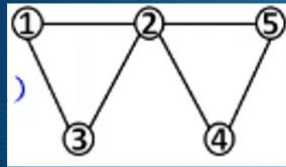


## 课堂内容

### 1、最短路径——bfs



#### 【题目描述】

每个顶点都站了一位同学，john站在点1的位置，他想知道其他同学距离他的最短距离分别是多少？

#### 【输入】

第一行两个值分别为n，m表示n个顶点，m条边  
接下来m行，每行两个数据，起点，终点

#### 【输出】

从顶点2至n距离1的最短距离

#### 【输入样例】

5 6

1 2

1 3

2 3

2 4

2 5

4 5

#### 【输出样例】

1 1 2 2

模板：

```
32 void Bfs() {
33     queue<int> que;
34     que.push(1);
35     _deep[1] = 1;
36     while(que.size() > 0) {
37         int x = que.front();
38         for(int i = _head[x]; i != 0; i = _e[i].next) {
39             int y = _e[i].to;
40             if(_deep[y] != 0) {
41                 continue;
42             }
43             _deep[y] = _deep[x] + 1;
44             que.push(y);
45         }
46         que.pop();
47     }
48 }
```

例题 1: P5663

题解：

第一问：为什么要用 bfs？

(1) 没有权值

(2) 相邻的点需要提供零件/原材料 → 走最近的点 → bfs

第二问：怎么利用最短路解决此问题？L 很大怎么办？

(1)  $L$  很小, 员工最多只需要提供一次零件/原材料时,  $L$  等于该点到 1 点的最短奇数距离或者最短偶数距离

(2)  $L$  很大, 员工需要提供多次零件/原材料时, 有如下规律:

$L$  为奇数,  $L$  大于此点到 1 点的最短奇数距离, 那么 1 一定需要提供原材料

$L$  为偶数,  $L$  大于此点到 1 点的最短偶数距离, 那么 1 一定需要提供原材料

代码实现:

(1) 第一步: 定义初始化

```
//记录点的位置和长度
struct node{
    int pos;
    int dis;
};
int _men[100010][3]; //记录这个点到1的最短路径长度 (最短奇数路径存到[][1], 偶数存到[][2])
```

(2) 存图——邻接表

```
void ParseIn() {
    int u = 0;
    int v = 0;
    cin >> _n >> _m >> _q;
    for(int i = 1; i <= _m; i++) {
        cin >> u >> v;
        //无向图
        AddEdge(u,v);
        AddEdge(v,u);
    }
    memset(_men, 127, sizeof(_men)); //初始化为无穷大
    Bfs();
}
```

(3) 第三步: bfs 求最短奇数距离和最短偶数距离

```

void Bfs() {
    int NextPoint = 0; //下一个点
    int curPos = 0; //当前位置
    int curDis = 0; //当前距离
    _que.push((node){1,0});
    while(!_que.empty()) {
        int curPos = _que.front().pos; //获取队头元素--这个点的编号
        int curDis = _que.front().dis; //获取队头元素--到1点的距离
        for(int i = _head[curPos]; i != 0; i = _e[i].next) {
            NextPoint = _e[i].to;
            if(curDis + 1 < _men[NextPoint][1] && (curDis + 1) % 2 != 0) { //到1点的距离为奇数
                _men[NextPoint][1] = curDis + 1; //更新最短奇数距离
                _que.push((node){NextPoint, curDis + 1}); //走到的点入队
            }
            if(curDis + 1 < _men[NextPoint][2] && (curDis + 1) % 2 == 0) { //到1点的距离为偶数
                _men[NextPoint][2] = curDis + 1; //更新最短偶数距离
                _que.push((node){NextPoint, curDis + 1}); //走到的点入队
            }
        }
        _que.pop(); //队头出队
    }
}

```

#### (4) 结果输出

```

void CoreWriteOut() {
    int a = 0;
    int L = 0;
    for(int i = 1; i <= _q; i++) {
        cin >> a >> L;
        if(_head[a] == 0) { //1、这个点没有边，那么一定不需要1提供原材料
            cout << "No" << endl;
        }
        else if(L % 2 == 1 && L >= _men[a][1]) { //2、L为奇数，L需要大于这个点到1点的最短奇数距离
            cout << "Yes" << endl;
        }
        else if(L % 2 == 0 && L >= _men[a][2]) { //3、L为偶数，L需要大于这个点到1点的最短偶数距离
            cout << "Yes" << endl;
        }
        else {
            cout << "No" << endl; //4、其他情况，则走不到
        }
    }
}

```

#### (5) 完整代码

```

1  #include <iostream>
2  #include <queue>
3  #include <cstring>
4  using namespace std;
5  struct edge{
6      int next;
7      int to;
8  } _e[200010];
9  //记录点的位置和长度
10 struct node{
11     int pos;
12     int dis;
13 };

