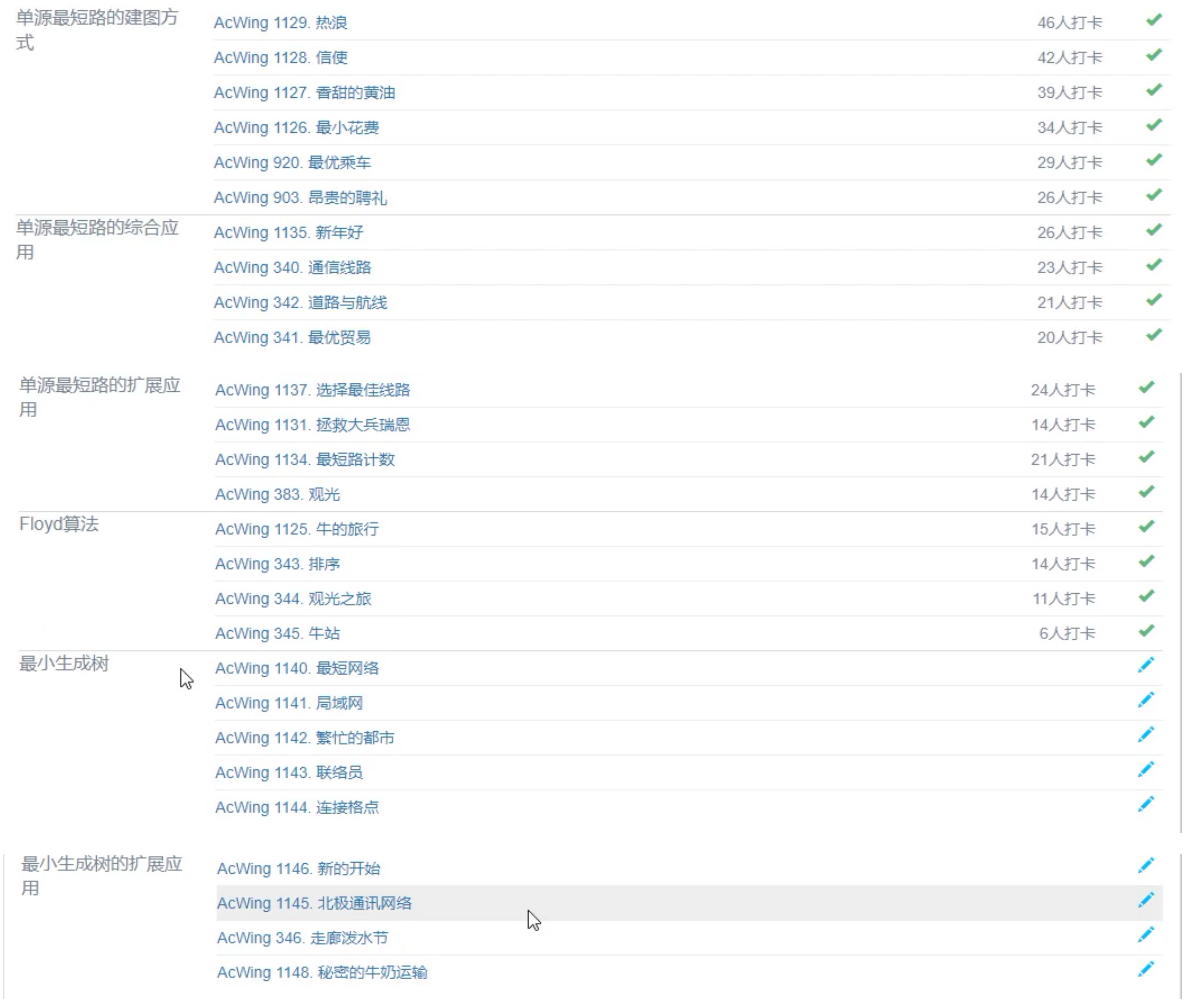
图论



单源最短路

建图方式

点集合，连通块

最优乘车

点集合

由于条线路可以到n个车站，所以可以把该条线路能到达的所有车站看成一个集合，集合内两个点联通的边权都设置成 1，因为所有点都在线路内，所以乘车次数为 1，也就是边权。这样从起点到该线路内的任意一点的乘车次数(最小花费) 都为 1。通往不同点 (不同线路) 的花费就会在原有的`dist[t]`上 + 1 了。因此得到的 `dist[n]` 就是从起点到 站点 `n` 的乘车次数，换成次数就为 `dist[n] - 1`

```
#include <iostream>
#include <sstream>
#include <string>
#include <algorithm>
#include <cstring>
using namespace std;

const int N = 510;
```

```
int n,m;
int dist[N],g[N][N],stop[N],q[N];
bool st[N];

void bfs()
{
    memset(dist,0x3f,sizeof dist);
    int hh = 0, tt = -1;
    q[++tt] = 1;
    dist[1] = 0;

    while( hh <= tt )
    {
        int t = q[hh++];
        st[t] = true;

        for(int j = 1; j <= n; j++)
            if(!st[j] && g[t][j] && dist[j] > dist[t] + 1)
            {
                dist[j] = dist[t] + 1;
                q[++tt] = j;
            }
    }

}

int main()
{
    cin >> m >> n;

    string line;
    getline(cin,line);
    for(int i = 0; i < m; i++)
    {
        getline(cin,line);
        stringstream ssin(line);
        int cnt = 0, p ;
        while(ssin >> p) stop[cnt ++ ] = p;
        for(int j = 0; j < cnt; j++) //组合枚举
            for(int k = j + 1; k < cnt; k++)
                g[stop[j]][stop[k]] = 1;
    }

    bfs();

    if(dist[n] == 0x3f3f3f3f)   puts("NO");
    else    cout << dist[n] - 1 << endl;

    return 0;
}
```

虚拟源点，限制可走路径

[昂贵的聘](#)

虚拟源点，有限制的可走路径

```cpp
#include <iostream>
#include <cstring>
#include <queue>
#include <algorithm>
using namespace std;

typedef pair<int,int> PII;

const int N = 110, M = N*N;
int h[N],ne[M],w[M],e[M],idx;
int level[N],dist[N],n,m;
bool st[N];

void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}

int dijkstra(int left,int right)
{
    priority_queue<PII,vector<PII>,greater<PII> > heap;
    memset(dist,0x3f,sizeof dist);
    memset(st,0,sizeof st);
    dist[0] = 0;
    heap.push({dist[0],0});

    while(heap.size())
    {
        PII t = heap.top();
        heap.pop();

        int ver = t.second;

        if(st[ver]) continue;
        st[ver] = true;

        for(int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(level[j] < left || level[j] > right) continue;
            if(dist[j] > t.first + w[i])
            {
                dist[j] = t.first + w[i];
                heap.push({dist[j],j});
            }
        }
    }

    return dist[1];
}

int main()
{
    cin >> m >> n;

    memset(h,-1,sizeof h);

    for(int i = 1; i <= n;i++)
```

