
* Removing an edge from a tree breaks it into two separate subtrees.

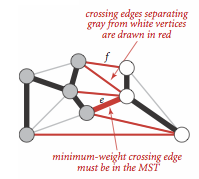
        

Note-> Minimum Spanning Forest: Minimum spanning forest is a generalization of minimum spanning tree for unconnected graphs. For every component of the graph, take its MST and the resulting collection is a minimum spanning forest.

### It follows the Greedy Approach which states making the locally best choice or decision ignoring effect on future.

### Proposition. (Cut property)

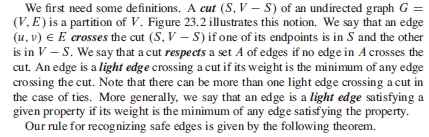
 Given any cut in an edge-weighted graph (with all edge weights distinct), the crossing edge of minimum weight is in the MST of the graph.

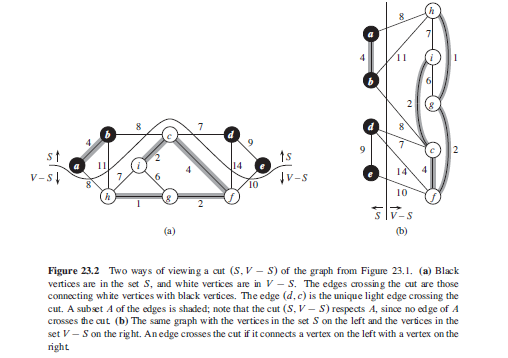


### Proposition. (Greedy MST algorithm)

 The following method colors black all edges in the the MST of any connected edge-weighted graph with V vertices: Starting with all edges colored gray, find a cut with no black edges, color its minimum-weight edge black, and continue until V-1 edges have been colored black.

Greedy algorithm for the
minimum spanning tree problem

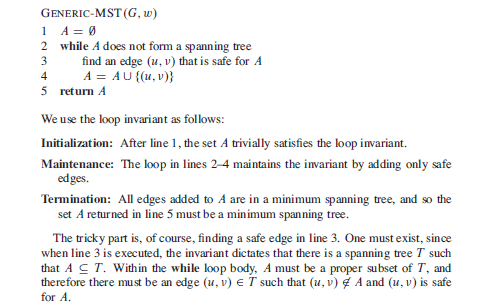


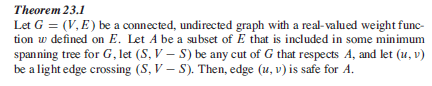


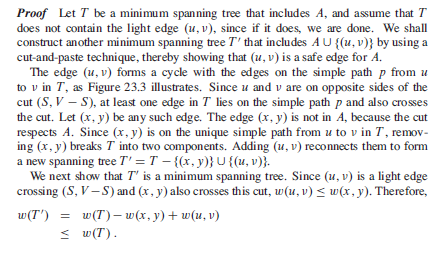
**Greedy Properties:**

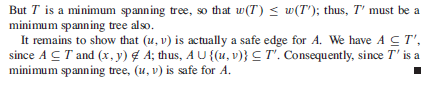
**Optimal Substructure**: Optimal Solutions to problem incorporate optimal solutions to subproblems S

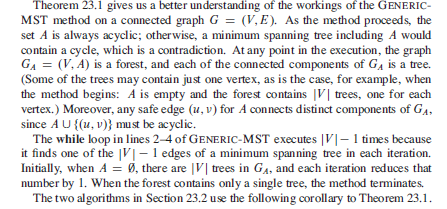
**Greedy-Choice Property**: locally optimal choices lead to globally optimal solution











**How to solve Kruskal’s Problem:**

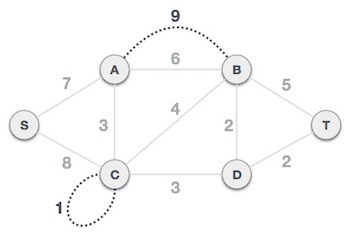
Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example −



## Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.

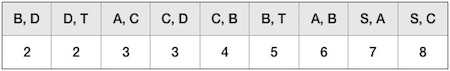


In case of parallel edges, keep the one which has the least cost associated and remove all others.



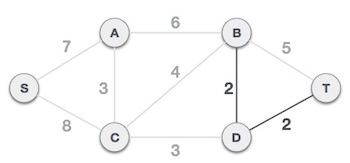
## Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).



