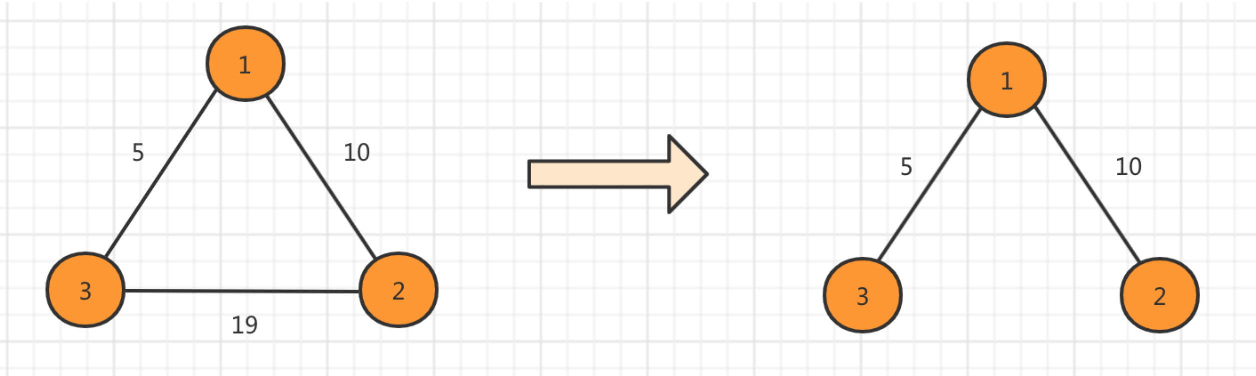
详解: 最小生成树算法

最小生成树（Minimum Spanning Tree, MST）是在一个给定的**无向图**G(V, E)中求一棵树，使得这棵树有图G的所有顶点，且所有边都来自图G中的边，并且满足整棵树的边权之和最小.

最下生成树满足如下性质:

- 最小生成树是树，因此其边数等于顶点数减1，且树内一定没有环;
- 对给定的图，其最小生成树不唯一，但边权之和是唯一的;
- 最小生成树是在无向图上生成的，因此其根结点可以是这个图上的任意一个点。题目一般会指出那个点作为生成树的根结点

示例




```
输入
3 3
1 2 10
1 3 5
3 2 19
输出
15
```

Prim算法

跟Dijkstra算法类似，将顶点分为两个集合，一个是已经在生成树中的顶点集合S，另一个是还未访问的点。

用数组d[]表示各个顶点到集合S的最短距离（只有这里d的含义与Dijkstra算法中d的含义不一样）

```
Prim(G, d[]) {
    初始化;
    for (循环n次) {
        u = 使d[u]最小的还未被访问的顶点的标号;
        记u已被访问
        for (从u出发能到达的所有顶点v){
            if(v未被访问 && 以u为中介点使得v与集合S的最短距离d[v]更优){
                将G[u][v]赋值给v与集合S的最短距离d[v]
            }
        }
    }
}
```

基于邻接矩阵实现的Prim算法

Java

```java
import java.util.Arrays;
import java.util.Scanner;

public class Main{

    private static final int INF = Integer.MAX_VALUE;

    public static void main(String[] args) throws InterruptedException {
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();
        int m = input.nextInt();
        int[][] graph = new int[n + 1][n + 1];
        for (int i = 0; i < n + 1; i++) Arrays.fill(graph[i], INF);
        int u, v, w;
        for (int i = 0; i < m; i++) {
            u = input.nextInt();
            v = input.nextInt();
            w = input.nextInt();
            graph[u][v] = graph[v][u] = w;
        }
        int start = 1; //指定一个结点作为生成树的根结点
        int res = prim(graph, n, 1);
        System.out.println(res);
    }

    public static int prim(int[][] graph, int n, int start) {
        int[] d = new int[n + 1];
        boolean[] visit = new boolean[n + 1];
        Arrays.fill(d, INF);
        d[start] = 0;
```

```
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            int u = -1, MIN = INF;
            for (int j = 1; j <= n; j++) {
                if (!visit[j] && d[j] < MIN) {
                    u = j;
                    MIN = d[j];
                }
            }
            if (u == -1) return -1;
            visit[u] = true;
            ans += d[u]; // 将与集合s距离最小的边加入最小生成树
            for (int v = 1; v <= n; v++) {
                if (!visit[v] && graph[u][v] < INF && graph[u][v] < d[v]) {
                    d[v] = graph[u][v];
                }
            }
        }
        return ans;
    }
}
```