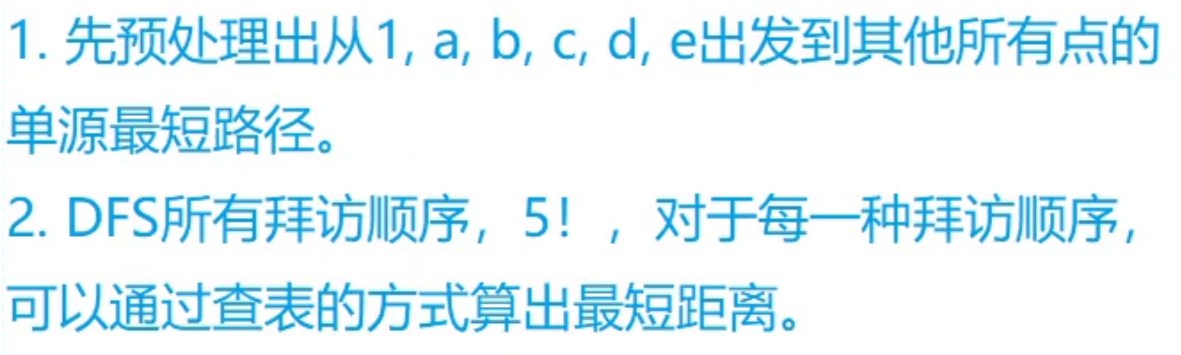
```
    {
        int price,cnt;
        cin >> price >> level[i] >> cnt;
        add(0,i,price);
        for(int j = 0; j < cnt; j ++)
        {
            int a,c;
            cin >> a >> c;
            add(a,i,c);
        }
    }

    int res = 0x3f3f3f3f;
    for(int i = level[1] - m; i <= level[1]; i++)
        res = min(res,dijkstra(i,i + m));

    cout << res << endl;

    return 0;
}
```

综合应用

深搜 + 最短路，查表优化

新年好

思路：




```
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
using namespace std;

typedef pair<int,int>PII;

const int N = 50010,M = 1e5 + 5,INF = 0x3f3f3f3f;
int dist[7][N],id[7],n,m,ans = 1e9;
int h[N],ne[M],w[M],e[M],idx;
bool st[N];

void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}
```

```
void dijkstra(int s,int d[])
{
    memset(st,0,sizeof st);
    memset(d,0x3f,N*4);
    d[s] = 0;
    priority_queue<PII,vector<PII>,greater<PII> >heap;

    heap.push({0,s});

    while(heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second;

        if(st[ver]) continue;
        st[ver] = true;

        for(int i = h[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(d[j] > d[ver] + w[i])
            {
                d[j] = d[ver] + w[i];
                heap.push({d[j],j});
            }
        }
    }
}

void dfs(int u,int cnt,int last,int res)
{
    if(cnt == 5)
    {
        ans = min(res,ans);
        return;
    }

    for(int i = 1; i <= 5; i++)
    {
        if(st[id[i]])   continue;
        st[id[i]] = true;
        dfs(id[i],cnt + 1,i,res + dist[last][id[i]]);
        st[id[i]] = false;
    }
}

int main()
{
    id[0] = 1;

    cin >> n >> m;

    for(int i = 1; i <= 5; i++ )    cin >> id[i];

    memset(h,-1,sizeof h);
```

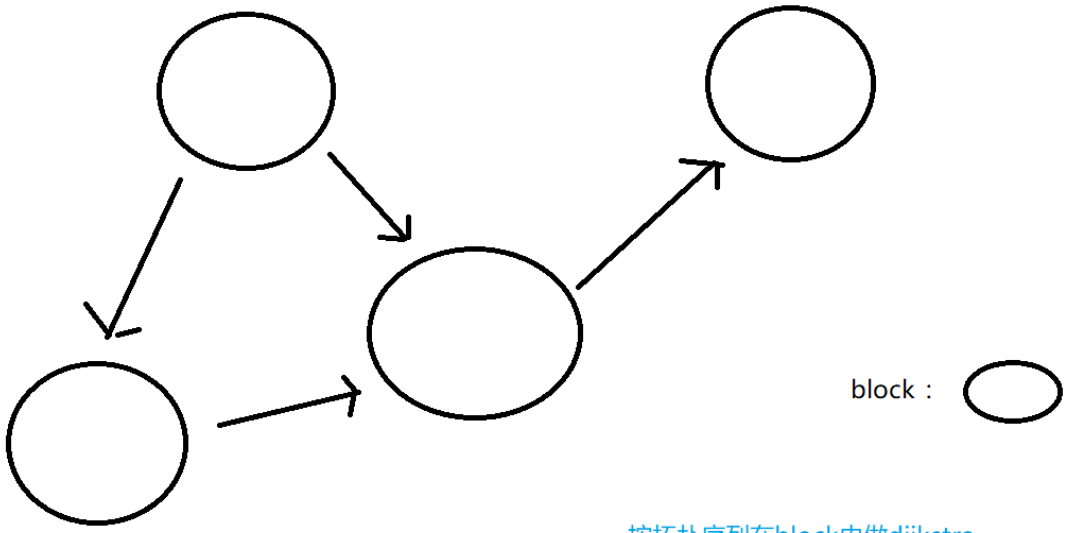
```
    for(int i = 0; i < m; i++)
    {
        int a,b,c;
        cin >> a >> b >> c;
        add(a,b,c);
        add(b,a,c);
    }

    for(int i = 0; i <= 5; i++) //打表，处理本人和亲戚到所有点的最短路
        dijkstra(id[i],dist[i]);

    memset(st,0,sizeof st);

    dfs(id[0],0,0,0); //枚举拜访顺序

    cout << ans << endl;

    return 0;
}
```

拓扑排序中的最短路

[道路与航线]

单源最短路 + 拓扑排序

1. 先输入所有双向道路，然后DFS出所有连通块，计算两个数组：id[]存储每个点属于哪个连通块；vector<int> block[]存储每个连通块里有哪些点；
2. 输入所有航线，同时统计出每个连通块的入度。
3. 按照拓扑序依次处理每个连通块。先将所有入度为0的连通块的编号加入队列中。
4. 每次从队头取出一个连通块的编号bid。
5. 将该block[bid]中的所有点加入堆中，然后对堆中所有点跑dijkstra算法。
6. 每次取出堆中距离最小的点ver。
7. 然后遍历ver的所有邻点j。如果id[ver] == id[j]，那么如果j能被更新，则将j插入堆中；如果id[ver] != id[j]，则将id[j]这个连通块的入度减1，如果减成0了，则将其插入拓扑排序的队列中。