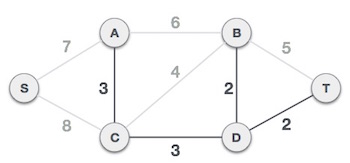
## Step 3 - Add the edge which has the least weightage

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

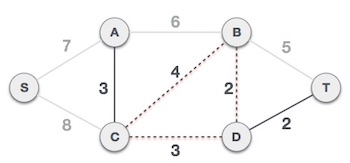


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

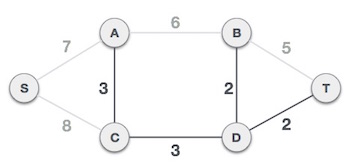
Next cost is 3, and associated edges are A,C and C,D. We add them again −



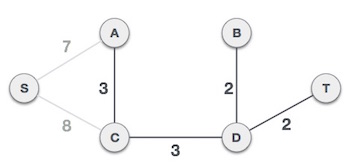
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. −



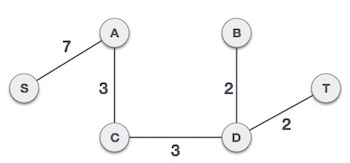
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



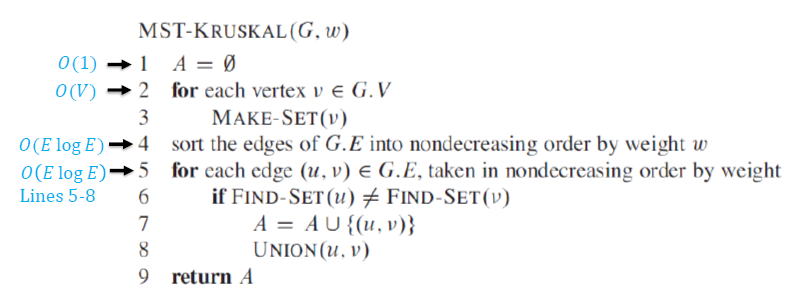
We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



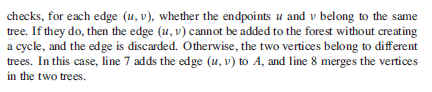
Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



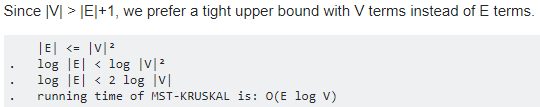
By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.







**Complexity: 𝑂(𝐸log𝐸)=𝑂(𝐸log𝑉)**



To implement Kruskal's algorithm, we use a priority queue to consider the edges in order by weight or by sorting the edges in an increasing order, a union-find data structure to identify those that cause cycles, and a queue to collect the MST edges.

Maintain connected components in MST so far in **Union-Find Structure.** T could be a Forest.

**How to solve Prim’s algorithm:**

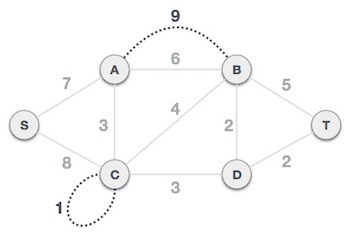
Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example −



## Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

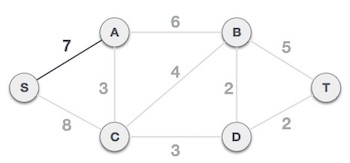


## Step 2 - Choose any arbitrary node as root node

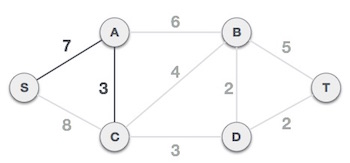
In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

## Step 3 - Check outgoing edges and select the one with less cost

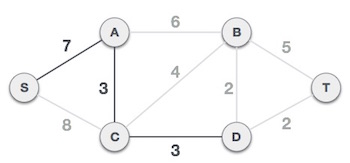
After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



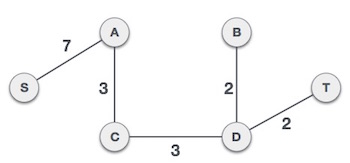
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