C++

```
#include <iostream>
#include <vector>
using namespace std;

const int N = 100;
const int INF = INT_MAX;

int prim(vector<vector<int>>& graph, int n, int start) {
    vector<int> d(n + 1, INF);
    vector<bool> visit(n + 1, false);
    d[start] = 0;
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        int u = -1, MIN = INF;
        for (int j = 1; j <= n; j++) {
            if (!visit[j] && d[j] < MIN) {
                u = j;
                MIN = d[j];
            }
        }
        if (u == -1) return -1;
        visit[u] = true;
        ans += d[u];
        for (int v = 1; v <= n; v++) {
```

```
                if (!visit[v] && graph[u][v] < INF && graph[u][v] < d[v]) {
                    d[v] = graph[u][v];
                }
            }
        }
    return ans;
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> graph(n + 1, vector<int>(n + 1, INF));
    int u, v, w;
    for (int i = 0; i < m; i++) {
        cin >> u >> v >> w;
        graph[u][v] = graph[v][u] = w;
    }
    int start = 1;
    int res = prim(graph, n, start);
    cout << res << endl;
    return 0;
}
```

基于邻接表实现的Prim算法

Java

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

public class Main{

    private static final int INF = Integer.MAX_VALUE;

    public static void main(String[] args) throws InterruptedException {
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();
        int m = input.nextInt();
        List<List<Node>> graph = new ArrayList<>(n + 1);
        for (int i = 0; i < n + 1; i++) graph.add(new ArrayList<>());
        int u, v, w;
        for (int i = 0; i < m; i++) {
            u = input.nextInt();
            v = input.nextInt();
            w = input.nextInt();
```

```java
            graph.get(u).add(new Node(v, w));
            graph.get(v).add(new Node(u, w));
        }
        int start = 1;
        int res = prim(graph, n, start);
        System.out.println(res);
    }

    public static int prim(List<List<Node>> graph, int n, int start) {
        int[] d = new int[n + 1];
        boolean[] visit = new boolean[n + 1];
        Arrays.fill(d, INF);
        d[start] = 0;
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            int u = -1, MIN = INF;
            for (int j = 1; j <= n; j++) {
                if (!visit[j] && d[j] < MIN) {
                    u = j;
                    MIN = d[j];
                }
            }
            if (u == -1) return -1;
            visit[u] = true;
            ans += d[u];
            for (int k = 0; k < graph.get(u).size(); k++) {
                int v = graph.get(u).get(k).v; //从u能到达的顶点v
                if (!visit[v] && graph.get(u).get(k).dis < d[v]) {
                    d[v] = graph.get(u).get(k).dis;
                }
            }
        }
        return ans;
    }

    static class Node{
        int v, dis;
        public Node(int v, int dis) {
            this.v = v;
            this.dis = dis;
        }
    }
}
```

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;
```