```

```

14 int _men[100010][3]; // 记录这个点到1的最短路径长度 (最短奇数路径存到[][1], 偶数存到[][2])
15 int _head[100010];
16 int _numEdge;
17 int _m, _q, _n;
18 queue<node> _que;
19
20 void AddEdge(int from, int to) {
21     _numEdge++;
22     _e[_numEdge].next = _head[from];
23     _e[_numEdge].to = to;
24     _head[from] = _numEdge;
25 }
26 void Bfs() {
27     int NextPoint = 0; // 下一个点
28     int curPos = 0; // 当前位置
29     int curDis = 0; // 当前距离
30     _que.push((node){1, 0});
31     while(!_que.empty()) {
32         int curPos = _que.front().pos; // 获取队头元素--这个点的编号
33         int curDis = _que.front().dis; // 获取队头元素--到1点的距离
34         for(int i = _head[curPos]; i != 0; i = _e[i].next) {
35             NextPoint = _e[i].to;
36             if(curDis + 1 < _men[NextPoint][1] && (curDis + 1) % 2 != 0) { // 到1点的距离为奇数
37                 _men[NextPoint][1] = curDis + 1; // 更新最短奇数距离
38                 _que.push((node){NextPoint, curDis + 1}); // 走到的点入队
39             }
40             if(curDis + 1 < _men[NextPoint][2] && (curDis + 1) % 2 == 0) { // 到1点的距离为偶数
41                 _men[NextPoint][2] = curDis + 1; // 更新最短偶数距离
42                 _que.push((node){NextPoint, curDis + 1}); // 走到的点入队
43             }
44         }
45         _que.pop(); // 队头出队
46     }
47
48 void ParseIn() {
49     int u = 0;
50     int v = 0;
51     cin >> _n >> _m >> _q;
52     for(int i = 1; i <= _m; i++) {
53         cin >> u >> v;
54         // 无向图
55         AddEdge(u, v);
56         AddEdge(v, u);
57     }
58     memset(_men, 127, sizeof(_men)); // 初始化为无穷大
59     Bfs();
60 }
61
62 void CoreWriteOut() {
63     int a = 0;
64     int L = 0;
65     for(int i = 1; i <= _q; i++) {
66         cin >> a >> L;
67         if(_head[a] == 0) { // 1、这个点没有边，那么一定不需要1提供原材料
68             cout << "No" << endl;
69         }
70         else if(L % 2 == 1 && L >= _men[a][1]) { // 2、L为奇数，L需要大于这个点到1点的最短奇数距离
71             cout << "Yes" << endl;
72         }
73         else if(L % 2 == 0 && L >= _men[a][2]) { // 3、L为偶数，L需要大于这个点到1点的最短偶数距离
74             cout << "Yes" << endl;
75         }
76         else {
77             cout << "No" << endl; // 4、其他情况，则走不到
78         }
79     }
80 }
81 int main() {
82     ParseIn();
83     CoreWriteOut();

```



## 2、多源最短路径——floyed

➤ 任意节点*i*到*j*的最短路径两种可能：

直接从*i*到*j*；

从*i*经过若干个节点*k*到*j*。

1. 在此 $d(i,k)$ 与 $d(k,j)$ 分别是目前为止所知道的*i*到*k*与*k*到*j*的最短距离。
2. 若有 $d(i,j) > d(i,k) + d(k,j)$ ，就表示从*i*出发经过*k*再到*j*的距离要比原来的*i*到*j*距离短，自然把*i*到*j*的 $d(i,j)$ 重写为 $d(i,k) + d(k,j)$ ，每当一个*k*查完了， $d(i,j)$ 就是目前的*i*到*j*的最短距离。
3. 重复这一过程，最后当查完所有的*k*时， $d(i,j)$ 里面存放的就是*i*到*j*之间的最短距离了。

输入：

n m  
u v w  
.....

输出：

打表，每个点到其他点的最短距离

```
模板： 12  #include <cstring>
13  using namespace std;
14  int g[110][110];
15  int n, m;
16  void Floyed(){
17      for(int k = 1; k <= n; k++){
18          for(int i = 1; i <= n; i++){
19              for(int j = 1; j <= n; j++){
20                  if(i != j && i != k && j != k){
21                      g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
22                  }
23              }
24          }
25      }
26  }
27  void print(){
28      for(int i = 1; i <= n; i++){
29          for(int j = 1; j <= n; j++){
30              cout << g[i][j] << " ";
31          }
32          cout << endl;
33      }
34  }
35  int main(){
36      int u = 0, v = 0, w = 0;
37      memset(g, 0x3f, sizeof(g));
38      cin >> n >> m;
39      for(int i = 1; i <= m; i++){
40          cin >> u >> v >> w;
41          g[u][v] = w;
42      }
43      Floyed();
44      print();
45      return 0;
```

## 例题 2: P2047

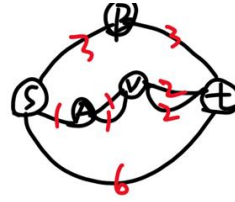
题解:

第一问: 为什么要用 floyd?

