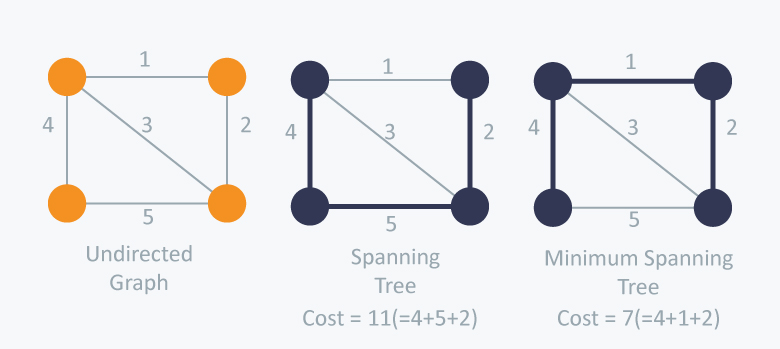
**Minimum Spanning Tree**

What is a Spanning Tree?  
Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

What is a Minimum Spanning Tree?  
The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees. Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

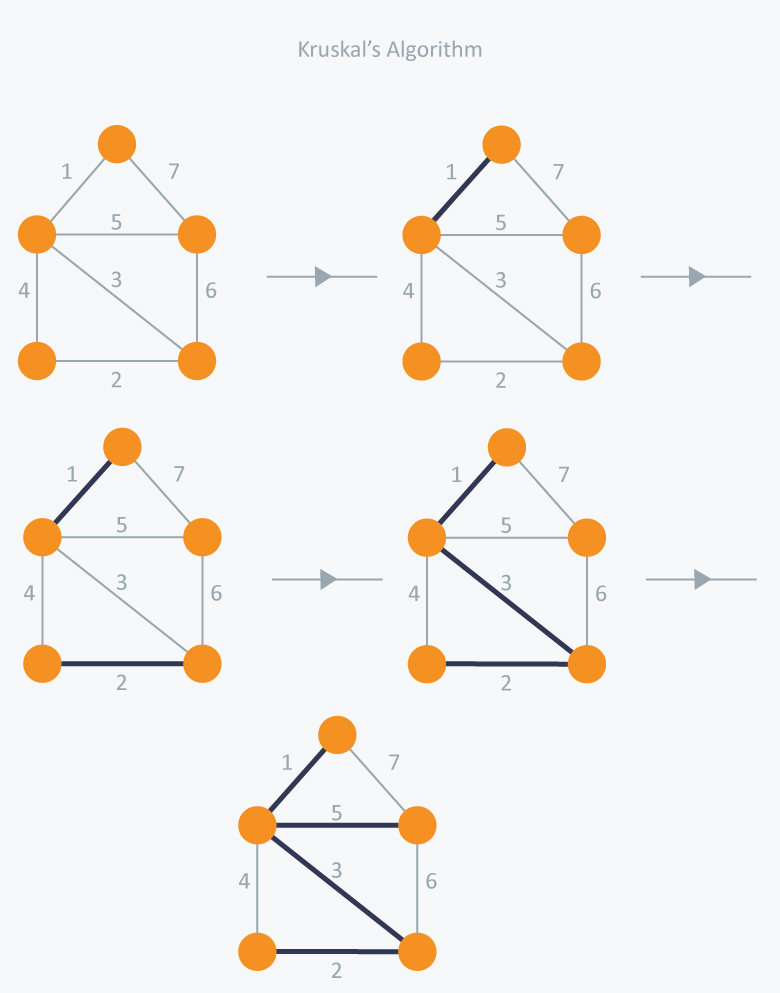
**Kruskal’s Algorithm**

Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

**Algorithm Steps:**

* Sort the graph edges with respect to their weights.
* Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
* Only add edges which doesn't form a cycle , edges which connect only disconnected components.