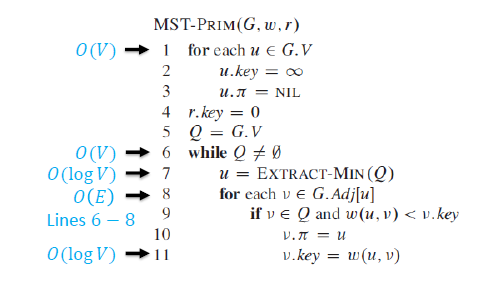
After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



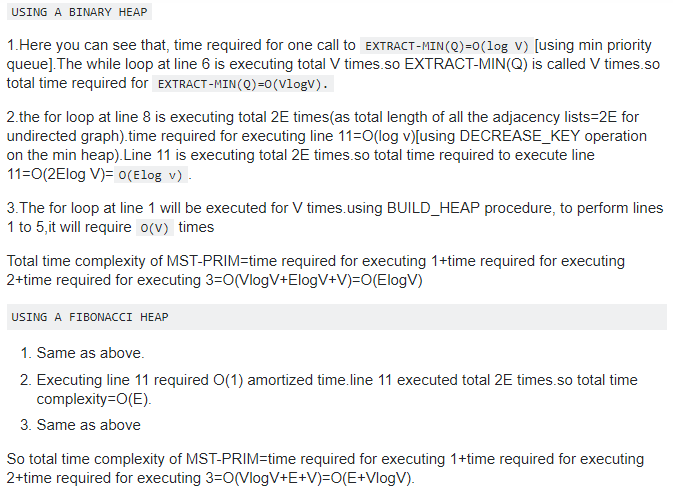
We may find that the output spanning tree of the same graph using two different algorithms is same.



Pi here in pseudocode is the Parent.

**Complexity: 𝑂(𝑉log𝑉+𝐸log𝑉) =𝑂(𝐸log𝑉)**

**–Using Fibonacci heaps: 𝑂(𝐸+𝑉log𝑉)**



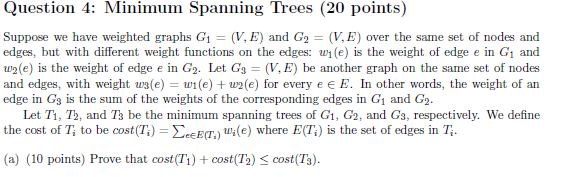
**Lazy implementation**. We use a priority queue to hold the crossing edges and find one of minimal weight. Each time that we add an edge to the tree, we also add a vertex to the tree. To maintain the set of crossing edges, we need to add to the priority queue all edges from that vertex to any non-tree vertex. But we must do more: any edge connecting the vertex just added to a tree vertex that is already on the priority queue now becomes ineligible (it is no longer a crossing edge because it connects two tree vertices). The lazy implementation leaves such edges on the priority queue, deferring the ineligibility test to when we remove them.

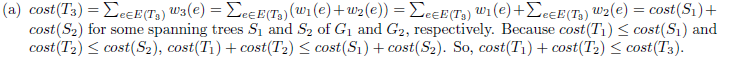
**Eager implementation**. To improve the lazy implementation of Prim's algorithm, we might try to delete ineligible edges from the priority queue, so that the priority queue contains only the crossing edges. But we can eliminate even more edges. The key is to note that our only interest is in the minimal edge from each non-tree vertex to a tree vertex. When we add a vertex v to the tree, the only possible change with respect to each non-tree vertex w is that adding v brings w closer than before to the tree. In short, we do not need to keep on the priority queue all of the edges from w to vertices tree—we just need to keep track of the minimum-weight edge and check whether the addition of v to the tree necessitates that we update that minimum (because of an edge v-w that has lower weight), which we can do as we process each edge in s adjacency list. In other words, we maintain on the priority queue just one edge for each non-tree vertex: the shortest edge that connects it to the tree.

