

广度优先搜索BFS

迷宫问题

手写队列

```
1  #include<iostream>
2  #include<algorithm>
3  #include<cstring>
4  using namespace std;
5  const int N = 110;
6  typedef pair<int, int> PII;
7
8  int map[N][N], dis[N][N];
9  int m, n;
10 int dx[4] = {-1, 0, 1, 0};
11 int dy[4] = {0, 1, 0, -1};
12 PII que[N * N];
13 int bfs()
14 {
15     int hh = 0, tt = 0;
16     que[0] = {0, 0};
17     memset(dis, -1, sizeof(dis));
18     dis[0][0] = 0;
19     while(hh <= tt)
20     {
21         PII t = que[hh++];
22         for(int i = 0 ; i < 4; i++)
23         {
24             int x = t.first + dx[i], y = t.second + dy[i];
25             if(x >= 0 && x < m && y >= 0 && y < n && map[x][y] == 0 &&
dis[x][y] == -1)
26             {
27                 dis[x][y] = dis[t.first][t.second] + 1;
28                 que[++tt] = {x, y};
29             }
30         }
31     }
32     return dis[m-1][n-1];
33 }
34
35 int main(void)
36 {
37     cin >> m >> n;
38     for(int i = 0 ; i < m ; i++)
39         for(int j = 0; j < n; j++)
40             cin >> map[i][j];
41     cout << bfs() << endl;
42     return 0;
43 }
44
```

调用queue

```

1  #include<iostream>
2  #include<cstring>
3  #include<queue>
4  using namespace std;
5  const int N = 110;
6  typedef pair<int, int> PII;
7  int map[N][N], dis[N][N];
8  int m, n;
9  int bfs()
10 {
11     queue<PII> q;
12     int dx[4] = {-1, 0, 1, 0};
13     int dy[4] = {0, 1, 0, -1};
14     q.push({0, 0});
15     memset(dis, -1, sizeof(dis));
16     dis[0][0] = 0;
17     while(!q.empty())
18     {
19         PII temp = q.front();
20         q.pop();
21         for(int i = 0; i < 4; i++)
22         {
23             int x = temp.first + dx[i], y = temp.second + dy[i];
24             if(x >= 0 && x < m && y >= 0 && y < n && map[x][y] == 0 &&
25             dis[x][y] == -1)
26             {
27                 dis[x][y] = dis[temp.first][temp.second] + 1;
28                 q.push({x, y});
29             }
30         }
31     }
32     return dis[m-1][n-1];
33 }
34 int main(void)
35 {
36     cin >> m >> n;
37     for(int i = 0; i < m ; i++)
38         for(int j = 0; j < n; j++)
39             cin >> map[i][j];
40     cout << bfs() << endl;
41     return 0;
42 }

```

八数码

```

1  #include<iostream>
2  #include<queue>
3  #include<algorithm>
4  #include<string>
5  #include<unordered_map>
6  using namespace std;
7  // 每个string 对应着一个状态，每个状态相当于图中的一个结点，状态的转移是通过找到x对应着在字符串中的下标，然后转化为3*3的矩阵后交换x和上下左右中的一个元素，得到一个新的字符串
8  // 初始状态通过输入获得，结束状态就是"12345678x"
9  // 对于bfs需要一个队列，存放状态，因此是一个queue<string>

```

```

10 // 对于距离distance的存储，用到了unordered_map这个hash表，将不同状态到起始状态的距离存
    入hash表，然后在需要的时候提取
11 // 转移依然是上下左右四种可能，因此定义一个dx和一个dy
12 int bfs(string start)
13 {
14     string end = "12345678x";
15     queue<string> q;
16     unordered_map<string, int> dis;
17     int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};
18     q.push(start);
19     dis[start] = 0;
20     // 队列不为空时进入循环
21     while(!q.empty())
22     {
23         // 每次对头元素出队，作为当前状态开始bfs
24         auto temp = q.front();
25         q.pop();
26         // 如果这时队头元素恰好为结束状态，bfs结束，从hash表中提取关键字值即可
27         if(temp == end)
28         {
29             return dis[temp];
30         }
31         // 找到x位于当前状态string中的下标，然后转化为3*3格局中的x和y
32         int distance = dis[temp];
33         int k = temp.find('x');
34         int x = k / 3, y = k % 3;
35         // 状态转移
36         for(int i = 0; i < 4; i++)
37         {
38             int a = x + dx[i], b = y + dy[i];
39             if(a >= 0 && a < 3 && b >= 0 && b < 3)
40             {
41                 // 建立新状态入队
42                 swap(temp[a * 3 + b], temp[k]);
43                 // dis.count(temp)说明temp还没存到dis中
44                 if(!dis.count(temp))
45                 {
46                     dis[temp] = distance + 1;
47                     q.push(temp);
48                 }
49                 swap(temp[a * 3 + b], temp[k]);
50             }
51         }
52     }
53     return -1;
54 }
55
56 int main(void)
57 {
58     string start;
59     for(int i = 1; i <= 9; i++)
60     {
61         char c;
62         cin >> c;
63         start += c;
64     }