按拓扑序列在block内做dijkstra。

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <cstring>
using namespace std;

const int N = 25010, M = 150010;

typedef pair<int,int> PII;

int T,R,P,S,blockcnt; //blockcnt 块的数量
int h[N],e[M],w[M],ne[M],idx = 1;

int id[N],dist[N],indegree[N]; //id[i] 存储点i的块的序号 indegree储存 块 的入度
bool st[N];
vector<int> block[N]; //存储道路块 i 里包含的节点
queue<int> q;

void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}

void dfs(int u,int bid) //找出所有公路联通的块
{
    id[u] = bid;
    block[bid].push_back(u);

    for(int i = h[u]; i ; i = ne[i])
    {
        int j = e[i];
        if(!id[j])  dfs(j,bid);
    }
}

void dijkstra(int bid)
```

```
{
    priority_queue<PII,vector<PII>,greater<PII> >heap;
    for(auto ver : block[bid])  heap.push({dist[ver],ver});//将当前块内的所有点入队

    while(heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second;

        if(st[ver]) continue;
        st[ver] = true;

        for(int i = h[ver]; i; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[ver] + w[i])
            {
                dist[j] = dist[ver] + w[i]; //更新块内的最短路，包括跨块的最短路
                if(id[ver] == id[j])    heap.push({dist[j],j}); // 在相同的块内才加入堆中
            }

            if(id[ver] != id[j] && --indegree[id[j]] == 0)  q.push(id[j]); //不在同一个块中，且入度为0入队
        }
    }
}

void topsort()
{
    memset(dist,0x3f,sizeof dist);

    dist[S] = 0;

    for(int i = 1; i <= blockcnt; i++) //枚举块序号
        if(!indegree[i])    q.push(i);

    while(q.size())
    {
        int t = q.front();
        q.pop();
        dijkstra(t);
    }
}

int main()
{
    scanf("%d %d %d %d",&T,&R,&P,&S);

    for(int i = 0; i < R; i++)//连接道路
    {
        int a,b,c;
        scanf("%d %d %d",&a,&b,&c);
        add(a,b,c),add(b,a,c);
    }
```

```
    for(int i = 1; i <= T; i++)
        if(!id[i])  dfs(i,++blockcnt);

    for(int i = 0; i < P; i++)//连接航班 ，拓扑
    {
        int a,b,c;
        scanf("%d %d %d",&a,&b,&c);
        add(a,b,c);
        indegree[id[b]]++; //将b对应的块编号入度++
    }

    topsort();

    for(int i = 1; i <= T; i++)
        if(dist[i] > 0x3f3f3f3f/2)  puts("NO PATH");
        else    printf("%d\n",dist[i]);

    return 0;
}
```

最短路 + 动态规划

[最优贸易]

最短路 + DP

```
#include<iostream>
#include<algorithm>
#include<queue>
#include<cstring>
using namespace std;

const int N = 1e5 + 10, M = 2e6 + 10;

int n,m;
int hs[N],ht[N],e[M],ne[M],idx;
int dmin[N],dmax[N],w[N]; //dmin[i] 前i个物品中最小的花费(最便宜)
bool st[N]; //dmax[i] 前i个物品中最大的花费(最贵)，跑反向图

void add(int h[],int a,int b)
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx++;
}

void spfa(int h[],int dist[],int type)
{
    queue<int> q;
    if(type == 0) //求最小
    {
        memset(dist,0x3f,sizeof dmin);
        q.push(1);
        dist[1] = w[1];
    }
    else    //求最大
    {
        memset(dist,-0x3f,sizeof dmax);
        q.push(n);
        dist[n] = w[n];
```

```
    }

    while(q.size())
    {
        int t = q.front();
        q.pop();

        st[t] = false;

        for(int i = h[t]; ~i; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > min(dist[t],w[j]) && type == 0 || dist[j] <
max(dist[t],w[j]) && type)
            {
                if(type == 0)   dist[j] = min(dist[t],w[j]);
                else    dist[j] = max(dist[t],w[j]);

                if(!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
}

int main()
{
    cin >> n >> m;

    memset(hs,-1,sizeof hs);
    memset(ht,-1,sizeof ht);

    for(int i = 1; i <= n; i++) cin >> w[i]; //每个节点物品的数量

    for(int i = 1; i<= m; i++)
    {
        int a,b,c;
        cin >> a >> b >> c;
        add(hs,a,b),add(ht,b,a);    //hs,正向图 ht,反向图
        if(c == 2)  add(hs,b,a),add(ht,a,b); // c == 2，双向图
    }

    spfa(hs,dmin,0); //正向跑spfa，求最小花费购买物品
    spfa(ht,dmax,1); //反向跑spfa，求最大花费出售物品

    int res = 0;
    for(int k = 1; k <= n; k++) res = max(res,dmax[k] - dmin[k]); //枚举分界点，找
出最大的 dmin[] - dmax[]

    cout << res << endl;

    return 0;
}
```

扩展应用

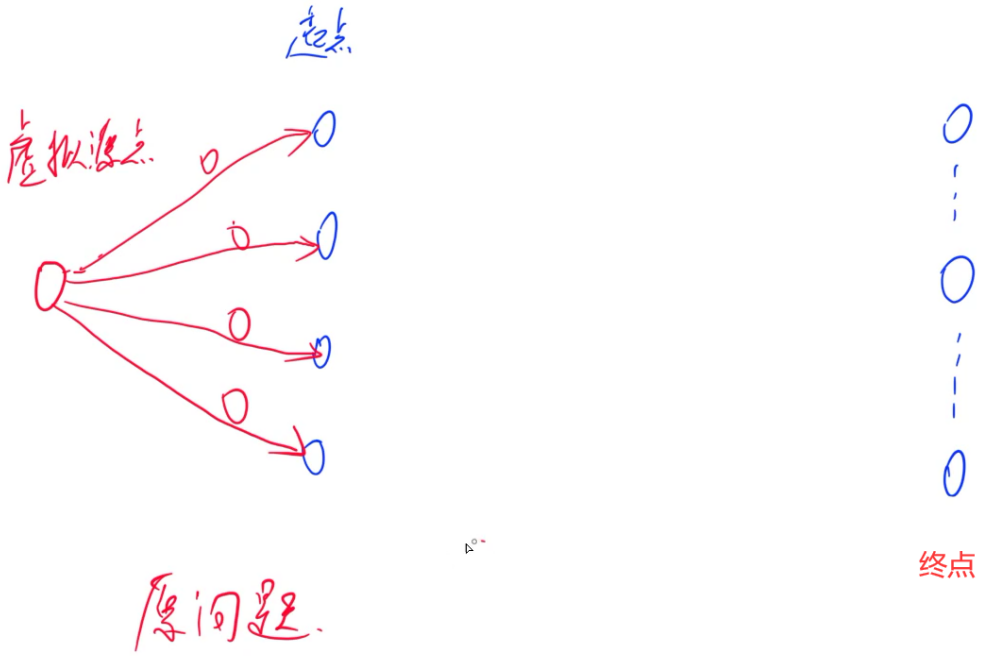
多起点最短路

选择最佳路线

这题有两种做法

一种是建一个反向图，求一遍重点到每个起点的距离，但这种做法仅限于只有一个终点的情况

另一种是建立一个到所有起点的边权都是 0 的虚拟源点，这种做法可以适合有多终点的情况，即多个起点可选择通往多个终点



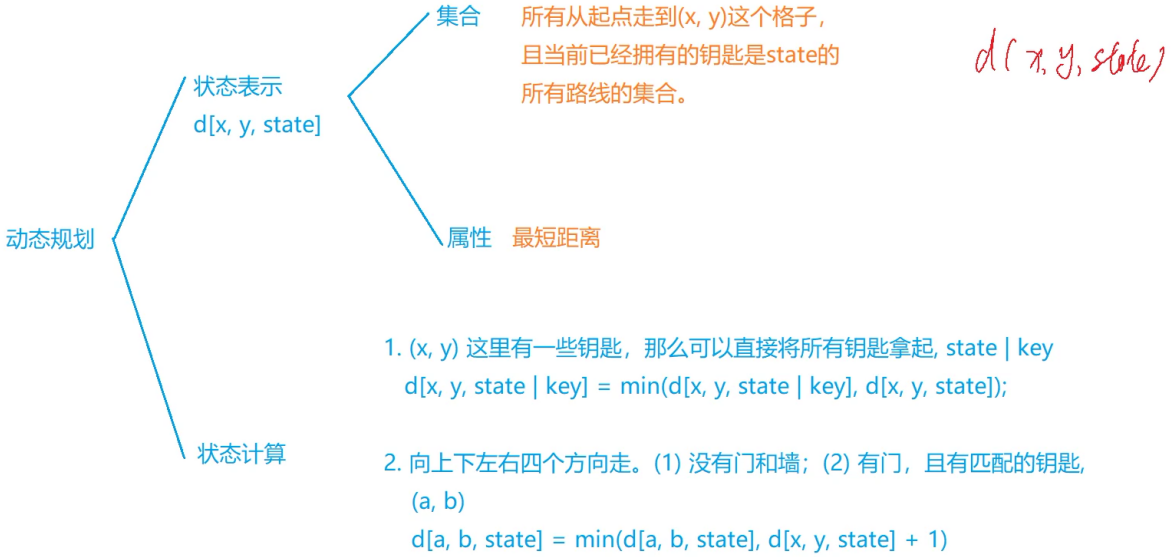