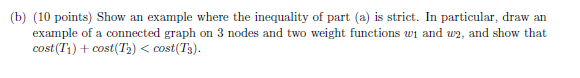
**Questions from MIT finals:**

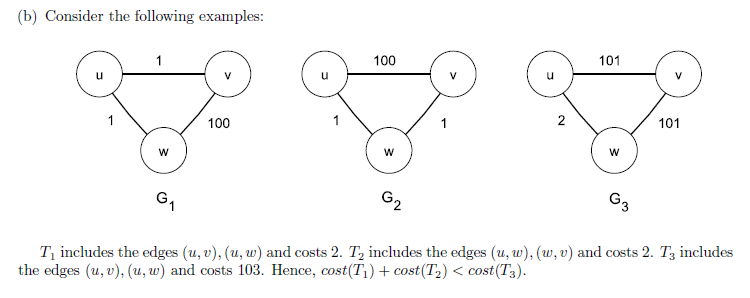
|  |  |  |  |
| --- | --- | --- | --- |
|  | T F | Negating all the edge weights in a weighted undirected graph G and | |
| then finding the minimum spanning tree gives us the *maximum*-weight spanning | | |
| tree of the original graph G. | | |
| Solution: True.  In a graph with unique edge weights, the spanning tree of second-lowest weight is unique.  Solution: False, can construct counter-example.  If we use a max-queue instead of a min-queue in Kruskal’s MST algorithm, it will return the spanning tree of maximum total cost (instead of returning the spanning tree of minimum total cost). (Assume the input is a weighted connected undirected graph.)  Solution: True. The proof is essentially the same as for the usual Kruskal’s algorithm. Alternatively, this is equivalent to negating all the edge weights and running Kruskal’s algorithm. | | |
|  | | |