```

```

65     cout << bfs(start) << endl;
66     return 0;
67 }

```

树和图的存储结构

单链表

```

1  int head;    //头结点指针
2  int e[N];    // 所有结点值的数组
3  int ne[N];   // 所有结点的next域
4  int idx;     // 全局计数器，idx表示当前用到的哪个结点
5  // 初始化
6  void init()
7  {
8      head = -1;
9      idx = 0;
10 }
11 // 将值a插入链表头
12 void insert(int a)
13 {
14     e[idx] = a;
15     ne[idx] = head;
16     head = idx++;
17 }
18 void remove()
19 {
20     head = ne[head];
21 }

```

邻接表

```

1  int h[N];    // 头结点数组
2  int e[N];    // 所有结点值的集合
3  int ne[N];   // next指针域
4  int idx;     // 表示当前用到的结点
5  // 添加一条边
6  void add(int a, int b)
7  {
8      e[idx] = b;
9      ne[idx] = h[a];
10     h[a] = idx++;
11 }
12 // 初始化
13 idx = 0;
14 memest(h, -1, sizeof(h));

```

树与图的遍历

时间复杂度 $O(n + m)$, n 表示点数, m 表示边数

(1) 深度优先遍历

```
1 void dfs(int u)
2 {
3     visit[u] = true;    //表示点u已经被遍历过
4     for(int i = h[u]; i != -1; i = ne[i])
5     {
6         j = e[i];
7         if(!visit[j]) dfs(j);
8     }
9 }
```

(2) 广度优先遍历

```
1 queue<int> q;
2 visit[1] = true;
3 q.push(1);
4 while(q.size())
5 {
6     int t = q.front();
7     q.pop();
8     for(int i = h[t]; i != -1; i = ne[i])
9     {
10         int j = e[i];
11         if(!visit[j])
12         {
13             visit[j] = true;
14             q.push(j);
15         }
16     }
17 }
```

树的重心

给定一颗树，树中包含 n 个结点（编号 $1 \sim n$ ）和 $n - 1$ 条无向边。

请你找到树的重心，并输出将重心删除后，剩余各个连通块中点数的最大值。

重心定义：重心是指树中的一个结点，如果将这个点删除后，剩余各个连通块中点数的最大值最小，那么这个节点被称为树的重心。

输入格式

第一行包含整数 n ，表示树的结点数。

接下来 $n - 1$ 行，每行包含两个整数 a 和 b ，表示点 a 和点 b 之间存在一条边。

输出格式

输出一个整数 m ，表示将重心删除后，剩余各个连通块中点数的最大值。

数据范围

$$1 \leq n \leq 10^5$$

```
1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  #include<cstdio>
5  using namespace std;
6  // 由于是无向图，要存储边的数量是n的两倍
7  const int N = 1e5 + 10, M = 2 * N;
8  // 分别代表头结点数组，边的集合，指针，当前可用结点
9  int h[N], e[M], ne[M], idx;
10 // 结果
11 int ans = N;
12 int n;
13 // 标记数组
14 bool visit[N];
15 // 在a和b之间建立一条边
16 void add(int a, int b)
17 {
18     e[idx] = b;
19     ne[idx] = h[a];
20     h[a] = idx++;
21 }
22 // dfs，访问结点u，因为只需要访问所有结点一遍，而不是多条路径，因此不需要恢复现场
23 int dfs(int u)
24 {
25     // 表明结点u已经被访问
26     visit[u] = true;
27     // size表示结点u的所有子树中点数最多的大小
28     // sum 表示其所有孩子数
29     int size = 0, sum = 0;
30     // 从u开始访问所有u的边，对每个边的终点dfs
31     for(int i = h[u]; i != -1; i = ne[i])
32     {
33         int j = e[i];
34         // 由于无向图，如果已经访问到了，就不再访问
35         if(visit[j]) continue;
36         int s = dfs(j);
37         // 更新size和sum
38         size = max(s, size);
39         sum += s;
40     }
41     // size表示删除u后，剩下的几部分中点数最大的
42     size = max(size, n - sum - 1);
43     // 如果当前size比ans小，跟新ans
44     ans = min(ans, size);
```

```

45     return sum + 1;
46 }
47 int main(void)
48 {
49     cin >> n;
50     memset(h, -1, sizeof(h));
51     // 建立图(邻接表形式)
52     for(int i = 0; i < n-1 ;i ++)
53     {
54         int a, b;
55         cin >> a >> b;
56         add(a, b);
57         add(b, a);
58     }
59     dfs(1);
60     cout << ans << endl;
61     return 0;
62 }