- (1) 有权值
- (2) 多源最短路问题

第二问: 怎么理解题意?

$$\sum_{\text{所有 } s, t \text{ 情况}} \frac{\text{经过 } v \text{ 所有 } s, t \text{ 最短路}}{\text{所有 } s, t \text{ 不同最短路}}$$



(1) 求解最短路数量分析:

如果已知  $s \rightarrow v$  的最短路径数量是  $\text{num}[s][v]$  个,  $v \rightarrow t$  的最短路径数量是  $\text{num}[v][t]$  个因此计算  $s \rightarrow t$  的最短路径数量有两种情况:

- ①: 更新了最短的路径, 需要更新  $s \rightarrow t$  的数量: 应用乘法原理:  
 $\text{num}[s][t] = \text{num}[s][v] * \text{num}[v][t]$
- ②: 同之前的最短路径长度相同, 需要累加:  $\text{num}[s][t] += \text{num}[s][v] * \text{num}[v][t]$

(2) 累加的过程

再次通过三次循环来进行判断, 当  $s$  和  $t$  之前的最短路径经过  $k$ , 累加:  
(经过  $k$  的所有  $s$  到  $t$  的最短路) / (所有  $s$  到  $t$  的最短路)

代码实现:

(1) 第一步: 存图的时候,  $\text{num}$  数组也要更新

```
int main(){
    int u = 0, v = 0, w = 0;
    memset(g, 0x3f, sizeof(g));
    cin >> n >> m;
    for(int i = 1; i <= m; i++){
        cin >> u >> v >> w;
        g[u][v] = w;
        g[v][u] = w;
        num[u][v] ++;
        num[v][u] ++;
    }
    Floyd();
    //print();
    return 0;
}
```

(2) 第二步: 找出每两点之间的最短路径的数量, 即更新  $\text{num}$  数组

```

void Floyd(){
    for(int k = 1; k <= n; k++){
        for(int i = 1; i <= n; i++){//s
            for(int j = 1; j <= n; j++){//t
                if(i != j && i != k && j != k){
                    if(g[i][k] + g[k][j] < g[i][j]){//刷新最短路径
                        g[i][j] = g[i][k] + g[k][j];
                        num[i][j] = num[i][k] * num[k][j]; //乘法原理
                    }
                    else if(g[i][k] + g[k][j] == g[i][j]){
                        num[i][j] += num[i][k] * num[k][j];
                    }
                    else {
                        //do nothing
                    }
                }
            }
        }
    }
}

```

(3) 第三步：计算每个点的贡献值，即累加所有：

(经过 k 的所有 s 到 t 的最短路) / (所有 s 到 t 的最短路)

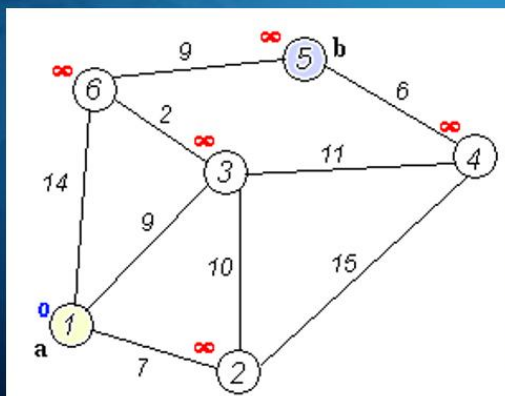
```

for(int k = 1; k <= n; k++){
    double ans = 0;
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            if(i != j && i != k && j != k){
                if( g[i][j] == g[i][k] + g[k][j] ){ //判断s和t之前的最短路径是否经过k
                    ans += (double)( num[i][k] * num[k][j] ) / num[i][j];
                }
            }
        }
    }
    cout << fixed << setprecision(3) << ans << endl;
}
}

```

### 3、单源有权最短路径问题——dijkstra

一个顶点到其余各顶点的最短路径算法，解决的是有权图中最短路径问题。迪杰斯特拉算法主要特点是从起始点开始，采用贪心算法的策略，每次遍历到始点距离最近且未访问过的顶点的邻接节点，直到扩展到终点为止。



输入：  
n m  
u v w  
.....