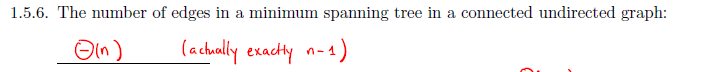
**Questions from Stanford finals:**

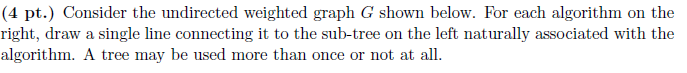


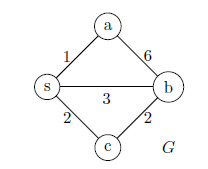


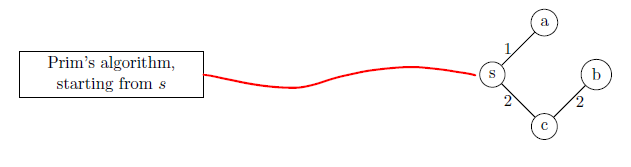


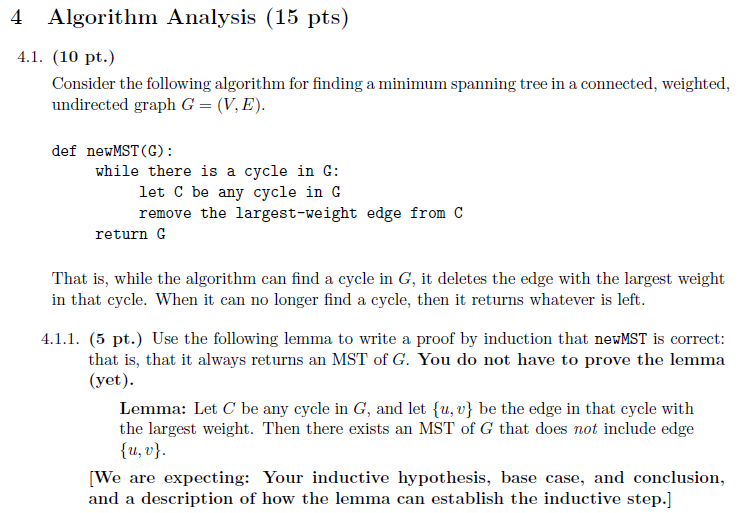


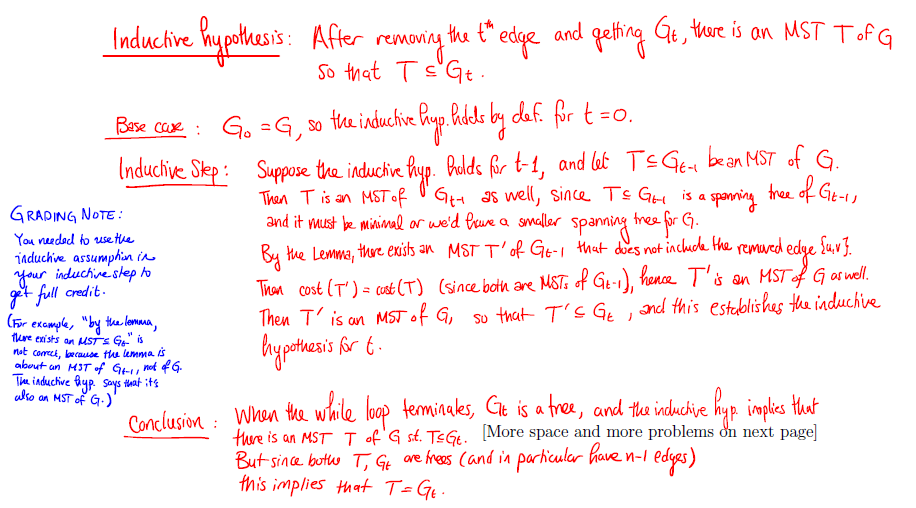


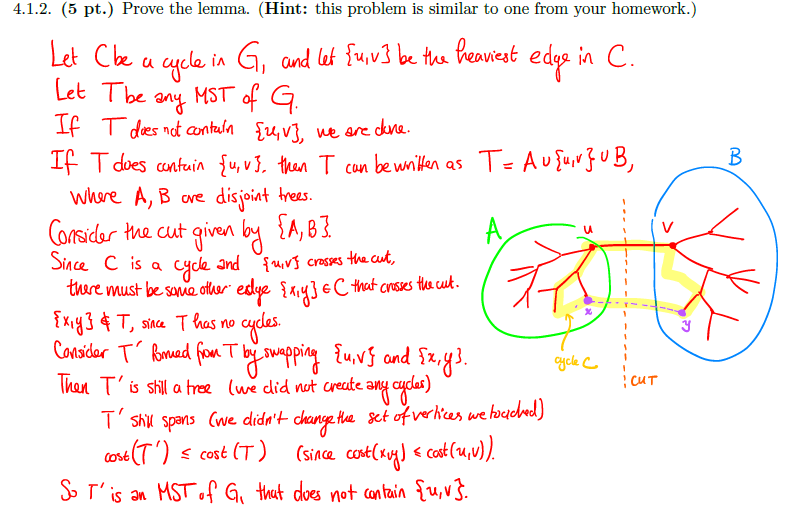


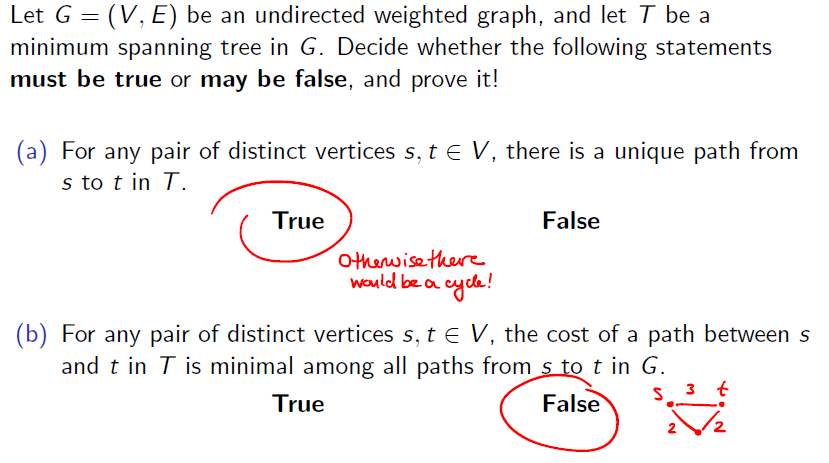










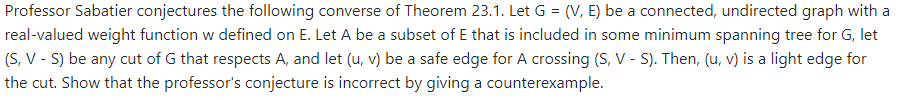


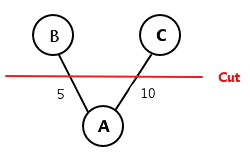
**Reference Questions:**



**Solution:** 

**-------------------------------------------------------------------**



**Solution:**





**Solution:** 

**---------------------------------------------------------------------------------**

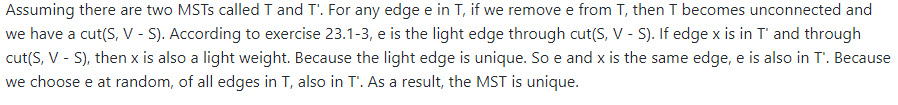