```

图中点的层次

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环。

所有边的长度都是 1，点的编号为 $1 \sim n$ 。

请你求出 1 号点到 n 号点的最短距离，如果从 1 号点无法走到 n 号点，输出 -1 。

输入格式

第一行包含两个整数 n 和 m 。

接下来 m 行，每行包含两个整数 a 和 b ，表示存在一条从 a 走到 b 的长度为 1 的边。

输出格式

输出一个整数，表示 1 号点到 n 号点的最短距离。

数据范围

$$1 \leq n, m \leq 10^5$$

```

1  #include<iostream>
2  #include<algorithm>
3  #include<cstring>
4  #include<queue>
5  using namespace std;
6  const int N = 1e5 + 10, M = 1e5 + 10;

```

```

7  int h[N], e[M], ne[M], idx;
8  int n, m;
9  bool visit[N];
10 int dis[N];
11 void add(int a, int b)
12 {
13     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
14 }
15 void bfs(int u)
16 {
17     queue<int> q;
18     visit[u] = true;
19     q.push(u);
20     dis[u] = 0;
21     while(q.size())
22     {
23         int temp = q.front();
24         q.pop();
25         for(int i = h[temp]; i != -1; i = ne[i])
26         {
27             int j = e[i];
28             if(!visit[j])
29             {
30                 visit[j] = true;
31                 q.push(j);
32                 dis[j] = dis[temp] + 1;
33             }
34         }
35     }
36 }
37 }
38 int main(void)
39 {
40     cin >> n >> m;
41     memset(h, -1, sizeof(h));
42     memset(dis, -1, sizeof(dis));
43     for(int i = 0 ; i < m; i++)
44     {
45         int a, b;
46         cin >> a >> b;
47         add(a, b);
48     }
49     bfs(1);
50     cout << dis[n] << endl;
51     return 0;
52 }

```

有向图的拓扑序列

手动模拟队列

```

1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  using namespace std;

```



```

5  const int N = 1e5 + 10;
6  int h[N], e[N], ne[N], idx;
7  int q[N], d[N];
8  int n, m;
9  void add(int a, int b)
10 {
11     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
12 }
13 bool toposort()
14 {
15     int hh = 0, tt = -1;
16     for(int i = 1; i <= n; i++)
17     {
18         if(!d[i])
19             q[++tt] = i;
20     }
21     while(hh <= tt)
22     {
23         int t = q[hh++];
24         for(int i = h[t]; i != -1; i = ne[i])
25         {
26             int j = e[i];
27             d[j]--;
28             if(d[j] == 0) q[++tt] = j;
29         }
30     }
31     return tt == n-1;
32 }
33 int main(void)
34 {
35     cin >> n >> m;
36     memset(h, -1, sizeof(h));
37     for(int i = 0; i < m; i++)
38     {
39         int a, b;
40         cin >> a >> b;
41         add(a, b);
42         d[b]++;
43     }
44     if(toposort())
45     {
46         for(int i = 0; i < n; i++)
47             cout << q[i] << " ";
48         cout << endl;
49     }
50     else
51         puts("-1");
52     return 0;
53 }

```

stl中的queue实现

```

1  #include<iostream>
2  #include<algorithm>
3  #include<queue>

