# Minimal Forest

locked

Problem | Submissions | Leaderboard | Discussions

На входа ще получите несвързан претеглен мултиграф. Трябва да намерите тежестта на минималната покриваща гора. За всяка компонента на графа с 1 връх тази тежест е 0. За всяка компонента с повече от 1 върха тази тежест е тежестта на някое минимално покриващо дърво на компонентата.

**Input Format**

На първият ред ще получите $N$ $M$

$N$ е броят на върховете в графа. Върховете са номерирани с числа от $0$ до $N-1$.

$M$ е броят на ребрата.

На следващите $M$ реда ще получите $M$ ребра $A$ $B$ $W$, където $A$ и $B$ са краищата на ребро, а $W$ е тежестта му.

**Constraints**

$N \leq 20000$

$M \leq 20000$

$W \leq 1000$

**Output Format**

На изхода изведете едно число - тежестта на минималната покриваща гора.

**Sample Input 0**

```
3 4
0 1 1
1 0 4
0 2 7
1 2 2
```

**Sample Output 0**

```
3
```

**Explanation 0**

Всички върхове са свързани помежду си, т.е имаме 1 компонента на свързаност. За да свържем 3 върха са ни необходими 2 ребра. Най-леките ребра, които ни вършат работа са с тежест 1 и 2. 1 + 2 = 3

**Sample Input 1**

```
5 4
0 1 5
2 3 1
2 4 1
4 3 10
```

**Sample Output 1**

```
7
```

**Explanation 1**

Графът има 2 компоненти на свързаност: 0-1, 2-3-4. Минималната тежест на покриващо дърво за 0-1 е 5, а за 2-3-4 е 2. 5 + 2 = 7

Submissions: 102
Max Score: 100
Difficulty: Easy

Rate This Challenge:

More

Current Buffer (saved locally, editable) C++

```cpp
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;
struct Pair{
  int index;
  int weight;
};
struct Node{
  list<Pair> neighbours;
  bool hasNeighbour(int index){
    for(auto neighbour:neighbours){
      if(neighbour.index==index){
        return true;
      }
    }
    return false;
  }
  void addNeighbour(int index,int weight){
    neighbours.push_back({index,weight});
  }
};
struct Edge{
  int from;
  int to;
  int weight;
  bool operator<(const Edge& rhs) const {
    return weight < rhs.weight;
  }
};
struct Graph{
  vector<Node> nodes;
  Graph(int nodeCount=0){
    nodes.resize(nodeCount);
  };
  void connect(int from,int to,int weight){
      nodes[from].addNeighbour(to,weight);
  }
  vector<Edge> getAllEdges() const{
    vector<Edge> edges;
    for(int from=0;from<nodes.size();from++){
      for(auto neighbour:nodes[from].neighbours){
        int to=neighbour.index;
        int weight=neighbour.weight;

          edges.push_back({from,to,weight});

      }
    }
    return edges;
  }
  int kruskal(){
    if(nodes.size()<=1){
      return 0;
    }
    vector<Edge> allEdges=getAllEdges();
    sort(allEdges.begin(),allEdges.end());
    list<Edge> tree;
    vector<int> components;
    components.resize(nodes.size());

    for(int i=0;i<components.size();i++){
      components[i]=i;
      }
    for(auto edge:allEdges){
      if(components[edge.from]!=components[edge.to]){
        tree.push_back(edge);
        int oldComponent=components[edge.from];
        int newComponent=components[edge.to];
        for(int i=0;i<components.size();i++){
          if(components[i]==oldComponent){
            components[i]=newComponent;
          }
        }
      }
    }
    int result=0;
    for(auto it:tree){
      result+=it.weight;
    }
    return result;
  }
};
int main() {
   int nodeCount;
   int edgeCount;
  cin>>nodeCount;
  cin>>edgeCount;
  int from,to,weight;
  Graph g(nodeCount);
  for(int i=0;i<edgeCount;i++){
    cin>>from;
    cin>>to;
    cin>>weight;
    g.connect(from,to,weight);
  }

  cout<<g.kruskal();
    return 0;
}
```

Line: 1 Col: 1

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