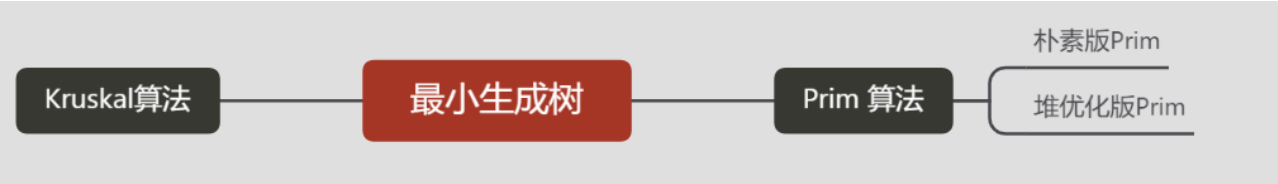
1：最小生成树知识大纲：




2：朴素版Prim 算法：(用于稠密图，无向图，O (n ^ 2))

基本思路：

首先定义一个集合 s，用来存储已经在连通块中的所有点

```
dist[i] <---- 无穷大
for(int i = 0; i < n; i++){
    t <-- 找到集合外距离最近的点
    st[t] = true;
    用 t 更新其他点到集合的距离
}
```

prim 算法做的事情是：给定一个无向图，在图中选择若干条边把图的所有节点连起来。要求边长之和最小。在图论中，叫做求最小生成树

prim 算法采用的是一种贪心的策略

每次将离连通部分的最近的点和点对应的边加入的连通部分，连通部分逐渐扩大，最后将整个图连通起来，并且边长之和最小

例题：

给定一个 $n$ 个点 $m$ 条边的无向图，图中可能存在重边和自环，边权可能为负数。

求最小生成树的树边权重之和，如果最小生成树不存在则输出 `impossible`。

给定一张边带权的无向图 $G=(V,E)$，其中 $V$ 表示图中点的集合，$E$ 表示图中边的集合，$n=|V|$，$m=|E|$。

由 $V$ 中的全部 $n$ 个顶点和 $E$ 中 $n-1$ 条边构成的无向连通子图被称为 $G$ 的一棵生成树，其中边的权值之和最小的生成树被称为无向图 $G$ 的最小生成树。

**输入格式**

第一行包含两个整数 $n$ 和 $m$。

接下来 $m$ 行，每行包含三个整数 $u,v,w$，表示点 $u$ 和点 $v$ 之间存在一条权值为 $w$ 的边。

**输出格式**

共一行，若存在最小生成树，则输出一个整数，表示最小生成树的树边权重之和，如果最小生成树不存在则输出 `impossible`。

```java
import java.util.*;
public class Main{
    public static int N = 510;
    public static int n, m;
    public static int max = 0x3f3f3f3f;
    public static int[] dist = new int[N];
    public static int[][] g = new int[N][N];
    public static boolean[] st = new boolean[N];
    public static int prim(){
        for(int i = 1; i <= n; i++){
            dist[i] = max;
        }
        dist[1] = 0;
        int res = 0;
        for(int i = 0; i < n; i++){
            int t = -1;
            for(int j = 1; j <= n; j++){
                if(!st[j] && (t == -1 || dist[j] < dist[t])){
                    t = j;
                }
            }
            if(dist[t] == max){
                return max;
            }
            res += dist[t];
            st[t] = true;
            for(int j = 1; j <= n; j++){
                dist[j] = Math.min(dist[j], g[t][j]);
            }
        }
        return res;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
```

```
                g[i][j] = max;
            }
        }
        while(m -- > 0){
            int u = sc.nextInt();
            int v = sc.nextInt();
            int w = sc.nextInt();
            g[u][v] = g[v][u] = Math.min(g[u][v], w);
        }
        int t = prim();
        if(t == max){
            System.out.println("impossible");
        }else{
            System.out.println(t);
        }
    }
}
```

3：Kruskal 算法：（用于稀疏图，O ( mlogm))