```

```

4  #include<cstring>
5  using namespace std;
6  const int N = 1e5 + 10;
7  int h[N], e[N], ne[N], idx;
8  int n, m;
9  int degree[N];
10 int q[N], cnt;
11 queue<int> que;
12 void add(int a, int b)
13 {
14     e[idx] = b, ne[idx] = h[a], h[a] = idx++;
15 }
16 bool toposort()
17 {
18     for(int i = 1; i <= n ; i++)
19     {
20         if(!degree[i])
21         {
22             que.push(i);
23             q[cnt++] = i;
24         }
25     }
26     while(que.size())
27     {
28         int temp = que.front();
29         que.pop();
30         for(int i = h[temp]; i != -1; i = ne[i])
31         {
32             int j = e[i];
33             degree[j]--;
34             if(!degree[j])
35             {
36                 que.push(j);
37                 q[cnt++] = j;
38             }
39         }
40     }
41     return cnt == n;
42 }
43 int main(void)
44 {
45     cin >> n >> m;
46     memset(h, -1, sizeof(h));
47     for(int i = 0; i < m; i++)
48     {
49         int a, b;
50         cin >> a >> b;
51         add(a, b);
52         degree[b]++;
53     }
54     if(toposort())
55     {
56         for(int i = 0 ; i < n; i++)
57             cout << q[i] << " ";
58         cout << endl;
59     }

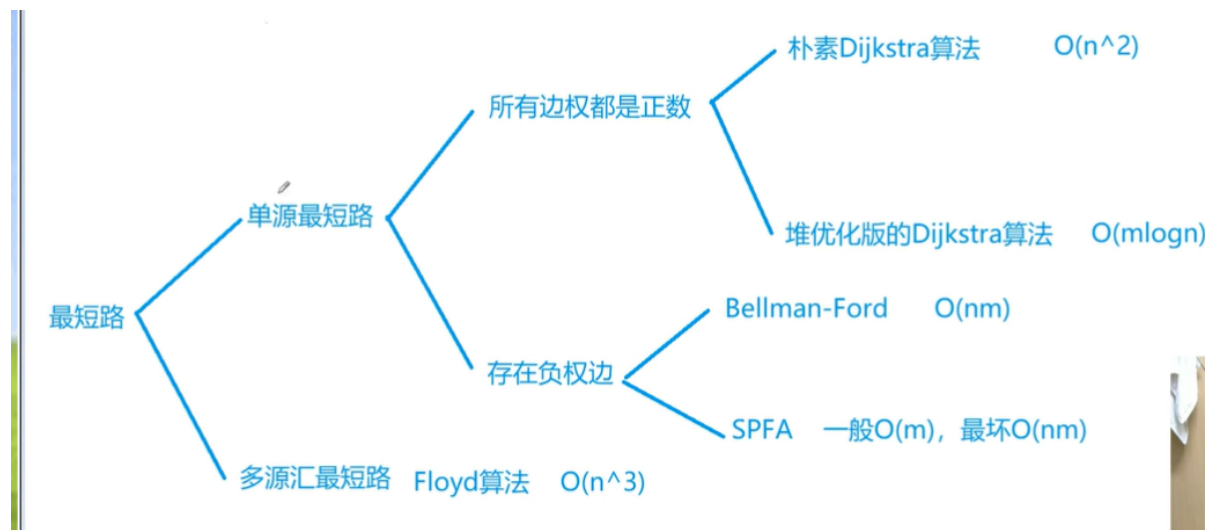
```

```

60     else
61         puts("-1");
62     return 0;
63 }

```

最短路算法



朴素Dijkstra算法

稠密图，用邻接矩阵存储

```

1  #include<iostream>
2  #include<algorithm>
3  #include<cstring>
4  using namespace std;
5  const int N = 510;
6  const int M = 1e5 + 10;
7  int n, m;
8  int g[N][N];
9  int dist[N];
10 bool visit[N];
11 int dijkstra()
12 {
13     memset(dist, 0x3f, sizeof dist);
14     dist[1] = 0;
15     for(int i = 0; i < n - 1; i++)
16     {
17         int t = -1;
18         // 找到一个distance最小的点作为新的起点
19         for (int j = 1; j <= n; j++)
20             if(!visit[j] && (t == -1 || dist[t] > dist[j]))
21                 t = j;
22         // 标记为已经访问
23         visit[t] = true;
24         // 更新从这个新起点到所有其他定点的最短距离
25         for(int j = 1; j <= n; j++)
26             dist[j] = min(dist[j], dist[t] + g[t][j]);
27     }
28     // 没有找到一条最短路
29     if(dist[n] == 0x3f3f3f3f) return -1;

```

```

30     return dist[n];
31 }
32 int main(void)
33 {
34     cin >> n >> m;
35     memset(g, 0x3f, sizeof g);
36     while(m--)
37     {
38         int a, b, c;
39         cin >> a >> b >> c;
40         g[a][b] = min(g[a][b], c);
41     }
42     int t = dijkstra();
43     cout << t << endl;
44     return 0;
45 }

```

堆优化的dijkstra算法

```

1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  #include<queue>
5  using namespace std;
6  typedef pair<int, int> PII;
7  const int N = 2e5 + 10;
8  int h[N], e[N], ne[N], w[N], idx;
9  bool visit[N];
10 int n, m;
11 int dist[N];
12 void add(int a, int b, int c)
13 {
14     e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
15 }
16 int dijkstra()
17 {
18     memset(dist, 0x3f, sizeof dist);
19     dist[1] = 0;
20     priority_queue<PII, vector<PII>, greater<PII>> heap;
21     heap.push({0, 1});
22     while(heap.size())
23     {
24         auto t = heap.top();
25         heap.pop();
26         int vertice = t.second, distance = t.first;
27         if(visit[vertice]) continue;
28         visit[vertice] = true;
29         for(int i = h[vertice]; i != - 1; i = ne[i])
30         {
31             int j = e[i];
32             if(dist[j] > distance + w[i])
33             {
34                 dist[j] = distance + w[i];
35                 heap.push({dist[j], j});
36             }