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is **"Disjoint Sets":**   
Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

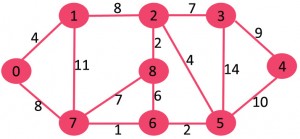


In Kruskal’s algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can’t have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 ( = 1 + 2 + 3 + 5).

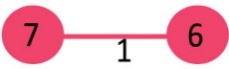
Below are the steps for finding MST using Kruskal’s algorithm

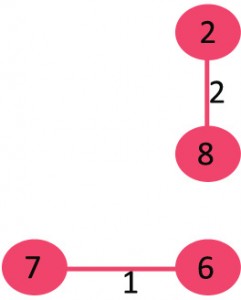
**1.** Sort all the edges in non-decreasing order of their weight.  
**2.** Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.  
**3.** Repeat step#2 until there are (V-1) edges in the spanning tree.

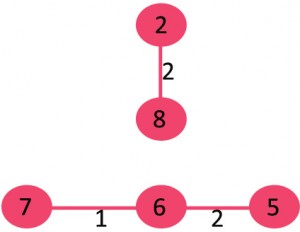
The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.

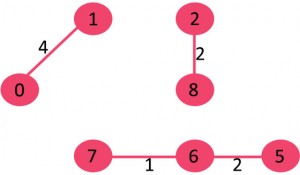
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-0.jpg)

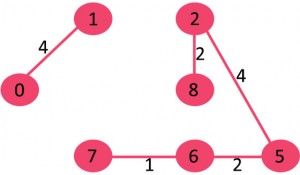
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

Now pick all edges one by one from sorted list of edges  
**1.** *Pick edge 7-6:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-1.jpg)

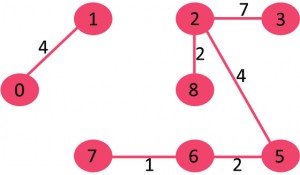
**2.** *Pick edge 8-2:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-2.jpg)

**3.** *Pick edge 6-5:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-3.jpg)

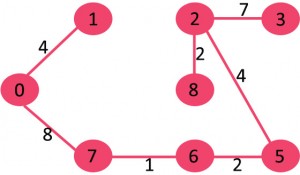
**4.** *Pick edge 0-1:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-4.jpg)

**5.** *Pick edge 2-5:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-5.jpg)

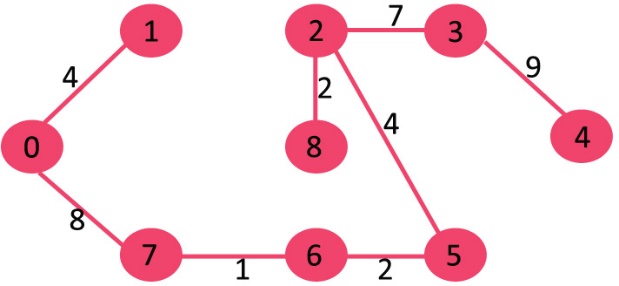
**6.** *Pick edge 8-6:* Since including this edge results in cycle, discard it.

**7.** *Pick edge 2-3:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-6.jpg)

**8.** *Pick edge 7-8:* Since including this edge results in cycle, discard it.

**9.** *Pick edge 0-7:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-7.jpg)

**10.** *Pick edge 1-2:* Since including this edge results in cycle, discard it.

**11.** *Pick edge 3-4:* No cycle is formed, include it.  
[](https://www.geeksforgeeks.org/wp-content/uploads/fig8new.jpeg)

Since the number of edges included equals (V – 1), the algorithm stops here.

**Time Complexity:** O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost O(V2), so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV).