输出：  
数组，每个点到原点1的最短距离

模板 1: 邻接矩阵存图

```

18 void Dijkstra() {
19     _dis[1] = 0; //起点为1
20     for(int i = 1; i <= _n; i++) {
21         _minx = 2e9;
22         _k = 0; //中转点
23         for(int j = 1; j <= _n; j++) {
24             //蓝点中找到源点v的最小距离点
25             if(!_b[j] && _dis[j] < _minx) {
26                 _minx = _dis[j];
27                 _k = j;
28             }
29         }
30         //中转点k蓝变白
31         _b[_k] = true;
32         //通过中转点更新其余蓝点
33         for(int j = 1; j <= _n; j++) {
34             if(_dis[_k] + _w[_k][j] < _dis[j]) {
35                 _dis[j] = _dis[_k] + _w[_k][j];
36                 // _pre[j] = _k; //记录到j点的中转点
37             }
38         }
39     }
40 }

41 int main () {
42     cin >> _n >> _m; //顶点数和边数
43     memset(_w, 0x3f, sizeof(_w));
44     memset(_dis, 0x3f, sizeof(_dis));
45     int v1 = 0, v2 = 0, d = 0;
46     for(int i = 1; i <= _m; i++) {
47         cin >> v1 >> v2 >> d;
48         //防止重边
49         _w[v1][v2] = min(d, _w[v1][v2]);
50         _w[v2][v1] = min(d, _w[v2][v1]);
51     }
52     Dijkstra();
53     // for(int i = 1; i <= _n; i++) {
54     //     cout << _pre[i] << endl;
55     // }
56     // cout << "1到5的距离为: " << _dis[5] << endl;
57     // Print(5);
58
59     for(int i = 1; i <= _n; i++) {
60         cout << _dis[i] << " ";
61     }
62     return 0;
63 }

```



## 模板 2: 邻接表+优先队列 优化

```
1  #include <iostream>
2  #include <queue>
3  #include <cstring>
4  using namespace std;
5  const int N = 100010;
6  const int M = 200010;
7  priority_queue< pair<int, int> > pq;
8  int head[N];
9  int numEdge;
10 int dis[N];
11 bool b[N];
12 int n, m;
13 struct edge{
14     int next;
15     int to;
16     int w;
17 }e[2 * M];
18
19 void AddEdge(int from, int to, int w){
20     numEdge++;
21     e[numEdge].next = head[from];
22     e[numEdge].to = to;
23     e[numEdge].w = w;
24     head[from] = numEdge;
25 }
26
27 void Dijkstra(int sta) {
28     memset(dis, 0x3f, sizeof(dis));
29     dis[sta] = 0;
30     pq.push(make_pair(0, sta));
31     while(!pq.empty()){
32         int x = pq.top().second; // 取最小值的编号
33         pq.pop();
34         if(!b[x]){
35             b[x] = true;
36             for(int i = head[x]; i != 0; i = e[i].next){
37                 int y = e[i].to;
38                 int z = e[i].w;
39                 if(dis[x] + z < dis[y]){
40                     dis[y] = dis[x] + z;
41                     pq.push(make_pair(-1 * dis[y], y)); // 小跟堆
42                 }
43             }
44         }
45     }
46     for(int i = 1; i <= n; i++){
47         cout << dis[i] << " ";
48     }
49 }
50
51 int main(){
52     int u, v, w;
53     cin >> n >> m;
54     for(int i = 1; i <= m; i++){
55         cin >> u >> v >> w;
56         AddEdge(u, v, w);
57         AddEdge(v, u, w);
58     }
59     int start = 1;
60     Dijkstra(start);
61     return 0;
62 }
```

练习：P1529

第一问：为什么要用 dijkstra？

- 1、有权值，没负值
- 2、单源最短路问题
- 3、边的数量比较大，邻接表存图

第二问：怎么解决点的编号为字母的问题？

通过字母减去 'A' 实现，将字母转化为数字

大写字母的点的编号在 0-26 之间

小写字母的点的编号大于 26

代码实现：

- (1) 第一步：存图的时候，将字母转化为数字  
起点为大写的字母 Z

```
int main(){
    char u, v;
    int w, mini = 2e9, num = 0;
    cin >> n;
    for(int i = 1; i <= n; i++){
        cin >> u >> v >> w;
        AddEdge(u - 'A', v - 'A', w);
        AddEdge(v - 'A', u - 'A', w);
    }
    int start = 'Z' - 'A';
    Dijkstra(start);
}
```