```

```

37     }
38 }
39 if(dist[n] == 0x3f3f3f3f) return -1;
40 return dist[n];
41 }
42 int main(void)
43 {
44     cin >> n >> m;
45     memset(h, -1, sizeof h);
46     while(m--)
47     {
48         int a, b, c;
49         cin >> a >> b >> c;
50         add(a, b, c);
51     }
52     cout << dijkstra() << endl;
53     return 0;
54 }
55

```

对于dijkstra算法

```

1  int dijkstra()
2  {
3      // 初始化 dist数组, 和小根堆(堆优化版本)
4      // dist[1] = 0, 将起始点的distance标为0
5      // 首元素进堆
6      while(heap 非空)
7      {
8          // 找到一个新的起始点 (离起始点距离最小的)
9          // 堆优化版本
10         取堆顶即为此时distance最小的点作为新的起始点
11         // 朴素版本
12         遍历n个点, 找到最小的j
13         查看这个点是否被访问过, 被访问过跳过, 否则标记为访问, 然后更新dist数组
14         // 堆优化版本注意松弛的时候进堆
15     }
16     如果dist == inf, 说明不存在路径, 返回-1
17     否则返回dist[n]即可
18 }

```

Bellman-ford算法

不能有负权回路, 有负权回路可能求不出最短路径

时间复杂度为 $O(nm)$

通用模版

```

1  int n, m;    // n表示点数, m表示边数
2  int dist[N]; // dist[x]存储1到x的最短路距离
3  struct edge //边, a表示出点, b表示入点, w表示边的权重
4  {
5      int a, b, w;
6  }edges[M];

```

```
7 int bellman_ford()
8 {
9     memset(dist, 0x3f, sizeof dist);
10    dist[1] = 0;
11    for(int i = 0; i < n; i++)
12    {
13        for(int j = 0; j < m; j++)
14        {
15            int a = edges[j].a, b = edges[j].b, w = edges[j].w;
16            if(dist[b] > dist[a] + w)
17                dist[b] = dist[a] + w;
18        }
19    }
20    if(dist[n] > 0x3f3f3f3f / 2) return -1;
21    return dist[n];
22 }
```

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

请你求出从 1 号点到 n 号点的最多经过 k 条边的最短距离，如果无法从 1 号点走到 n 号点，输出 `impossible`。

注意：图中可能 **存在负权回路**。

输入格式

第一行包含三个整数 n, m, k 。

接下来 m 行，每行包含三个整数 x, y, z ，表示存在一条从点 x 到点 y 的有向边，边长为 z 。

点的编号为 $1 \sim n$ 。

输出格式

输出一个整数，表示从 1 号点到 n 号点的最多经过 k 条边的最短距离。

如果不存在满足条件的路径，则输出 `impossible`。

数据范围

$$1 \leq n, k \leq 500,$$

$$1 \leq m \leq 10000,$$

$$1 \leq x, y \leq n,$$

任意边长的绝对值不超过 10000。

```
1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  using namespace std;
5  const int N = 510, M = 10010;
6  int n, m, k;
7  int dist[N], backup[N]; // backup存的是上一次的结果
8  struct edge
9  {
10     int a, b, w;
11 }edges[M];
12 int bellman_ford()
13 {
14     memset(dist, 0x3f, sizeof dist);
```

```

15     dist[1] = 0;
16     for(int i = 0; i < k; i++)
17     {
18         memcpy(backup, dist, sizeof(dist));
19         for(int j = 0; j < m; j++)
20         {
21             int a = edges[j].a, b = edges[j].b, w = edges[j].w;
22             dist[b] = min(dist[b], backup[a] + w);
23         }
24     }
25     if(dist[n] > 0x3f3f3f3f / 2) return -1;
26     return dist[n];
27 }
28
29 int main(void)
30 {
31     cin >> n >> m >> k;
32     for(int i = 0; i < m; i++)
33     {
34         int a, b, w;
35         cin >> a >> b >> w;
36         edges[i] = {a, b, w};
37     }
38     int t = bellman_ford();
39     if(t == -1)
40         puts("impossible");
41     else
42         cout << t << endl;
43     return 0;
44 }
45

```

SPFA

“关于SPFA，它死了”