**Implementation:**   
#include <iostream>  
#include <vector>  
#include <utility>  
#include <algorithm>  
using namespace std;

const int MAX = 1e4 + 5;

int id[MAX], nodes, edges;

pair <long long, pair<int, int> > p[MAX];

void initialize()

{

for(int i = 0;i < MAX;++i)

id[i] = i;

}

int root(int x)

{

while(id[x] != x)

{

id[x] = id[id[x]];

x = id[x];

}

return x;

}

void union1(int x, int y)

{

int p = root(x);

int q = root(y);

id[p] = id[q];

}

long long kruskal(pair<long long, pair<int, int> > p[])

{

int x, y;

long long cost, minimumCost = 0;

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

{

// Selecting edges one by one in increasing order from the beginning

x = p[i].second.first;

y = p[i].second.second;

cost = p[i].first;

// Check if the selected edge is creating a cycle or not

if(root(x) != root(y))

{

minimumCost += cost;

union1(x, y);

}

}

return minimumCost;

}

int main()

{

int x, y;

long long weight, cost, minimumCost;

initialize();

cin >> nodes >> edges;

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

{

cin >> x >> y >> weight;

p[i] = make\_pair(weight, make\_pair(x, y));

}

// Sort the edges in the ascending order

sort(p, p + edges);

minimumCost = kruskal(p);

cout << minimumCost << endl;

return 0;

}

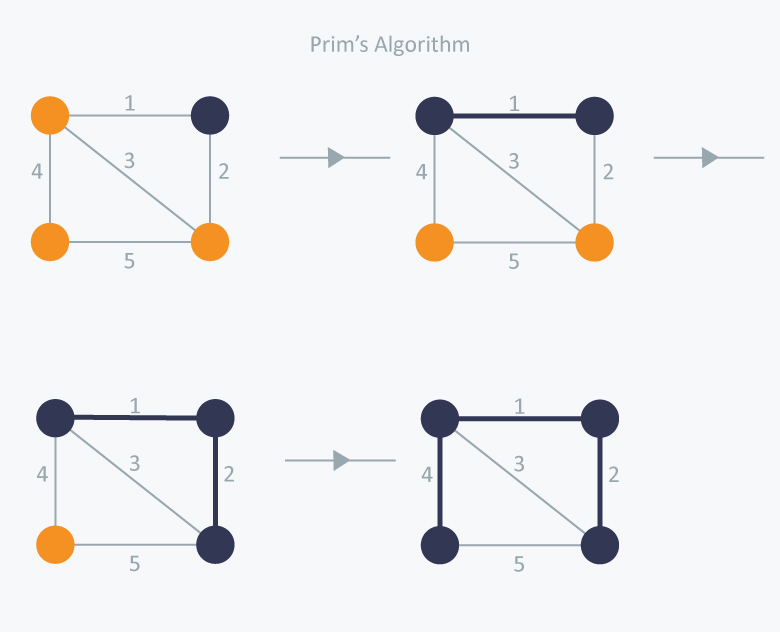
**Prim’s Algorithm**

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

**Algorithm Steps:**

* Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
* Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
* Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:

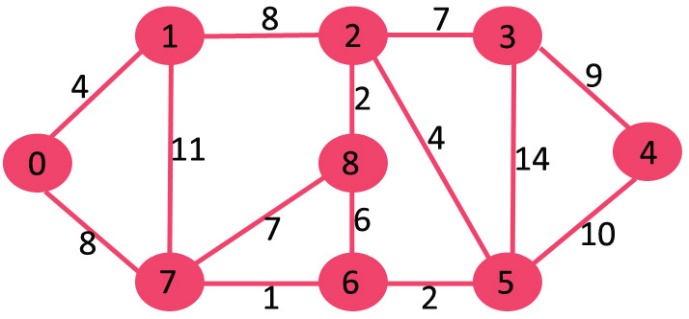


In Prim’s Algorithm, we will start with an arbitrary node (it doesn’t matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim’s algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can’t choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost 7 ( = 1 + 2 +4).

***How does Prim’s Algorithm Work?*** The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning*Tree. And they must be connected with the minimum weight edge to make it a *Minimum*Spanning Tree.

