The Code Library of xiaohai

Version 2.0

Volume 2 (Part IV, V)

xiaohai

Lingxiao Ma *Yi Li* *Lanjun Duan* *Anran Li*

College of Imformation Science and Technology

Beijing Normal University

acm International Collegiate Programming Contest

2015 年 1 月 29 日

Contents

Part IV

Algorithm

Chapter 15

Graph Theory

§ 15.1 Theorm

- 将一个树连成双连通分量至少需要的边= (叶子节点数+1)/2
- n顶点k条边的图至少有$n-k$个连通分量
- 如果一个图有一条不是环的边，则它至少有2个顶点不是割点
- 一个图是二分图是不存在奇环的充要条件，用交叉染色法判断是不是二分图
- n个顶点，则有n^2个有序对，有2^{n^2}个简单有向图，有$3^{\binom{n}{2}}$个定向图(简单图的定向)，有$2^{\binom{n}{2}}$个竞赛图(完全图的定向)
- //

§ 15.2 图数据结构

15.2.1 链式前向星

```
/**
*链式前向星
*/
const int maxn=0;
const int maxm=0;
struct Edge
{
    int u,v;
    int w;
    int next;
}edge[maxm];
int head[maxn],edgeNum;
void addSubEdge(int u,int v,int w)
{
    edge[edgeNum].u=u;
    edge[edgeNum].v=v;
```

```
    edge[edgeNum].w=w;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void addEdge(int u,int v,int w)
{
    addSubEdge(u,v,w);
    addSubEdge(v,u,w);
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
/*扫描*/
for(int i=head[u];i!=-1;i=edge[i].next)
{
    //内容
}
```

§ 15.3 搜索

15.3.1 Dancing Links

精确覆盖

精确覆盖 在一个全集X中若干子集的集合为S，精确覆盖是指，S的子集$S*$，满足X中的每一个元素在$S*$中**恰好**出现一次。

精确覆盖DLX模板

```
const int maxnode = 100010;
const int maxr = 1010;
const int maxc = 1010;
struct DLX
{
    int n, m, size;
    int U[maxnode], D[maxnode], R[maxnode], L[maxnode], Row[maxnode], Col[maxnode];
    int H[maxc], S[maxr];
    int ansd, ans[maxc];
    void init(int _n, int _m)
    {
        n = _n;
        m = _m;
        for (int i = 0; i <= m; i++)
        {
```

```
            S[i] = 0;
            U[i] = D[i] = i;
            L[i] = i - 1;
            R[i] = i + 1;
        }
        R[m] = 0; L[0] = m;
        size = m;
        for (int i = 1; i <= n; i++)
            H[i] = -1;
    }
    void Link(int r, int c)
    {
        ++S[Col[++size] = c];
        Row[size] = r;
        D[size] = D[c];
        U[D[c]] = size;
        U[size] = c;
        D[c] = size;
        if (H[r] < 0)H[r] = L[size] = R[size] = size;
        else
        {
            R[size] = R[H[r]];
            L[R[H[r]]] = size;
            L[size] = H[r];
            R[H[r]] = size;
        }
    }
    void remove(int c)
    {
        L[R[c]] = L[c]; R[L[c]] = R[c];
        for (int i = D[c]; i != c; i = D[i])
            for (int j = R[i]; j != i; j = R[j])
            {
                U[D[j]] = U[j];
                D[U[j]] = D[j];
                --S[Col[j]];
            }
    }
    void resume(int c)
    {
        for (int i = U[c]; i != c; i = U[i])
            for (int j = L[i]; j != i; j = L[j])
                ++S[Col[U[D[j]] = D[U[j]] = j]];
        L[R[c]] = R[L[c]] = c;
    }
```

```
    //d为递归深度
    bool Dance(int d)
    {
        if (R[0] == 0)
        {
            ansd = d;
            return true;
        }
        int c = R[0];
        for (int i = R[0]; i != 0; i = R[i])
            if (S[i] < S[c])
                c = i;
        remove(c);
        for (int i = D[c]; i != c; i = D[i])
        {
            ans[d] = Row[i];
            for (int j = R[i]; j != i; j = R[j])remove(Col[j]);
            if (Dance(d + 1))return true;
            for (int j = L[i]; j != i; j = L[j])resume(Col[j]);
        }
        resume(c);
        return false;
    }
};
```