队列优化的Bellman_ford算法

时间复杂度 平均情况下 $O(m)$, 最坏情况下 $O(nm)$, n 表示点数, m 表示边数

通用板子

SPFA求最短路

```

1  int n;
2  int h[N], w[N], e[N], ne[N], idx;
3  int dist[N];
4  bool visit[N];
5  int spfa()
6  {
7      memset(dist, 0x3f, sizeof dist);
8      dist[1] = 0;
9      queue<int> q;
10     q.push(1);
11     visit[1] = true;
12     while(q.size())

```



```

13     {
14         auto t = q.front();
15         q.pop();
16         visit[t] = false;
17         for(int i = h[t]; i != -1; i = ne[i])
18         {
19             int j = e[i];
20             if(dist[j] > dist[t] + w[i])
21             {
22                 dist[j] = dist[t] + w[i];
23                 if(!visit[j])
24                 {
25                     q.push(j);
26                     visit[j] = true;
27                 }
28             }
29         }
30     }
31     if(dist[n] == 0x3f3f3f3f) return -1;
32     return dist[n];
33 }

```

SPFA判断负环

```

1  #include<iostream>
2  #include<queue>
3  #include<algorithm>
4  #include<cstring>
5  using namespace std;
6  const int N = 2010, M = 10010;
7  int h[N], e[M], w[M], ne[M], idx;
8  int dist[N], cnt[N];
9  bool visit[N];
10 int n, m;
11 void add(int a, int b, int c)
12 {
13     e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx++;
14 }
15 bool spfa()
16 {
17     memset(dist, 0x3f, sizeof dist);    // 可省略
18     queue<int> q;
19     for(int i = 1; i <= n; i++)
20     {
21         q.push(i);
22         visit[i] = true;
23     }
24     while(q.size())
25     {
26         int t = q.front();
27         q.pop();
28         visit[t] = false;
29         for(int i = h[t]; i != -1; i = ne[i])
30         {
31             int j = e[i];

```

```

32         if(dist[j] > dist[t] + w[i])
33         {
34             dist[j] = dist[t] + w[i];
35             cnt[j] = cnt[t] + 1;
36             if(cnt[j] >= n) return true;
37             if(!visit[j])
38             {
39                 visit[j] = true;
40                 q.push(j);
41             }
42         }
43     }
44 }
45 return false;
46 }
47 int main(void)
48 {
49     cin >> n >> m;
50     memset(h, -1, sizeof h);
51     for(int i = 0; i < m; i++)
52     {
53         int a, b, c;
54         cin >> a >> b >> c;
55         add(a, b, c);
56     }
57     if(spfa()) puts("Yes");
58     else puts("No");
59     return 0;
60 }

```

Floyd算法

从i点出发，只经过k个点到达j点的最短距离

```

1  #include<iostream>
2  #include<cstring>
3  #include<algorithm>
4  using namespace std;
5  const int N = 210, INF = 1e9;
6  int g[N][N];
7  int n, m, Q;
8  void floyd()
9  {
10     for(int k = 1; k <= n; k++)
11     {
12         for(int i = 1; i <= n; i++)
13         {
14             for(int j = 1; j <= n; j++)
15                 g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
16         }
17     }
18 }
19 int main(void)
20 {
21     cin >> n >> m >> Q;

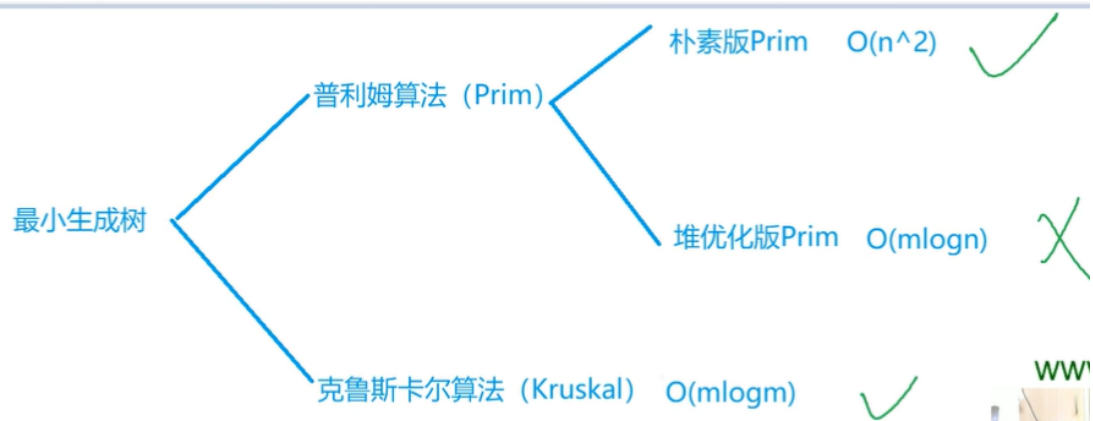
```

```

22 // 初始化
23 for(int i = 1; i <= n; i++)
24 {
25     for(int j = 1; j <= n; j++)
26     {
27         if(i == j) g[i][j] = 0;
28         else g[i][j] = INF;
29     }
30 }
31 while(m--)
32 {
33     int a, b, c;
34     cin >> a >> b >> c;
35     g[a][b] = min(g[a][b], c);
36 }
37 floyed();
38 for(int i = 0 ;i < Q; i++)
39 {
40     int a, b;
41     cin >> a >> b;
42     if(g[a][b] > INF / 2) puts("impossible");
43     else cout << g[a][b] << endl;
44 }
45 return 0;
46 }