```
#include<iostream>
#include<algorithm>
#include<queue>
#include<cstring>
using namespace std;

const int N = 1010, M = 25010;
int h[N],e[M],ne[M],w[M],idx;
int dist[N],q[N];
bool st[N];
int n,m,T;

void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}

int spfa()
{
    memset(dist,0x3f,sizeof dist);
    int hh = 0, tt = 0;
    q[tt++] = 0;
    dist[0] = 0;
```

```
    while(hh != tt)
    {
        int t = q[hh++];
        if(hh == N) hh = 0;
        st[t] = false;

        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if(!st[j])
                {
                    q[tt++] = j;
                    if(tt == N) tt = 0;
                    st[j] = true;
                }
            }
        }
    }

    if(dist[T] == 0x3f3f3f3f)   return -1;
    else    return dist[T];
}

int main()
{
    while(scanf("%d %d %d",&n,&m,&T) != -1)
    {
        memset(h,-1,sizeof h);
        idx = 0;
        for(int i = 0; i < m ; i++)
        {
            int a,b,c;
            scanf("%d %d %d",&a,&b,&c);
            add(a,b,c);
        }
        int s;
        scanf("%d",&s);
        for(int i = 0; i < s; i++)
        {
            int ver;
            scanf("%d",&ver);
            add(0,ver,0); //建立虚拟源点
        }
        cout << spfa() << endl;
    }

    return 0;
}
```

拆点，分层图

这种题型的是将图论问题转化成DP问题，按DP（接合角度）问题分析后，若该类图是拓扑序的则可以通过DP求解，**DP只能求解拓扑序上的最短路问题**，否则需要再由DP转化成最短路求解

[拯救大兵瑞恩]



state是一个二进制状态，由于最多可能存在10种钥匙，所以state最多存在10位 `0 1`，当第 `i` 位上的位数为 1 时表示已经有该种钥匙 `i`

只有存在该种钥匙 `i` 时才能通过边权为 `w[i] == i(钥匙 i 的类型)` 的路径

由于拾钥匙并不花费时间，所以拾钥匙的花费为 `0` ，而向相邻的两个点移动时，花费是 `1`，所以只存在 `0,1` 两种花费，故可以使用双端队列BFS来解决，并且双端队列BFS的时间复杂度是稳定线性的。

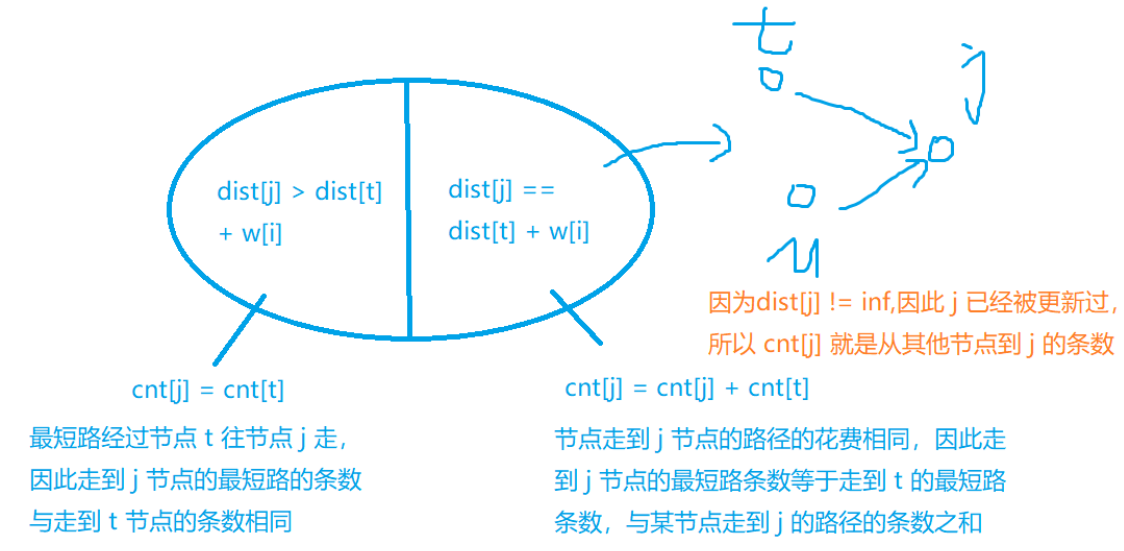
```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<set>
#include<deque>
using namespace std;

const int N = 11, M = N * N, E = M * M,P = 1 << 11;

int h[M],e[E],w[E],ne[E],idx;
int dist[M][P],g[N][N],key[M],n,m,p; //g存的是二维坐标（x,y）映射成一维坐标t，key[i]是存储点 i 下是否有钥匙，和钥匙的种类（二进制数），dist[ver][state]存储的是经过前ver个点，且手中钥匙状态为state的最短路，state(二进制数)
bool st[M][P]; //判断当前状态有没有访问过

typedef pair<int,int> PII;

set<PII> edges;//存储哪些点之间是存在障碍的，如墙，门

void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}

void build()
{
    int dx[] = {0,1,-1,0} , dy[] = {1,0,0,-1};

    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            for(int k = 0; k < 4; k++) //向四个方向扩展
```