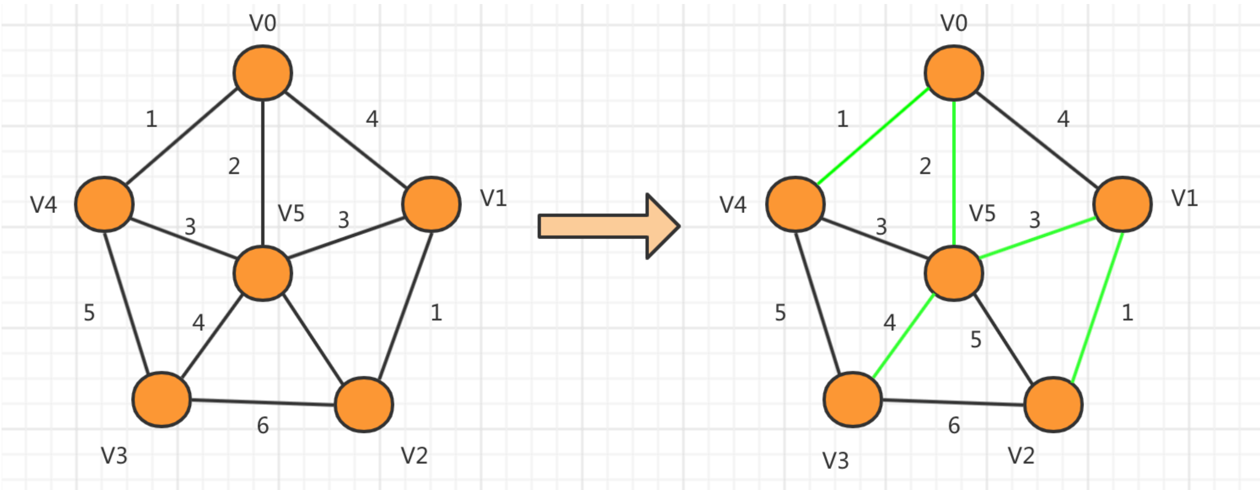
```
const int N = 100;
const int INF = INT_MAX;

struct Node{
    int v, dis; //v为边的目标顶点，dis为边权
    Node(int _v, int _dis): v(_v), dis(_dis) {}
};

int prim(vector<vector<Node*>>& graph, int n, int start) {
    vector<int> d(n + 1, INF);
    vector<bool> visit(n + 1, false);
    d[start] = 0;
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        int u = -1, MIN = INF;
        for (int j = 1; j <= n; j++) {
            if (!visit[j] && d[j] < MIN) {
                u = j;
                MIN = d[j];
            }
        }
        if (u == -1) return -1;
        visit[u] = true;  // 将u放入集合s中
        ans += d[u];
        // 遍历从u能到达的顶点，更新它们到达集合s的最短距离
        for (int k = 0; k < graph[u].size(); k++) {
            int v = graph[u][k]->v; //从u能到达的顶点
            if (!visit[v] && graph[u][k]->dis < d[v]) {
                d[v] = graph[u][k]->dis;
            }
        }
    }
    return ans;
}

int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<Node*>> graph(n + 1);
    int u, v, w;
    for (int i = 0; i < m; i++) {
        cin >> u >> v >> w;
        graph[u].push_back(new Node(v, w));
        graph[v].push_back(new Node(u, w));
    }
    int start = 1;
    int res = prim(graph, n, start);
    cout << res << endl;
```

```
    return 0;
}
```

Kruskal算法

采用**边贪心**的策略，思想很简单：

1. 对所有边权按从小到大进行排序；
2. 按边权从小到大测试所有边，如果当前测试边所连接的两个顶点不在同一个连通块中，则把这条边加入当前最小生成树中；否则，将边舍弃；
3. 执行步骤2，知道最小生成树中的边数等于总顶点数减1或是测试完所有边时结束。而当结束时如果最小生成树中的边数小于总顶点树减1，说明该图不连通.




```
int kruskal() {
    令最小生成树的边权之和为 ans, 最小生成树的当前边数为Num_Edge;
    将所有边按边权从小到大排序;
    for (从小到大枚举所有边){
        if (当前测试边的两个端点在不同的连通块中) {
            将测试边加入最小生成树中;
            ans += 测试边的边权;
            最小生成树的当前边数Num_Edge加1;
            当边数等于顶点数减1时结束循环;
        }
    }
    return ans;
}
```