基本思路：

```
将所有边按权重大小从小到大排序
枚举每条边a，b，权重为c
如果a，b 不连通，将这条边加入到集合中
```

判断是否会产生回路的方法为：使用并查集

在初始状态下给各个个顶点在不同的集合中

遍历过程的每条边，判断这两个顶点的是否在一个集合中

如果边上的这两个顶点在一个集合中，说明两个顶点已经连通，这条边不要，如果不在一个集合中，则要这条边

例如，Kruskal 算法查找如图对应的最小生成树，需要经历以下几个步骤：

例题：

```
import java.util.*;
import java.io.*;
```

```java
public class Main{
    public static int N = 100010;
    public static int M = N * 2;
    public static int n, m;
    public static int max = 0x3f3f3f3f;
    public static int[] p = new int[N];
    public static PII[] q = new PII[M];
    public static int find(int x){
        if(p[x] != x){
            p[x] = find(p[x]);
        }
        return p[x];
    }
    public static void main(String[] args) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] str = br.readLine().split(" ");
        n = Integer.parseInt(str[0]);
        m = Integer.parseInt(str[1]);
        for(int i = 0; i < m; i++){
            String[] st = br.readLine().split(" ");
            int u = Integer.parseInt(st[0]);
            int v = Integer.parseInt(st[1]);
            int w = Integer.parseInt(st[2]);
            q[i] = new PII(u, v, w);
        }
        Arrays.sort(q, 0, m);
        int res = 0;
        int cnt = 0;
        for(int i = 1; i <= n; i++){
            p[i] = i;
        }
        for(int i = 0; i < m; i++){
            PII node = q[i];
            int u = node.u;
            int v = node.v;
            int w = node.w;
            if(find(u) != find(v)){
                p[find(u)] = find(v);
                res += w;
                cnt ++;
```

```
            }
        }
        if(cnt < n - 1){
            System.out.println("impossible");
        }else{
            System.out.println(res);
        }
    }
}
class PII implements Comparable<PII>{
    int u, v, w;
    public PII(int u, int v, int w){
        this.u = u;
        this.v = v;
        this.w = w;
    }
    public int compareTo(PII o){
        return Integer.compare(w, o.w);
    }
}
```

4：二分图知识大纲：

5：染色法： （O（n + m））

将所有点分成两个集合，使得所有边只出现在集合之间，就是二分图

二分图：一定不含有奇数环，可能包含长度为偶数的环， 不一定是连通图

dfs 版本：

代码思路：

染色可以使用1和2区分不同颜色，用0表示未染色

遍历所有点，每次将未染色的点进行dfs, 默认染成1或者2

由于某个点染色成功不代表整个图就是二分图, 因此只有某个点染色失败才能立刻break / return

染色失败相当于存在相邻的2个点染了相同的颜色

例题：

```java
import java.util.*;
public class Main{
    public static int N = 100010;
    public static int M = N * 2;
    public static int n, m, idx;
    public static int[] h = new int[N];
    public static int[] e = new int[M];
    public static int[] ne = new int[M];
    public static int[] color = new int[M];
    public static void add(int u, int v){
        e[idx] = v;
        ne[idx] = h[u];
        h[u] = idx;
        idx ++;
    }
    public static boolean dfs(int u, int c){
        color[u] = c;
        for(int i = h[u]; i != -1; i = ne[i]){
            int j = e[i];
            if(color[j] == 0){
                if(!dfs(j, 3 - c)){
                    return false;
                }
            }else if(color[j] == c){
                return false;
            }
        }
        return true;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            h[i] = -1;
        }
        while(m -- > 0){
            int u = sc.nextInt();
```

```
            int v = sc.nextInt();
            add(u, v);
            add(v, u);
        }
        boolean flag = true;
        for(int i = 1; i <= n; i++){
            if(color[i] == 0){
                if(!dfs(i, 1)){
                    flag = false;
                    break;
                }
            }
        }
        if(flag){
            System.out.println("Yes");
        }else{
            System.out.println("No");
        }
    }
}
```

6：匈牙利算法：（最坏情况下为O（mn），但实际上运行时间远小于O（mn））

例题：

```
import java.util.*;
public class Main{
    public static int N = 510;
    public static int M = 100010;
    public static int n1, n2, m, idx;
    public static int[] h = new int[N];
    public static int[] e = new int[M];
    public static int[] ne = new int[M];
    public static boolean[] st = new boolean[M];
    public static int[] match = new int[M];
    public static void add(int u, int v){
        e[idx] = v;
        ne[idx] = h[u];
        h[u] = idx;
```

```
        idx ++;
    }
    public static boolean find(int x){
        for(int i = h[x]; i != -1; i = ne[i]){
            int j = e[i];
            if(!st[j]){
                st[j] = true;
                if(match[j] == 0 || find(match[j])){
                    match[j] = x;
                    return true;
                }
            }
        }
        return false;
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n1 = sc.nextInt();
        n2 = sc.nextInt();
        m = sc.nextInt();
        Arrays.fill(h, -1);
        while(m -- > 0){
            int u = sc.nextInt();
            int v = sc.nextInt();
            add(u, v);
        }
        int res = 0;
        for(int i = 1; i <= n1; i++){
            Arrays.fill(st, false);
            if(find(i)){
                res ++;
            }
        }
        System.out.println(res);
    }
}
```