- (2) 第二步：dijkstra 求单源最短路径

```
void Dijkstra(int sta) {
    memset(dis, 0x3f, sizeof(dis));
    dis[sta] = 0;
    pqe.push(make_pair(0, sta));
    while(!pqe.empty()){
        int x = pqe.top().second; // 取最小值的编号
        pqe.pop();
        if(!b[x]){
            b[x] = true;
            for(int i = head[x]; i != 0; i = e[i].next){
                int y = e[i].to;
                int z = e[i].w;
                if(dis[x] + z < dis[y]){
                    dis[y] = dis[x] + z;
                    pqe.push(make_pair(-1 * dis[y], y)); // 小跟堆
                }
            }
        }
    }
}
```

(3) 第三步：求大写字母的点到原点 Z 的最短距离，并输出

```
for(int i = 'A' - 'A'; i <= 'Y' - 'A'; i++){
    if(dis[i] < mini){
        mini = dis[i];
        num = i;
    }
}
cout << char(num + 'A') << " " << mini;
return 0;
```

完整代码

```
1  #include <iostream>
2  #include <queue>
3  #include <cstring>
4  using namespace std;
5  const int N = 200;
6  const int M = 10010;
7  int head[N], numEdge;
8  int m;
9  int dis[N];
10 bool b[N];
11 priority_queue< pair<int, int> >pque; //默认大根堆
12 struct edge{
13     int next;
14     int to;
15     int w;
16 }e[2 * M];
17
18 void AddEdge(int from, int to, int w){
19     numEdge++;
20     e[numEdge].next = head[from];
21     e[numEdge].to = to;
22     e[numEdge].w = w;
23     head[from] = numEdge;
24 }
25
26 void Dijkstra(int sta) {
27     memset(dis, 0x3f, sizeof(dis));
28     dis[sta] = 0;
29     pque.push(make_pair(0, sta));
30     while(!pque.empty()){
31         int x = pque.top().second; //取最小值的编号
32         pque.pop();
33         if(!b[x]){
34             b[x] = true;
35             for(int i = head[x]; i != 0; i = e[i].next){
36                 int y = e[i].to;
37                 int z = e[i].w;
38                 if(dis[x] + z < dis[y]){
39                     dis[y] = dis[x] + z;
40                     pque.push(make_pair(-1 * dis[y], y)); //小跟堆
41                 }
42             }
43         }
44     }
45 }
```

```
1
2
3 int main(){
4     char u, v;
5     int w, mini = 2e9, num = 0;
6     cin >> m;
7     for(int i = 1; i <= m; i++){
8         cin >> u >> v >> w;
9         AddEdge(u - 'A', v - 'A', w);
10        AddEdge(v - 'A', u - 'A', w);
11    }
12    int start = 'Z' - 'A';
13    Dijkstra(start);
14
15    for(int i = 'A' - 'A'; i <= 'Y' - 'A'; i++){
16        if(dis[i] < mini){
17            mini = dis[i];
18            num = i;
19        }
20    }
21    cout << char(num + 'A') << " " << mini;
22    return 0;
23 }
```

#### 4、并查集作用： 动态维护和处理元素之间的关系

- 找到一个元素的所属集合
- 将两个元素各自所属的集合进行合并（当给出两元素的无序对(a,b)时，能够快速合并 a 和 b 所在的集合）
- 反复查找某个元素在哪个集合中

练习： P1551 模板题

代码实现：

第一步：新建一个并查集（数组实现，初始祖先是自己）

```
int main(){
    int x, y;
    cin >> n >> m >> q;
    for(int i = 1; i <= n; i++){
        fa[i] = i;
    }
}
```

第二步：通过 find 函数更新祖先（即合并亲戚网的过程）

```
for(int i = 1; i <= m; i++){
    cin >> x >> y;
    x = Find(x);
    y = Find(y);
    fa[y] = x;
}

int Find(int x){//实现了找祖先的过程
    if(x == fa[x]){//说明自己就是祖先
        return x;
    }
    else {
        return Find(fa[x]);
    }
}
```

第三步：判断是不是亲戚，即判断祖先是不是同一个即可

```
for(int i = 1; i <= q; i++){
    cin >> x >> y;
    x = Find(x);
    y = Find(y);
    if(x == y){
        cout << "yes" << endl;
    }
    else {
        cout << "no" << endl;
    }
}
```



完整代码

```
1 #include <iostream>
2 using namespace std;
3 int fa[110];
4 int n, m, q;
5 void print(){
6     for(int i = 1; i <= n; i++){
7         cout << fa[i] << " ";
8     }
9     cout << endl;
10 }
11 int Find(int x){//实现了找祖先的过程
12     if(x == fa[x]){//说明自己就是祖先
13         return x;
14     }
15     else {
16         return Find(fa[x]);
17     }
18 }
19
20 int Find1(int x){//实现了找祖先的过程
21     if(x != fa[x]){
22         fa[x] = Find(fa[x]);
23     }
24     return fa[x];
25 }
26
27 int main(){
28     int x, y;
29     cin >> n >> m >> q;
30     for(int i = 1; i <= n; i++){
31         fa[i] = i;
32     }
33
34     for(int i = 1; i <= m; i++){
35         cin >> x >> y;
36         x = Find(x);
37         y = Find(y);
38         fa[y] = x;
39     }
40
41     for(int i = 1; i <= q; i++){
42         cin >> x >> y;
43         x = Find(x);
44         y = Find(y);
45         if(x == y){
46             cout << "yes" << endl;
47         }
48         else {
49             cout << "no" << endl;
50         }
51     }
52
53     return 0;
54 }
```