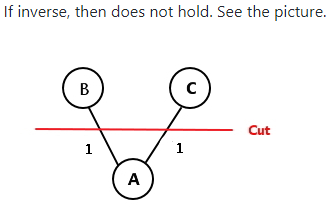
**Solution:** 

**-------------------------------------------------------------------------------------**

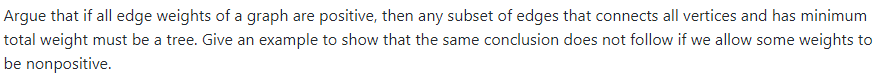
**Solution:** 

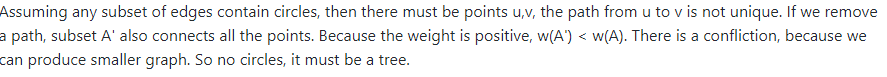


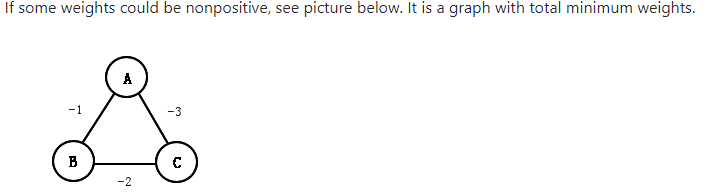
**Solution:** 



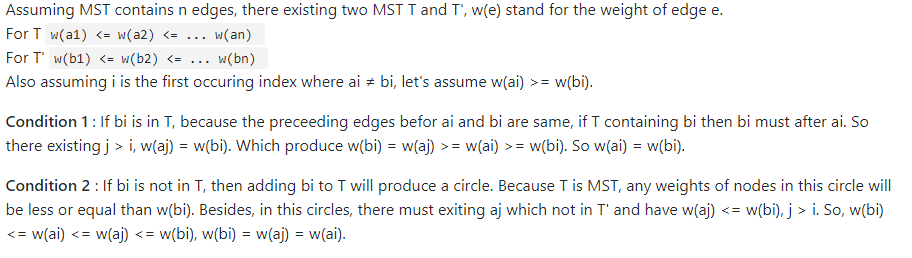
**-------------------------------------------------------------------------**



**Solution:** 





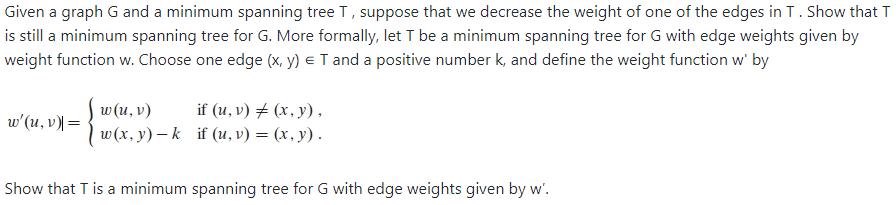
**Solution:** 

**-----------------------------------------------------------------------------**



**Solution:** 

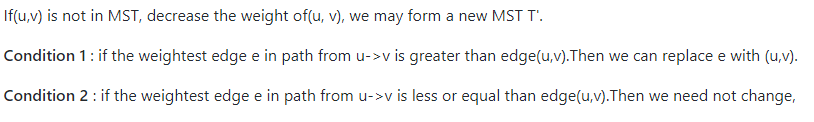
----------------------------------------------------------------------------------------