```
            {
                int i2 = i + dx[k] , j2 = j + dy[k];
                if(!i2 || i2 > n || !j2 || j2 > m)  continue;
                int a = g[i][j] , b = g[i2][j2];
                if(edges.count({a,b}) == 0) add(a,b,0); //如果a，b之间不存在障碍，连
边
            }
}

int bfs()
{
    deque<PII> q; //初始化
    memset(dist,0x3f,sizeof dist);
    dist[1][0] = 0;
    q.push_back({1,0});

    while(q.size())
    {
        auto t = q.front();
        q.pop_front();

        int ver = t.first, state = t.second;

        if(ver == g[n][m])  return dist[ver][state];//当当前点已经到终点是，直接返回

        if(st[ver][state])  continue; //BFS每次遍历到的都是最短，所以出队有标记一下已访
问
        st[ver][state] = true;

        if(key[ver]) //当前所以在点有钥匙
        {
            int newstate = state | key[ver]; //新的二进制状态 newstate
            if(dist[ver][newstate] > dist[ver][state])
            {
                dist[ver][newstate] = dist[ver][state]; //更新最短路
                q.push_front({ver,newstate}); //因为花费是0所以加到队头
            }
        }

        for(int i = h[ver]; i != -1; i = ne[i]) //遍历邻边
        {
            int j = e[i];
            if(w[i] && !( (state >> w[i] - 1) & 1) )    continue; //可到达的点之间
存在门，且没有该门的钥匙，continue
            if(dist[j][state] > dist[ver][state] + 1)
            {
                dist[j][state] = dist[ver][state] + 1;
                q.push_back({j,state}); //花费是1，加到队尾
            }
        }
    }

    return -1;
}

int main()
{
    cin >> n >> m >> p;
```

```
    int k;
    cin >> k;

    for(int i = 1,t = 1; i <= n; i++) //打表，存（x,y）的映射一维坐标
        for(int j = 1; j <= m; j++)
            g[i][j] = t++;

    memset(h,-1,sizeof h);

    for(int i = 0; i < k; i++)
    {
        int x1,y1,x2,y2,c;
        cin >> x1 >> y1 >> x2 >> y2 >> c;
        int a = g[x1][y1], b = g[x2][y2];
        edges.insert({a,b}),edges.insert({b,a}); // a ，b之间存在障碍
        if(c)   add(a,b,c),add(b,a,c);  //障碍是门，存储门的类型c
    }

    build(); //建图

    int s;
    cin >> s;
    for(int i = 0; i < s; i++)
    {
        int x,y,id;
        cin >> x >> y >> id;
        key[g[x][y]] |= 1 << id - 1; //存储钥匙，和钥匙类型，-1是因为这样可以省出最低位二进制数，因为id是从1开始的
    }

    cout << bfs() << endl;

    return 0;
}
```

最短路计数问题

最短路计数可以看成是在一个**拓扑序**上做DP，类似于背包问题的count属性。



要注意权值为 0 的自环和负权自环！！！

切记！！！一定要在拓扑序上才能用DP求解，因此路径应该满足拓扑序

只有bfs，dijsktra(第一次出队)构成的序列才是拓扑序，spfa出队顺序不满足拓扑序(可以spfa跑出来最短路，再根据dist[j] == dist[t] + w[i] 来构建一颗拓扑树，然后在树上做DP)

[最短路计数]

```
#include<iostream>
#include<cstring>
#include<queue>
using namespace std;

const int N = 1e5 + 10, M = 4e5 + 10,mod = 100003;

int h[N],e[M],w[M],ne[M],idx = 1;
int dist[N],cnt[N],n,m;
bool st[N];

typedef pair<int,int> PII;

void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}

void dijsktra()
{
    priority_queue<PII,vector<PII>,greater<PII> >heap;
    memset(dist,0x3f,sizeof dist);
    heap.push({0,1});
    dist[1] = 0;
    cnt[1] = 1;

    while(heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.second;

        if(st[ver]) continue;
        st[ver] = true;

        for(int i = h[ver]; i; i = ne[i])
        {
            int j = e[i];
            if(dist[j] > dist[ver] + w[i])
            {
                dist[j] = dist[ver] + w[i];
                cnt[j] = cnt[ver]; //第一种情况
                heap.push({dist[j],j});
            }
            else if(dist[j] == dist[ver] + w[i]) //第二种情况
            {
                cnt[j] = (cnt[j] + cnt[ver]) % mod;
            }
        }
    }
```

```
}

int main()
{
    cin >> n >> m;
    while(m--)
    {
        int a,b;
        cin >> a >> b;
        add(a,b,1),add(b,a,1);
    }

    dijsktra();

    for(int i = 1; i <= n; i++)
        cout << cnt[i] << endl;

    return 0;
}
```

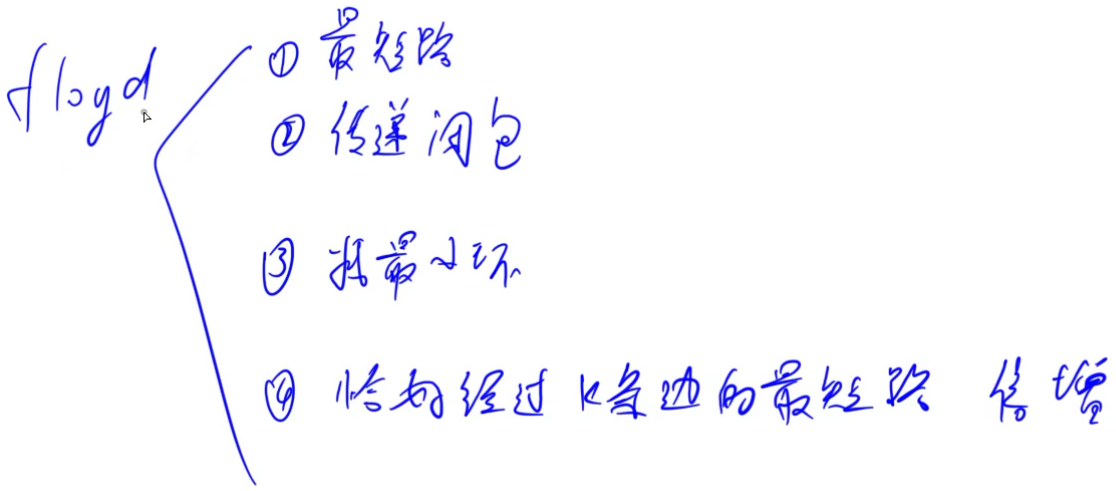
次短路

```
#include<cstdio>
#include<iostream>
#include<queue>
#include<cstring>
using namespace std;

const int N = 1010, M = 10010;

struct Ver
{
    int ver,type,dist;
    bool operator > (const Ver &t)  const
    {
        return dist > t.dist;
    }
};

int n,m,S,T;
int h[N],e[M],ne[M],w[M],idx;
int dist[N][2],cnt[N][2]; // dist[][0]最短路 dist[][1]次短路
bool st[N][2];

void add(int a,int b,int c)
{
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}