```

最小生成树



稠密图用朴素版prim

稀疏图用kruskal

prim算法

朴素prim算法

① 朴素 Prim

5

$dist[i] \leftarrow +\infty$

for ($i=0$; $v < n$; $v++$)

找集合外距离最近的点

用它更新其他点到集合的距离

$st[v] = true$

通用模版

时间复杂度为 $O(n^2 + m)$, n 表示点数, m 表示边数

```
1 int n; // 表示点数
2 int g[N][N]; // 邻接矩阵, 存储所有边
3 int dist[N]; // 存储其他点到当前最小生成树的距离
4 bool visit[N]; // 存储每个点是否已经在生成树中
5 int prim()
6 {
7     memset(dist, 0x3f, sizeof dist);
8     int res = 0;
9     for(int i = 0; i < n; i++)
10     {
11         int t = -1;
12         for(int j = 1; j <= n; j++)
13         {
14             if(!visit[j] && (t == -1 || dist[t] > dist[j]))
15                 t = j;
16         }
17         if(i && dist[t] == INF) return INF;
18         if(i) res += dist[t];
19         visit[t] = true;
20         for(int j = 1; j <= n; j++)
21             dist[j] = min(dist[j], g[t][j]);
22     }
23     return res;
24 }
```

Kruskal 算法

通用模版

```
1 int n, m; // n是点数, m是边数
2 int p[N]; // 并查集的父节点数组
```

```

3  struct Edge{
4      int a, b, w;
5      bool operator< (const Edge &w) const
6      {
7          return w < W.w;
8      }
9  }edges[M];
10 int find(int x) // 并查集核心操作
11 {
12     if(p[x] != x) p[x] = find(p[x]);
13     return p[x];
14 }
15 int kruskal()
16 {
17     sort(edges, edges + m);
18     for(int i = 1; i <= n; i++) p[i] = i;
19     int res = 0, cnt = 0;
20     for(int i = 0; i < m; i++)
21     {
22         int a = edges[i].a, b = edges[i].b, w = edges[i].w;
23         a = find(a), b = find(b);
24         if(a != b)
25         {
26             p[a] = b;
27             res += w;
28             cnt ++;
29         }
30     }
31     if(cnt < n - 1) return INF;
32     return res;
33 }

```

算法实现

```

1  #include <cstring>
2  #include <iostream>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 100010, M = 200010, INF = 0x3f3f3f3f;
8
9  int n, m;
10 int p[N];
11
12 struct Edge
13 {
14     int a, b, w;
15
16     bool operator< (const Edge &w) const
17     {
18         return w < W.w;
19     }
20 }edges[M];
21

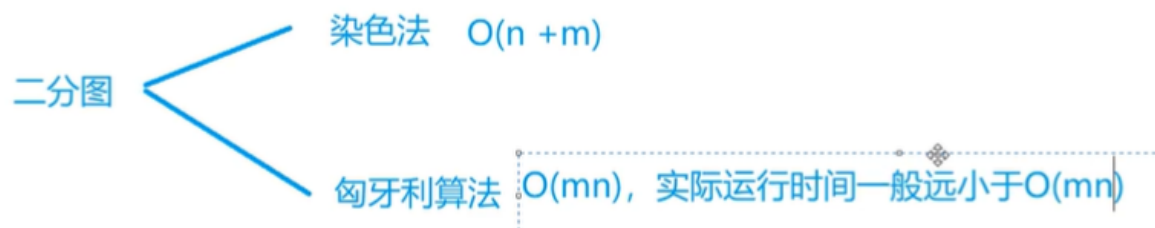
```

```

22 int find(int x)
23 {
24     if (p[x] != x) p[x] = find(p[x]);
25     return p[x];
26 }
27
28 int kruskal()
29 {
30     sort(edges, edges + m);
31
32     for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集
33
34     int res = 0, cnt = 0;
35     for (int i = 0; i < m; i ++ )
36     {
37         int a = edges[i].a, b = edges[i].b, w = edges[i].w;
38
39         a = find(a), b = find(b);
40         if (a != b)
41         {
42             p[a] = b;
43             res += w;
44             cnt ++ ;
45         }
46     }
47
48     if (cnt < n - 1) return INF;
49     return res;
50 }
51
52 int main()
53 {
54     scanf("%d%d", &n, &m);
55
56     for (int i = 0; i < m; i ++ )
57     {
58         int a, b, w;
59         scanf("%d%d%d", &a, &b, &w);
60         edges[i] = {a, b, w};
61     }
62
63     int t = kruskal();
64
65     if (t == INF) puts("impossible");
66     else printf("%d\n", t);
67
68     return 0;
69 }