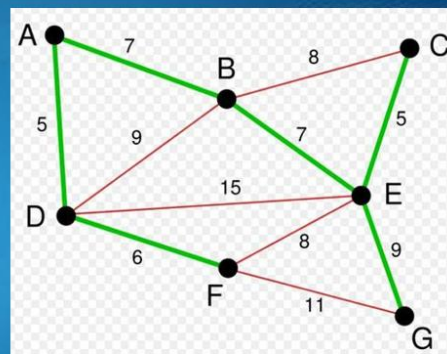
## 5、最小生成树

假设：我们要在n个城市中建立一个通信网络，则连通这n个城市需要布置n-1条通信线路，这个时候我们需要考虑如何在成本最低的情况下建立这个通信网？

数据结构建立：

- n个城市就是图上的n个顶点，
  - 边表示两个城市的通信线路
  - 每条边上的权重就是我们搭建这条线路所需要的成本
- n个顶点的连通网可以建立不同的生成树，每一颗生成树都可以作为一个通信网，当我们构造这个连通网所花的成本最小时，搭建该连通网的生成树，就称为**最小生成树**。

定义：所有边的权值之和最小的生成树



最小生成树的算法

kruskal (克鲁斯卡尔算法)

思想：将连通网图中的所有边按照权值大小进行升序排序，从小到大依次选择。

方法:

- 建立并查集，每个点各自构成一个集合
- 边升序排列，依次扫边 (x,y,z)
- 若 x,y 属于同一集合，则忽略此边，继续扫
- 否则，合并 x,y，累加 z 到 MST 中
- 所有边扫完，结束算法

练习: P3366 (模板题)

代码实现:

第一步: 存边，并排序

```
int main(){
    cin >> n >> m;
    for(int i = 1; i <= m; i++){
        cin >> e[i].x >> e[i].y >> e[i].z;
    }
    sort(e + 1, e + m + 1, Cmp);
}
```

第二步: 新建一个并查集 (数组实现，初始祖先是自己)

```
for(int i = 1; i <= n; i++){
    fa[i] = i;
}
```

第三步: 从小到大选边，x 和 y 已经合并，则不选此边并且合并 x 和 y，  
如果 x 和 y 没合并，选此边，并累加权值。

```
for(int i = 1; i <= m; i++){
    int x = Find(e[i].x);
    int y = Find(e[i].y);
    if( x != y ){
        fa[y] = x;
        ans += e[i].z;
    }
}
cout << ans;
```

完整代码

```

20 #include <iostream>
21 #include <algorithm>
22 using namespace std;
23 struct edge{
24     int x;
25     int y;
26     int z;
27 } e[500010];
28 int fa[100010];
29 int n, m, ans;
30 int Find(int x){
31     if(fa[x] != x){
32         fa[x] = Find(fa[x]);
33     }
34     return fa[x];
35 }
36 bool Cmp(edge a, edge b){
37     return a.z < b.z;
38 }

```

```

1 int main(){
2     cin >> n >> m;
3     for(int i = 1; i <= m; i++){
4         cin >> e[i].x >> e[i].y >> e[i].z;
5     }
6     sort(e + 1, e + m + 1, Cmp);
7     for(int i = 1; i <= n; i++){
8         fa[i] = i;
9     }
10    for(int i = 1; i <= m; i++){
11        int x = Find(e[i].x);
12        int y = Find(e[i].y);
13        if(x != y){
14            fa[y] = x;
15            ans += e[i].z;
16        }
17    }
18    cout << ans;
19 }

```

练习：P4047

- 1、n 个点的所有边中连 n-k 条边可以生成 k 棵最小生成树
- 2、输出为 num==n-k 时下一次的 e[i].z

代码实现：

第一步：输入坐标，为每两个点之间建边

```

double measure(int a,int b)//求欧几里得距离
{
    return sqrt(pow((ass[a].x-ass[b].x),2)+pow((ass[a].y-ass[b].y),2));
}

int main()
{
    cin >> n >> k;
    for(int i=1;i<=n;i++){
        cin >> ass[i].x >> ass[i].y;
    }
    o = 0;//边数;
    for(int i=1;i<=n;i++){
        for(int j=1;j<i;j++)
        {
            if(i!=j){
                o++; //边计数
                e[o].x=i;
                e[o].y=j;
                e[o].l=measure(i,j); //为所有的点之间建边;
            }
        }
    }
}