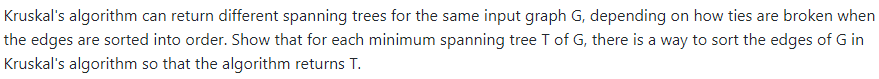


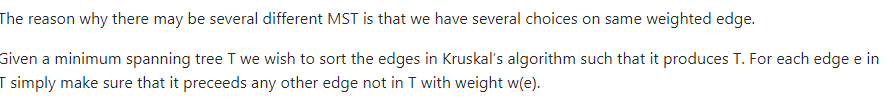
**Solution:** 

**-----------------------------------------------------------------------------**



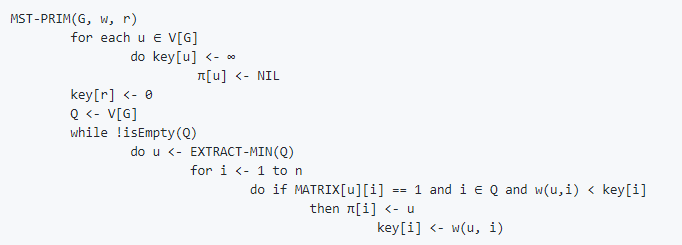
**Solution:** 

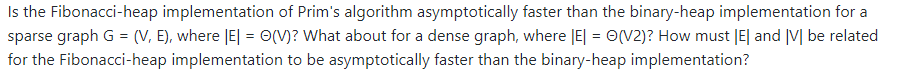


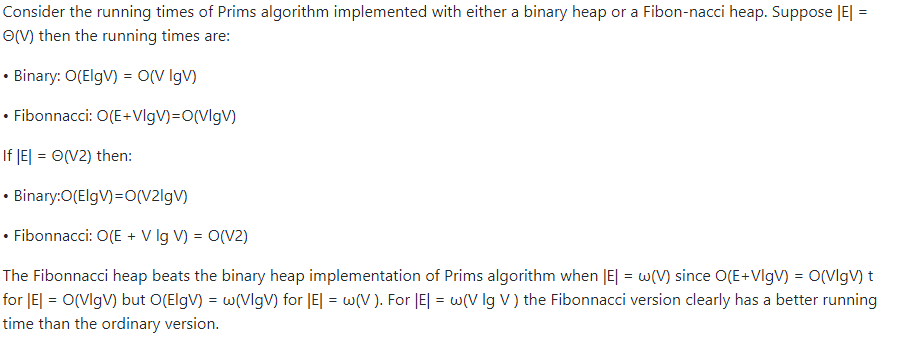
**Solution:** 

------------------------------------------------------------------------------------

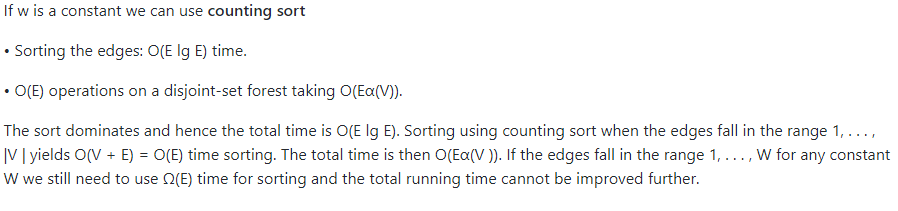


**Solution:** 



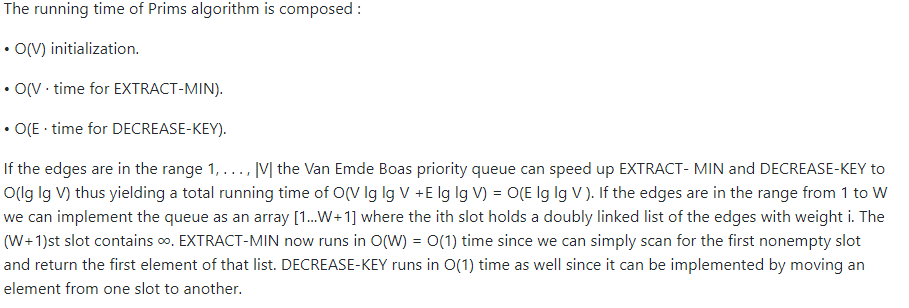
**Solution:** 



**Solution:** 

**-------------------------------------------------------------------------**



**Solution:** 

**-----------------------------------------------------------------**

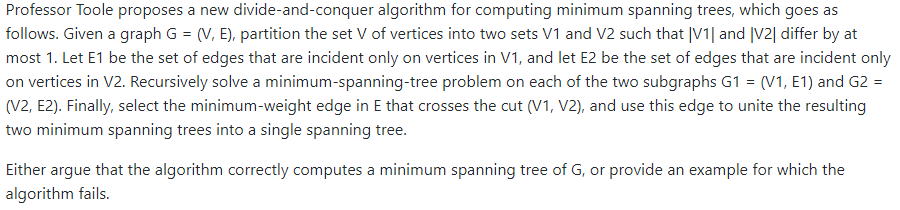


**Solution:**

**--------------**



**Solution:** 



**Solution:** 