int dijsktra()
{
    memset(dist,0x3f,sizeof dist);
    memset(st,0,sizeof st);
    memset(cnt,0,sizeof cnt);
    priority_queue<Ver,vector<Ver>,greater<Ver> >heap;
    dist[S][0] = 0;
    cnt[S][0] = 1;
```

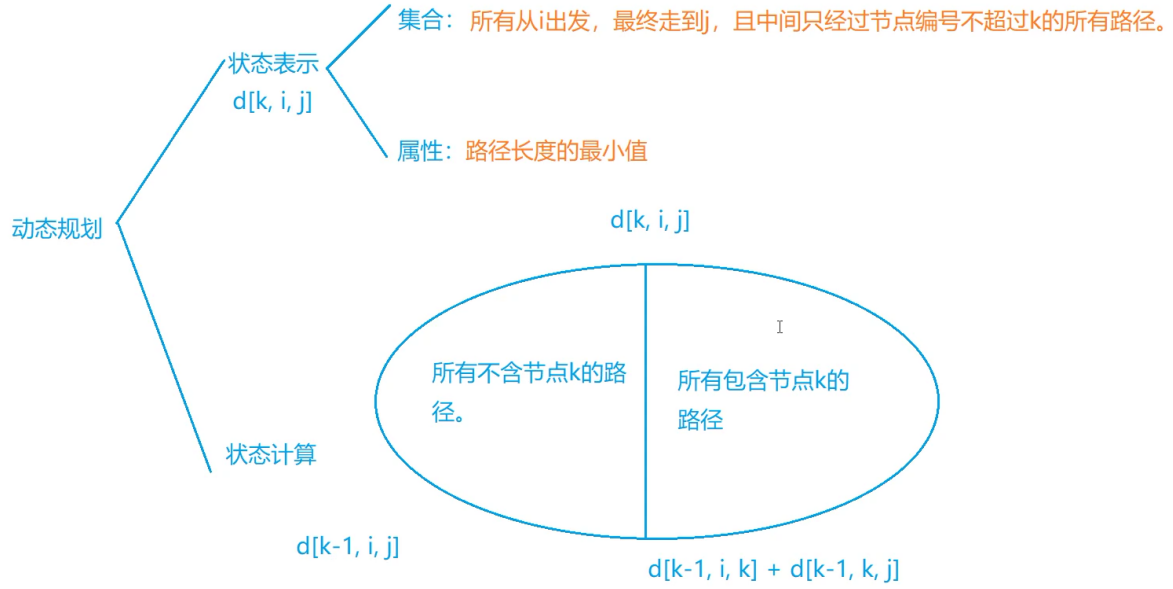
```
    heap.push({S,0,0});

    while(heap.size())
    {
        Ver t = heap.top();
        heap.pop();

        int ver = t.ver , type = t.type , distance = t.dist , count = cnt[ver]
[type];

        if(st[ver][type])   continue;
        st[ver][type] = true;

        for(int i = h[ver]; i; i = ne[i])
        {
            int j = e[i];
            if(dist[j][0] > distance + w[i]) //注意ifelse的顺序不能乱，要先更新最短
路，再更新次短路
            {
                dist[j][1] = dist[j][0],cnt[j][1] = cnt[j][0];
                heap.push({j,1,dist[j][1]});
                dist[j][0] = distance + w[i], cnt[j][0] = count;
                heap.push({j,0,dist[j][0]});
            }
            else if(dist[j][0] == distance + w[i])  cnt[j][0] += count;
            else if(dist[j][1] > distance + w[i] && dist[j][0] < distance +
w[i])
            {//严格次短路
                dist[j][1] = distance + w[i],cnt[j][1] = count;
                heap.push({j,1,dist[j][1]});
            }
            else if(dist[j][1] == distance + w[i])  cnt[j][1] += count;
        }
    }

    int res = cnt[T][0];
    if(dist[T][0] + 1 == dist[T][1])    res += cnt[T][1];

    return res;
}

int main()
{
    int cases;
    scanf("%d",&cases);
    while(cases--)
    {
        idx = 1;
        memset(h,0,sizeof h);
        scanf("%d %d",&n,&m);
        while(m--)
        {
            int a,b,c;
            scanf("%d %d %d",&a,&b,&c);
            add(a,b,c);
        }
        scanf("%d %d",&S,&T);
```

```
        printf("%d\n",dijsktra());
    }

    return 0;
}
```

多源最短路



floyd原理：



多源连通块的最短路

牛的旅行

1. 用floyd算法求出任意两点之间的最短距离
2. 求maxd[i]， 表示和i连通的且距离i最远的点的距离
3. 情况1：所有maxd[i]的最大值
   情况2：枚举在哪两个点之间连边。i, j，需要满足d[i, j] = INF。maxd[i] + dist[i, j] + maxd[j]

```
#include<iostream>
#include<algorithm>
```

```cpp
#include<cmath>
using namespace std;

#define x first
#define y second
#define INF 0x3f3f3f3f

typedef pair<int,int> PII;

const int N = 155;

PII q[N];
char g[N][N];
double d[N][N],maxd[N];

double get_dist(PII a,PII b)//计算边权
{
    double dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}

int main()
{
    int n;
    cin >> n;
    for(int i = 0; i < n; i++)  cin >> q[i].x >> q[i].y;//存坐标

    for(int i = 0; i < n; i++) cin >> g[i];//存领接矩阵

    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            if(i != j)
            {
                if(g[i][j] == '1')  d[i][j] = get_dist(q[i],q[j]);
                else    d[i][j] = INF;
            }
    //floyd
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                d[i][j] = min(d[i][j],d[i][k] + d[k][j]);

    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            if(d[i][j] < INF)
                maxd[i] = max(maxd[i],d[i][j]);//找到某个点走到i的最大值

    double res1 = 0;
    for(int i = 0; i < n; i++)  res1 = max(res1,maxd[i]);//找连通块里的直径

    double res2 = INF;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            if(d[i][j] >= INF)//i，j属于两个不同的连通块
                res2 = min(res2,maxd[i] + get_dist(q[i],q[j]) + maxd[j]); //连接，就最小直径

    printf("%lf\n",max(res1,res2));//最大直径
```

```
    return 0;
}
```

求传递闭包

传递闭包： 若 A -> B , b -> C ， 则它们之间的传递闭包就是 A -> C

floyd实现：

```
void floyd()
{
    memcpy(d,g,sizeof d);

    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                d[i][j] |= d[i][k] && d[k][j]; //注意是 |= ，因为如果已经可以连通，那么就算i走不到k ， k 走不到 j 也不能改变 i 到 j 的连通性
}
```

排序

闭包传递的一个经典应用

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;

const int N = 30;
int d[N][N],g[N][N];
bool st[N];
int n,m;

void floyd() //floyd实现传递闭包
{
    memcpy(d,g,sizeof d);

    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                d[i][j] |= d[i][k] && d[k][j];
}

int check()
{
    for(int i = 0; i < n; i++)
        if(d[i][i]) return 2; // 存在 A < A 的情况，返回 2

    for(int i = 0; i < n; i++)
        for(int j = 0; j < i; j++)
            if(!d[i][j] && !d[j][i]) //即 A < B 和 B < A 只能存在一个
                return 0; //还不能确定唯一的关系
```

```
    return 1; //可以确定唯一的关系
}

char get_min() //找到关系里面最小的数
{
    for(int i = 0; i < n; i++)
        if(!st[i])
        {
            bool flag = true;
            for(int j = 0; j < n; j++)
                if(!st[j] && d[j][i]) // j没有出队过 &&  j < i 成立
                {
                    flag = false;
                    break;
                }
            if(flag)
            {
                st[i] = true; //标记后，i 就已经出队了
                return 'A' + i;
            }
        }
}

int main()
{
    while(cin >> n >> m , n || m)
    {
        memset(g,0,sizeof g);

        int type = 0,t;
        char str[5];
        for(int i = 1; i <= m; i++)
        {
            cin >> str;
            int a = str[0] - 'A' , b = str[2] - 'A';

            if(!type)
            {
                g[a][b] = 1; //每次确定一个关系后，求一下传递闭包 ，这个关系指 A < B
                floyd();
                type = check();//检查是否符合
                if(type)    t = i;//迭代次数
            }
        }

        if(!type)   puts("Sorted sequence cannot be determined.");
        else if(type == 2)  printf("Inconsistency found after %d
relations.\n",t);
        else
        {
            memset(st,0,sizeof st);
            printf("Sorted sequence determined after %d relations: ",t);
            for(int i = 0; i < n; i++)  printf("%c",get_min());
            printf(".\n");
        }
    }

    return 0;
```

```
}
```