伪代码中有两个细节私护不太直观，即：

1. 如何判断测试边的两个端点在不同的连通块中；
2. 如何将测试边加入最小生成树中。

把每个连通块当作一个集合，那么就可以把问题转换为判断两个端点是否在同一个集合中，这正好可以使用**并查集**。

并查集可以通过查询两个结点所在集合的根结点判断是否来自同一集合；

将测试边加入连通块可以通过将两个端点所在集合合并，也正好利用了并查集的合并特性；

Java

```java
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

public class Main{
    public static int[] father;
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int n = input.nextInt();
        int m = input.nextInt();
        Edge[] edges = new Edge[m];
        father = new int[n + 1];
        for (int i = 0; i < n; i++) father[i] = i;
        int u, v, w;
        for (int i = 0; i < m; i++) {
            u = input.nextInt();
            v = input.nextInt();
            w = input.nextInt();
            edges[i] = new Edge(u, v, w);
        }
        int res = kruskal(edges, n, m);
        System.out.println(res);
    }

    public static int kruskal(Edge[] edges, int n, int m) {
        Arrays.sort(edges, new Comparator<Edge>() {
            @Override
            public int compare(Edge o1, Edge o2) {
                if (o1.w > o2.w) return 1;
                else if (o1.w == o2.w) return 0;
                else return -1;
            }
        });
        int ans = 0;
        int numEdges = 0;
        for (int i = 0; i < m; i++) {
            int faU = findFather(edges[i].u);
            int faV = findFather(edges[i].v);
            if (faU != faV) {
                ans += edges[i].w;
                father[faU] = faV;
```

```
                numEdges++;
                if (numEdges == n - 1) break;
            }
        }
        if (numEdges != n - 1) return -1;
        else return ans;
    }

    public static int findFather(int a) {
        int x = a;
        while (x != father[x]) {
            x = father[x];
        }
        // 压缩路径
        while (a != father[a]) {
            int z = a;
            a = father[a];
            father[z] = x;
        }
        return x;
    }

    static class Edge{
        int u, v, w;
        public Edge(int u, int v, int w) {
            this.u = u;
            this.v = v;
            this.w = w;
        }
    }
}
```

C++

```
#include <iostream>
#include <vector>
using namespace std;

const int MAXV = 100;
const int MAXE = 100;

struct edge{
    int u, v; //边的两个断点
    int cost; //边权
}E[MAXE];  //最多有MAXE条边

bool cmp(edge a, edge b) {
```

```cpp
    return a.cost < b.cost;
}

// 并查集部分
int father[MAXV]; //并查集数组
int findFather(int x) { //并查集查询函数
    int a = x;
    while (x != father[x]) {
        x = father[x];
    }
    // 路径压缩
    while (a != father[a]) {
        int z = a;
        a = father[a];
        father[z] = x;
    }
    return x;
}

// 返回最小生成树的边权之和，参数n为顶点个数，m为图的边数
int kruskal(int n, int m) {
    int ans = 0, Num_Edge = 0;
    for (int i = 0; i < n; i++) {
        father[i] = i;
    }
    sort(E, E + m, cmp);
    for (int i = 0; i < m; i++) { //枚举所有边
        int faU = findFather(E[i].u);
        int faV = findFather(E[i].v);
        if (faU != faV) { // 不在同一个连通块，将边加入最小生成树
            father[faU] = faV;
            ans += E[i].cost;
            Num_Edge++;
            if (Num_Edge == n - 1) break;
        }
    }
    if (Num_Edge != n - 1) return -1; //无法连通返回-1
    else return ans;
}

int main() {
    int n, m;
    cin >> n >> m;
    int u, v, w;
    for (int i = 0; i < m; i++) {
        cin >> u >> v >> w;
        E[i].u = u;
        E[i].v = v;
        E[i].cost = w;
```

```
    }
    int res = kruskal(n, m);
    cout << res << endl;
    return 0;
}
```