```

第二步：建立并查集、按边从小到大的顺序排序

```
for(int i=1;i<=n;i++){
    fa[i] = i;
}
sort(e+1,e+1+o,cmp);
kruskal();
```

第三步：从小到大开始连边，并统计连的边的数量是否达到  $n-k$ ，如果达到，说明已经生成  $k$  个最小生成树，此时输出下一条边的长度，即为部落之间的最近距离的最大值。

```
int find(int a){
    if(fa[a]!=a)
        fa[a] = find(fa[a]);
    return fa[a];
}

void unionn(int a,int b){
    fa[find(b)] = find(a);
}

void kruskal(){
    for(int i=1;i<=o;i++)
    {
        if(num==n-k) flag = 1;
        if(find(e[i].x)!=find(e[i].y))
        {
            num++; //统计连边的数量
            unionn(e[i].x,e[i].y); //合并两个居住点为一个部落
            if(flag){
                cout << fixed << setprecision(2) << e[i].l;
                return ;
            }
        }
    }
}
```

(完整代码在下页)



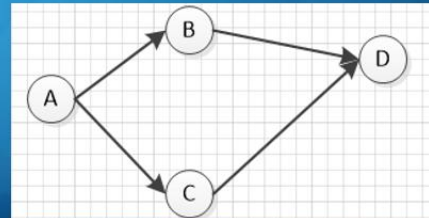
```

6 struct s{
7     int x,y;
8     double l;
9 }ass[MA],e[MA]; //结构体存点;
10
11 double cmp(s x,s y){
12     return x.l < y.l;
13 }
14 int find(int a){
15     if(fa[a]!=a)
16         fa[a] = find(fa[a]);
17     return fa[a];
18 }
19 void unionn(int a,int b){
20     fa[find(b)] = find(a);
21 }
22 void kruskal(){
23     for(int i=1;i<=o;i++){
24         {
25             if(num==n-k) flag = 1;
26             if(find(e[i].x)!=find(e[i].y))
27             {
28                 num++; //统计连边的数量
29                 unionn(e[i].x,e[i].y); //合并两个居住点为一个部落
30                 if(flag){
31                     cout << fixed << setprecision(2) << e[i].l;
32                     return ;
33                 }
34             }
35         }
36     }
37     double measure(int a,int b) //求欧几里得距离
38     {
39         return sqrt(pow((ass[a].x-ass[b].x),2)+pow((ass[a].y-ass[b].y),2));
40     }
41     int main()
42     {
43         cin >> n >> k;
44         for(int i=1;i<=n;i++){
45             cin >> ass[i].x >> ass[i].y;
46         }
47         o = 0; //边数;
48         for(int i=1;i<=n;i++){
49             for(int j=1;j<i;j++){
50                 {
51                     if(i!=j){
52                         o++; //边计数
53                         e[o].x=i;
54                         e[o].y=j;
55                         e[o].l=measure(i,j); //为所有的点之间建边;
56                     }
57                 }
58             }
59             for(int i=1;i<=n;i++){
60                 fa[i] = i;
61             }
62             sort(e+1,e+1+o,cmp);
63             kruskal();

```

## 6、拓扑排序

- 一项大的工程可以看作是由若干个子工程组成的集合，这些子工程之间必定存在一定的先后顺序，即某些子工程必须在其他的一些子工程完成后才能开始。
- 在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系。这样的有向图称为顶点表示活动的网，我们称为**AOV网** (Activity On Vertex Network)
- 拓扑排序：把AOV网中所有活动排成一个序列
- 拓扑排序可以帮助我们**合理安排工程进度**，由AOV网构造拓扑序列具有很高的应用价值



算法步骤：

- 选择一个入度为 0 的顶点并输出
- 从 AOV 网中删除此顶点和以此顶点为起点的所有关联边
- 重复上述两步，直到不存在入度为 0 的顶点为止
- 若输出的顶点数小于 AOV 网中的顶点数，则说明图中有回路，否则输出序列为一种拓扑排序