求最小环

恰好经过N条边的最短路

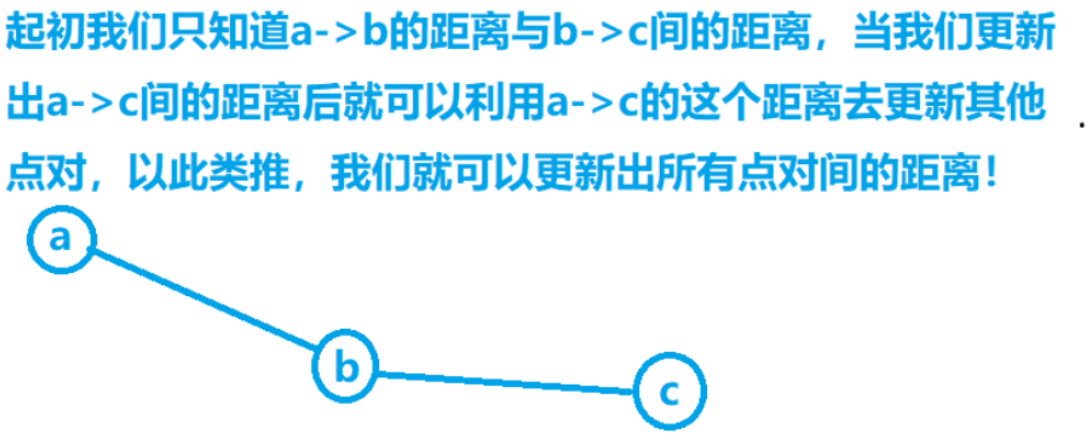
牛站

类floyd算法：

普通Floyd

```
for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
                d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
```

我们不难发现之所以Floyd可以得到所有点对间的最短距离是因为每次都是一条边一条边的的更新，图示如下



类Floyd算法

```
static int temp[N][N];
    memset(temp,0x3f,sizeof temp);
    for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
                temp[i][j]=min(temp[i][j],a[i][k]+b[k][j]);
```

我们发现做一次类Floyd好像不用并不会去用该次的结果自我更新，只会用上一次的结果来更新一次，不像Floyd那样可以自己更新自己多次！图示如下

起初我们只知道a->b的距离与b->c间的距离，当我们更新出a->c间的距离后却不能利用a->c的这个距离去更新其他点对，因为我们只更新temp这个临时数组，却没有用更新后的结果再去更新temp自己，而是继续用a和b去更新，其实这里的a和b就是temp上一次更新的结果

换句话说就是我们每次只能利用a，b的结果去更新temp，这样的话我们就做到了更新的边数限制，因为我们每次都要初始化temp，保证了更新来源唯一，第一次拿着1条边的结果更新了2条边的结果，第二次拿着2条边的结果更新了四条边的结果，以成倍的速度更新！

知道了区别后应该就不难理解为什么类Floyd的算法具有了边数的特性了

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<map>
using namespace std;

const int N = 210;
int k,idx,T,S,E;
int g[N][N],res[N][N];

map<int,int> ids; //用map做离散化处理，因为有1000个点，而边最多只有 100

void mul(int c[][N],int a[][N],int b[][N])
{
    static int temp[N][N];
    memset(temp,0x3f,sizeof temp);

    for(int k = 1; k <= idx; k++)
        for(int i = 1; i <= idx; i++)
            for(int j = 1; j <= idx; j++)
                temp[i][j] = min(temp[i][j],a[i][k] + b[k][j]);

    memcpy(c,temp,sizeof temp); //一定得注意sizeof 对象要是数组，不能是指针
}

void qmi() //采用快速幂的方式降低时间复杂度，因为从 i -> j 的过程中，经过的点具有独立性，即运算符合结合律
{
```

```
    memset(res,0x3f,sizeof res);
    for(int i = 1; i <= idx; i++)   res[i][i] = 0;

    while(k)
    {
        if(k & 1)   mul(res,res,g);
        mul(g,g,g);
        k >>= 1;
    }
}

int main()
{
    cin >> k >> T >> S >> E;

    memset(g,0x3f,sizeof g);

    ids[S] = ++idx;
    ids[E] = ++idx;

    while(T--)
    {
        int a,b,c;
        cin >> c >> b >> a;
        if(!ids.count(a))   ids[a] = ++idx;
        if(!ids.count(b))   ids[b] = ++idx;

        int x = ids[a], y = ids[b];
        g[x][y] = g[y][x] = min(g[x][y],c);
    }

    qmi();

    cout << res[ids[S]][ids[E]] << endl;

    return 0;
}
```

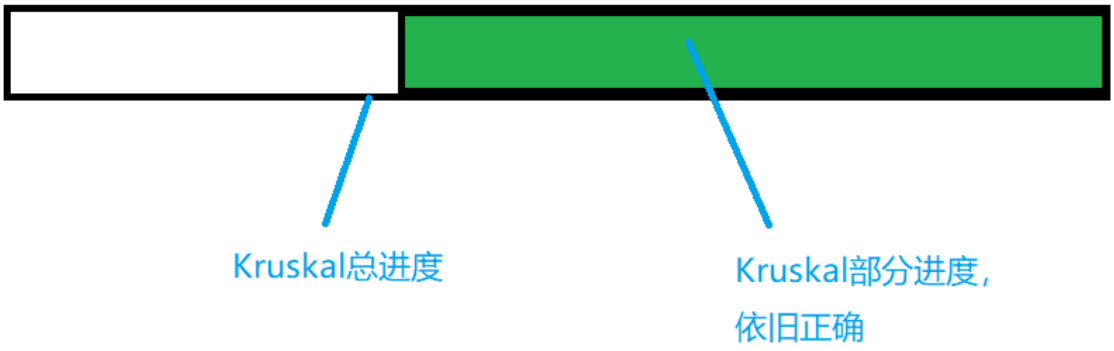
最小生成树

prim算法和Kruskal算法：

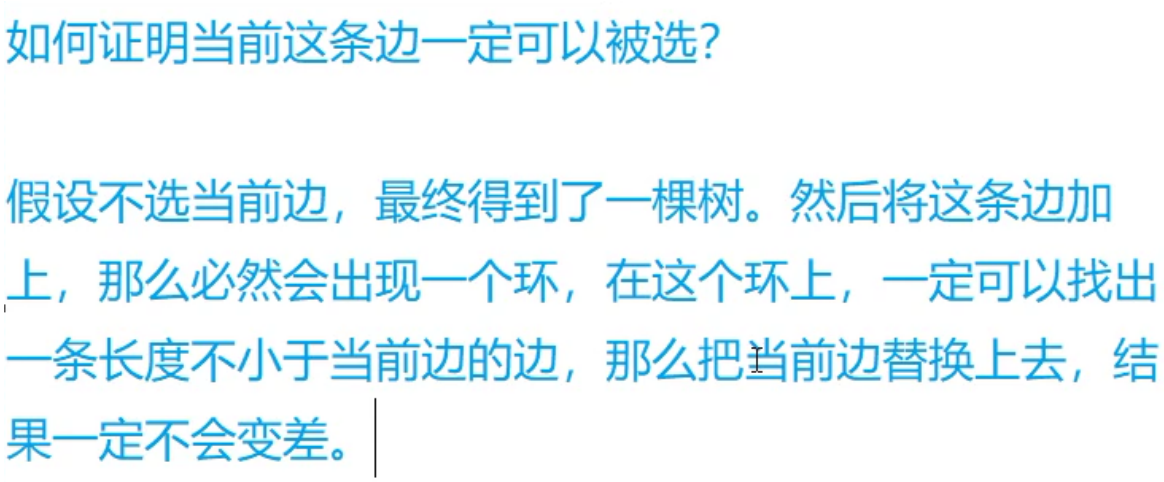
prim算法由某个点向外扩张，每次加进来的边都是最小生成树里的边，与之连通的点也都是最小生成树里的节点

Kruskal算法，先将边排序，然后依次将边最小的两个点加入到最小生成树里的集合

Kruskal算法有个特点就是，它的每一步都是正确的，无论从哪里开始，都是正确的



所加进去的边一定都是最小生成树里的边。



prim算法的简单应用

prim算法处理邻接矩阵会好处理一些

最短网格

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;

const int N = 105;

int n,g[N][N];
int dist[N];
bool st[N];

int prim()
{
    memset(dist,0x3f,sizeof dist);
    dist[1] = 0;
    int res = 0;

    for(int i = 1; i <= n; i++)
    {
```

```
        int t = -1;
        for(int j = 1; j <= n; j++)
            if(!st[j] && (t == -1 || dist[j] < dist[t]))
                t = j;

        res += dist[t];
        st[t] = true;

        for(int j = 1; j <= n; j++)
            dist[j] = min(dist[j],g[t][j]);
    }

    return res;
}

int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            cin >> g[i][j];

    cout << prim() <<endl;

    return 0;
}
```