```

二分图



染色法

判断一个图是不是二分图

一个图是二分图当且仅当图中不含奇数环（奇圈）

```
1  int n;    // n表示点数
2  int h[N], e[M], ne[M], idx;
3  int color[N];
4  bool dfs(int u, int c)
5  {
6      color[u] = c;
7      for(int i = h[u]; i != -1; i = ne[i])
8      {
9          int j = e[i];
10         if(color[j] == -1)
11         {
12             if(!dfs(j, !c)) return false;
13         }
14         else if (color[j] == c) return false;
15     }
16     return true;
17 }
18 bool check()
19 {
20     memset(color, -1, sizeof color);
21     bool flag = true;
22     for(int i = 1; i <= n; i++)
23         if(color[i] == -1)
24             if(!dfs(i, 0))
25             {
26                 flag = false;
27                 break;
28             }
29     return flag;
30 }
```

```
1  #include <cstring>
2  #include <iostream>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 100010, M = 200010;
8
```

```

9  int n, m;
10 int h[N], e[M], ne[M], idx;
11 int color[N];
12
13 void add(int a, int b)
14 {
15     e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
16 }
17
18 bool dfs(int u, int c)
19 {
20     color[u] = c;
21
22     for (int i = h[u]; i != -1; i = ne[i])
23     {
24         int j = e[i];
25         if (!color[j])
26         {
27             if (!dfs(j, 3 - c)) return false;
28         }
29         else if (color[j] == c) return false;
30     }
31
32     return true;
33 }
34
35 int main()
36 {
37     scanf("%d%d", &n, &m);
38
39     memset(h, -1, sizeof h);
40
41     while (m -- )
42     {
43         int a, b;
44         scanf("%d%d", &a, &b);
45         add(a, b), add(b, a);
46     }
47
48     bool flag = true;
49     for (int i = 1; i <= n; i ++ )
50         if (!color[i])
51         {
52             if (!dfs(i, 1))
53             {
54                 flag = false;
55                 break;
56             }
57         }
58
59     if (flag) puts("Yes");
60     else puts("No");
61
62     return 0;
63 }
64

```


匈牙利算法

```
1  int n1, n2;
2  int h[N], e[M], ne[M], idx;
3  int match[N];
4  bool visit[N];
5  bool find(int x)
6  {
7      for(int i = h[x]; i != -1; i = ne[i])
8      {
9          int j = e[i];
10         if(!visit[j])
11         {
12             visit[j] = true;
13             if(match[j] == 0 || find(match[j]))
14             {
15                 match[j] = x;
16                 return true;
17             }
18         }
19     }
20     return false;
21 }
22 int res = 0;
23 for(int i = 1; i <= n1; i++)
24 {
25     memset(visit, false, sizeof visit);
26     if(find(i)) res ++;
27 }
```

```
1  #include <cstring>
2  #include <iostream>
3  #include <algorithm>
4
5  using namespace std;
6
7  const int N = 510, M = 100010;
8
9  int n1, n2, m;
10 int h[N], e[M], ne[M], idx;
11 int match[N];
12 bool st[N];
13
14 void add(int a, int b)
15 {
16     e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
17 }
18
19 bool find(int x)
20 {
21     for (int i = h[x]; i != -1; i = ne[i])
22     {
23         int j = e[i];
```

```

24     if (!st[j])
25     {
26         st[j] = true;
27         if (match[j] == 0 || find(match[j]))
28         {
29             match[j] = x;
30             return true;
31         }
32     }
33 }
34
35 return false;
36 }
37
38 int main()
39 {
40     scanf("%d%d%d", &n1, &n2, &m);
41
42     memset(h, -1, sizeof h);
43
44     while (m -- )
45     {
46         int a, b;
47         scanf("%d%d", &a, &b);
48         add(a, b);
49     }
50
51     int res = 0;
52     for (int i = 1; i <= n1; i ++ )
53     {
54         memset(st, false, sizeof st);
55         if (find(i)) res ++ ;
56     }
57
58     printf("%d\n", res);
59
60     return 0;
61 }

```