***Algorithm***  
**1)** Create a set *mstSet* that keeps track of vertices already included in MST.  
**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.  
**3)** While mstSet doesn’t include all vertices  
….**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.  
….**b)** Include *u* to mstSet.  
….**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

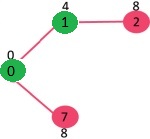
The idea of using key values is to pick the minimum weight edge from [cut](http://en.wikipedia.org/wiki/Cut_(graph_theory)). The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

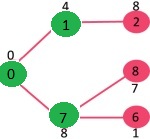
Let us understand with the following example:  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.

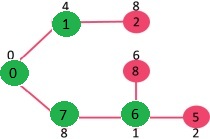
[](https://www.geeksforgeeks.org/wp-content/uploads/MST1.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST2.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).  
[](https://www.geeksforgeeks.org/wp-content/uploads/MST3.jpg)

Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST4.jpg)

We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST5.jpg)

***How to implement the above algorithm?***  
We use a boolean array mstSet[] to represent the set of vertices included in MST. If a value mstSet[v] is true, then vertex v is included in MST, otherwise not. Array key[] is used to store key values of all vertices. Another array parent[] to store indexes of parent nodes in MST. The parent array is the output array which is used to show the constructed MST.

Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list, then the time complexity of Prim’s algorithm can be reduced to O(E log V) with the help of binary heap.

**Implementation:**

#include <iostream>

#include <vector>

#include <queue>

#include <functional>

#include <utility>

using namespace std;

const int MAX = 1e4 + 5;

typedef pair<long long, int> PII;

bool marked[MAX];

vector <PII> adj[MAX];

long long prim(int x)

{

priority\_queue<PII, vector<PII>, greater<PII> > Q;

int y;

long long minimumCost = 0;

PII p;

Q.push(make\_pair(0, x));

while(!Q.empty())

{

// Select the edge with minimum weight

p = Q.top();

Q.pop();

x = p.second;

// Checking for cycle

if(marked[x] == true)

continue;

minimumCost += p.first;

marked[x] = true;

for(int i = 0;i < adj[x].size();++i)

{

y = adj[x][i].second;

if(marked[y] == false)

Q.push(adj[x][i]);

}

}

return minimumCost;

}

int main()

{

int nodes, edges, x, y;

long long weight, minimumCost;

cin >> nodes >> edges;

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

{

cin >> x >> y >> weight;

adj[x].push\_back(make\_pair(weight, y));

adj[y].push\_back(make\_pair(weight, x));

}

// Selecting 1 as the starting node

minimumCost = prim(1);

cout << minimumCost << endl;

return 0;

}

**Time Complexity:**  
The time complexity of the Prim’s Algorithm is O((V+E)logV)  
because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

**Applications of Minimum Spanning Tree Problem**Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices. MST is fundamental problem with diverse applications.

**Network design.**  
*– telephone, electrical, hydraulic, TV cable, computer, road*  
The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn’t a tree you can always remove some edges and save money.

**Approximation algorithms for NP-hard problems.**  
*–* [*traveling salesperson problem*](http://en.wikipedia.org/wiki/Travelling_salesman_problem)*,* [*Steiner tree*](http://en.wikipedia.org/wiki/Steiner_tree_problem)  
A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.Note that if you have a path visiting all points exactly once, it’s a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree. So in general the MST weight is less than the TSP weight, because it’s a minimization over a strictly larger set.On the other hand, if you draw a path tracing around the minimum spanning tree, you trace each edge twice and visit all points, so the TSP weight is less than twice the MST weight. Therefore this tour is within a factor of two of optimal.

**Indirect applications.**  
– max bottleneck paths  
– LDPC codes for error correction  
– image registration with Renyi entropy  
– learning salient features for real-time face verification  
– reducing data storage in sequencing amino acids in a protein  
– model locality of particle interactions in turbulent fluid flows  
– autoconfig protocol for Ethernet bridging to avoid cycles in a network

**Cluster analysis**  
k clustering problem can be viewed as finding an MST and deleting the k-1 most  
expensive edges.