模板：

```
1 #include <iostream>
2 #include <queue>
3 using namespace std;
4 int numEdge, n, m;
5 int head[110];
6 int rdu[110];
7 queue<int> que;
8 struct edge{
9     int next;
10    int to;
11 } e[10010];
12 void AddEdge(int from, int to){
13     numEdge++;
14     e[numEdge].next = head[from];
15     e[numEdge].to = to;
16     head[from] = numEdge;
17 }
18
19 void TuoPu(){
20     for(int i = 1; i <= n; i++){
21         if(rdu[i] == 0){
22             que.push(i);
23         }
24     }
25     while(!que.empty()){
26         int x = que.front();
27         cout << x << " ";
28         que.pop();
29         for(int i = head[x]; i != 0; i = e[i].next){
30             int y = e[i].to;
31             rdu[y]--;
32             if(rdu[y] == 0){
33                 que.push(y);
34             }
35         }
36     }
37 }
38
39 int main(){
40     cin >> n >> m;
41     int u = 0, v = 0;
42     for(int i = 1; i <= m; i++){
43         cin >> u >> v;
44         rdu[v]++;
45         AddEdge(u, v);
46     }
47     TuoPu();
48 }
```

练习：P3243

字典序：

正向：

13452

15234

反向：

25431

43251

是否可以考虑反向建图，使用大根堆，每次优先处理最大的入度为0的点（最后再处理最小的入度为0的点），然后逆序输出答案？

处理后：3 4 2 5 1

逆序：1 5 2 4 3

代码实现：

第一步：每次询问，所有数据清零

```
int main () {
    int u = 0, v = 0;
    cin >> _T;
    for(int t = 1; t <= _T; t++) {

        //初始化
        _numEdge = 0;
        memset(_head, 0, sizeof(_head));
        memset(_e, 0, sizeof(_e));
        memset(_ruDu, 0, sizeof(_ruDu));
        vector<int> _myList;
        priority_queue<int> _que; //大根堆
        _ans = 0;
```

第二步：反向存图

```
    cin >> _n >> _m;
    for(int i = 1; i <= _m; i++) {
        cin >> u >> v;
        AddEdge(v, u); //反向存图
        _ruDu[u]++;
    }
```

第三步：拓扑排序，并且把最大编号的点存到动态数组里

```

for(int i = 1; i <= _n; i++) {
    if(_ruDu[i] == 0) {
        _que.push(i); //入度为0, 入队
    }
}
int curP = 0;
while(!_que.empty()) {
    curP = _que.top();
    _myList.push_back(curP); //保证最大的编号先存
    _que.pop();
    for(int i = _head[curP]; i != 0; i = _e[i].next) {
        int y = _e[i].to;
        _ruDu[y]--;
        if(_ruDu[y] == 0) {
            _que.push(y);
        }
    }
}
}

```

第四步：逆序输出

```

if(_myList.size() != _n) {
    cout << "Impossible!" << endl;
}
else {
    for(int i = _myList.size() - 1; i >= 0; i--) {
        cout << _myList[i] << " ";
    }
    cout << endl;
}
}

```

完整代码

```

5 using namespace std;
6 struct edge{
7     int next;
8     int to;
9     int w;
10 }_e[100010];
11 int _ruDu[100010];
12 int _ans = 0;
13 int _n, _m;
14 int _numEdge = 0;
15 int _head[100010];
16 int _T;
17 void AddEdge(int from, int to) {
18     _numEdge++;
19     _e[_numEdge].next = _head[from];
20     _e[_numEdge].to = to;
21     _head[from] = _numEdge;
22 }

```



```

24 int main () {
25     int u = 0, v = 0;
26     cin >> _T;
27     for(int t = 1; t <= _T; t++) {
28         //初始化
29         _numEdge = 0;
30         memset(_head, 0, sizeof(_head));
31         memset(_e, 0, sizeof(_e));
32         memset(_ruDu, 0, sizeof(_ruDu));
33         vector<int> _myList;
34         priority_queue<int> _que; //大根堆
35         _ans = 0;
36
37         cin >> _n >> _m;
38         for(int i = 1; i <= _m; i++) {
39             cin >> u >> v;
40             AddEdge(v, u); //反向存图
41             _ruDu[u]++;
42         }
43         for(int i = 1; i <= _n; i++) {
44             if(_ruDu[i] == 0) {
45                 _que.push(i); //入度为0, 入队
46             }
47         }
48         int curP = 0;
49         while(!_que.empty()) {
50             curP = _que.top();
51             _myList.push_back(curP); //保证最大的编号先存
52             _que.pop();
53             for(int i = _head[curP]; i != 0; i = _e[i].next) {
54                 int y = _e[i].to;
55                 _ruDu[y]--;
56                 if(_ruDu[y] == 0) {
57                     _que.push(y);
58                 }
59             }
60         }
61         if(_myList.size() != _n) {
62             cout << "Impossible!" << endl;
63         }
64         else {
65             for(int i = _myList.size() - 1; i >= 0; i--) {
66                 cout << _myList[i] << " ";
67             }
68             cout << endl;
69         }
70     }
71 }

```