Kruskal算法简单应用

局域网

题目要求的是不在最小生成树里的边的权值和

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;

const int N = 105,M = 210;

int n,m,p[N];

struct Edge{
    int a,b,w;
    bool operator < (const Edge & t)    const //注意两个const必须加上
    {
        return w < t.w;
    }
}edges[M];

int find(int x)
{
    if(p[x] != x)   p[x] = find(p[x]);
    return p[x];
}
```

```
int main()
{
    cin >>n >> m;

    for(int i = 1; i <= n; i++) p[i] = i;

    for(int i = 0; i < m; i++)
    {
        int a,b,c;
        cin >> a >> b >> c;
        edges[i] = {a,b,c};
    }

    sort(edges,edges + m);

    int res = 0;
    for(int i = 0; i < m; i++)
    {
        int pa = find(edges[i].a) , pb = find(edges[i].b);
        if(pa != pb)    p[pa] = pb;
        else    res += edges[i].w; //因为题目要求的是不在最小生成树里的边
    }

    cout << res <<endl;

    return 0;
}
```

连通块之间的最小生成树

连通块之间的最小生成树，与普通的最小生成树非常类似，只是相当于把连通块看成一个点，然后进行最小生成树的操作。这个过程也叫**缩点**

联络员

简单的连通块间的最小生成树问题

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;

const int N = 2010,M = 10010;

int n,m,p[N];

struct Edge{
    int a,b,w;
    bool operator < (const Edge & t)    const
    {
        return w < t.w;
    }
}edges[M];

int find(int x)
{
```

```
    if(p[x] != x)   p[x] = find(p[x]);
    return p[x];
}

int main()
{
    cin >>n >> m;

    for(int i = 1; i <= n; i++) p[i] = i;

    int res = 0,cnt = 0;
    for(int i = 0; i < m; i++)
    {
        int t,a,b,c;
        cin >> t >> a >> b >> c;

        if(t == 1) //缩点过程
        {
            res += c;
            p[find(a)] = find(b);
        }
        else edges[cnt++] = {a,b,c}; //不在一个连通块才加入
    }

    sort(edges,edges + cnt);//缩点后剩下cnt个点

    for(int i = 0; i < cnt; i++)
    {
        int pa = find(edges[i].a) , pb = find(edges[i].b);
        if(pa != pb)
        {
            p[pa] = pb;
            res += edges[i].w;
        }
    }

    cout << res <<endl;

    return 0;
}
```

拓展应用

最近公共祖先 LCA

倍增LCA

倍增LCA运用了二进制组合的方法

`fa[i][k]`储存的是节点号`i`向上跳2^k次的节点

因此要求`fa[i][k]`，可以用`fa[fa[i][k-1]][k-1]`来求

因为`k`定义的是跳跃2^k次，而`2^k`又可以由$2^{k-1}+2^{k-1}$组合而成

找公共祖先步骤

1. 先将 a 跳到和 b 同一深度 `应满足（depth[a] > depth[b]）`
2. a，b同时往上跳，但`fa[a][k] != fa[b][k]`，直到`fa[a][0] == fa[b][0]`为止
3. `fa[a][0] 或 fa[b][0]` 即为 a，b 的最近公共祖先

[祖孙询问]

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<queue>
using namespace std;

const int N = 4e4 + 10, M = 8e4 + 10;

int h[N],e[M],ne[M],idx;
int depth[N],fa[N][17],n,m;

void add(int a,int b)
{
    e[idx] = b,ne[idx] = h[a],h[a] = idx++;
}

void bfs(int root)
{
    memset(depth,0x3f,sizeof depth);
    depth[root] = 1 , depth[0] = 0; //depth[0]哨兵
    queue<int>q;
    q.push(root);

    while(q.size())
    {
        int t = q.front();
        q.pop();

        for(int i = h[t]; ~i ; i = ne[i])
        {
            int j = e[i];
            if(depth[j] > depth[t] + 1)
            {
                depth[j] = depth[t] + 1;
                fa[j][0] = t;
                q.push(j);
                for(int k = 1; k <= 15; k++)
                    fa[j][k] = fa[fa[j][k-1]][k-1]; //倍增跳跃
            }
        }
    }
}

int lca(int a,int b)
{
    if(depth[a] < depth[b]) swap(a,b); //如果 a 在 b 的上面

    for(int k = 15; k >= 0; k--) //枚举
        if(depth[fa[a][k]] >= depth[b]) //如果可以跳 二进制组合未组满
```

```
            a = fa[a][k]; //跳
    //退出循环时，a ，b 在同一层
    if(a == b)  return a;

    for(int k = 15; k >= 0; k--)
        if(fa[a][k] != fa[b][k]) //当前节点的第2^k个父节点，不同
        {
            a = fa[a][k];
            b = fa[b][k];
        }
    //退出循环时，a or b 再跳一下就是a ，b的最近公共祖先
    return fa[a][0];
}

int main()
{
    memset(h,-1,sizeof h);

    scanf("%d",&n);

    int root = 0;
    for(int i = 0; i < n; i++)
    {
        int a,b;
        scanf("%d %d",&a,&b);
        if(b == -1) root = a;
        else add(a,b) , add(b,a);
    }

    bfs(root);

    scanf("%d",&m);

    for(int i = 0; i < m; i++)
    {
        int a,b;
        scanf("%d %d",&a,&b);
        int p = lca(a,b);
        if(p == a)  puts("1");
        else if(p == b) puts("2");
        else puts("0");
    }

    return 0;
}
```

Tarjan离线LCA