例：HUST 1017

```
/*
样例：HUST 1017
*/
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <set>
#include <map>
#include <string>
#include <math.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
const int maxnode = 100010;
const int MaxM = 1010;
const int MaxN = 1010;
```

```
struct DLX
{
    int n, m, size;
    int U[maxnode], D[maxnode], R[maxnode], L[maxnode], Row[maxnode], Col[maxnode];
    int H[MaxN], S[MaxM];
    int ansd, ans[MaxN];
    void init(int _n, int _m)
    {
        n = _n;
        m = _m;
        for (int i = 0; i <= m; i++)
        {
            S[i] = 0;
            U[i] = D[i] = i;
            L[i] = i - 1;
            R[i] = i + 1;
        }
        R[m] = 0; L[0] = m;
        size = m;
        for (int i = 1; i <= n; i++)
            H[i] = -1;
    }
    void Link(int r, int c)
    {
        ++S[Col[++size] = c];
        Row[size] = r;
        D[size] = D[c];
        U[D[c]] = size;
        U[size] = c;
        D[c] = size;
        if (H[r] < 0)H[r] = L[size] = R[size] = size;
        else
        {
            R[size] = R[H[r]];
            L[R[H[r]]] = size;
            L[size] = H[r];
            R[H[r]] = size;
        }
    }
    void remove(int c)
    {
        L[R[c]] = L[c]; R[L[c]] = R[c];
        for (int i = D[c]; i != c; i = D[i])
            for (int j = R[i]; j != i; j = R[j])
            {
```

```
                U[D[j]] = U[j];
                D[U[j]] = D[j];
                --S[Col[j]];
            }
    }
    void resume(int c)
    {
        for (int i = U[c]; i != c; i = U[i])
            for (int j = L[i]; j != i; j = L[j])
                ++S[Col[U[D[j]] = D[U[j]] = j]];
        L[R[c]] = R[L[c]] = c;
    }
    //d为递归深度
    bool Dance(int d)
    {
        if (R[0] == 0)
        {
            ansd = d;
            return true;
        }
        int c = R[0];
        for (int i = R[0]; i != 0; i = R[i])
            if (S[i] < S[c])
                c = i;
        remove(c);
        for (int i = D[c]; i != c; i = D[i])
        {
            ans[d] = Row[i];
            for (int j = R[i]; j != i; j = R[j])remove(Col[j]);
            if (Dance(d + 1))return true;
            for (int j = L[i]; j != i; j = L[j])resume(Col[j]);
        }
        resume(c);
        return false;
    }
};

DLX g;
int main()
{
    //freopen("in.txt","r",stdin);
    //freopen("out.txt","w",stdout);
    int n, m;
    while (scanf("%d%d", &n, &m) == 2)
    {
```

```
        g.init(n, m);
        for (int i = 1; i <= n; i++)
        {
            int num, j;
            scanf("%d", &num);
            while (num--)
            {
                scanf("%d", &j);
                g.Link(i, j);
            }
        }
        if (!g.Dance(0))printf("NO\n");
        else
        {
            printf("%d", g.ansd);
            for (int i = 0; i < g.ansd; i++)
                printf(" %d", g.ans[i]);
            printf("\n");
        }
    }
    return 0;
}
```

重复覆盖

重复覆盖 在一个全集X中若干子集的集合为S，重复覆盖是指，S的子集$S*$，满足X中的每一个元素在$S*$中**至少**出现一次。

重复覆盖DLX模板

```
/*
重复覆盖：DLX
输入：Link()
输出：ans, bool Dance(int k)
*/
const int maxnode = 360000;
const int maxc = 500;
const int maxr = 500;
const int inf = 0x3f3f3f3f;
struct DLX
{
    int L[maxnode], R[maxnode], D[maxnode], U[maxnode], C[maxnode];
    int S[maxc], H[maxr], size;
    int ans;
    ///不需要S域
    void Link(int r, int c)
```

```
{
    S[c]++; C[size] = c;
    U[size] = U[c]; D[U[c]] = size;
    D[size] = c; U[c] = size;
    if (H[r] == -1) H[r] = L[size] = R[size] = size;
    else
    {
        L[size] = L[H[r]]; R[L[H[r]]] = size;
        R[size] = H[r]; L[H[r]] = size;
    }
    size++;
}
void remove(int c)
{
    for (int i = D[c]; i != c; i = D[i])
        L[R[i]] = L[i], R[L[i]] = R[i];
}
void resume(int c)
{
    for (int i = U[c]; i != c; i = U[i])
        L[R[i]] = R[L[i]] = i;
}
int h() ///用精确覆盖去估算剪枝
{
    int ret = 0;
    bool vis[maxc];
    memset (vis, false, sizeof(vis));
    for (int i = R[0]; i; i = R[i])
    {
        if (vis[i])continue;
        ret++;
        vis[i] = true;
        for (int j = D[i]; j != i; j = D[j])
            for (int k = R[j]; k != j; k = R[k])
                vis[C[k]] = true;
    }
    return ret;
}
//根据具体问题选择限制搜索深度或直接求解。
bool Dance(int k)
{
    if (k + h() >= ans) return 0;
    if (!R[0])
    {
        if (k < ans)ans = k;
```

```
            return 1;
        }
        int c = R[0];
        for (int i = R[0]; i; i = R[i])
            if (S[i] < S[c])c = i;
        for (int i = D[c]; i != c; i = D[i])
        {
            remove(i);
            for (int j = R[i]; j != i; j = R[j])
                remove(j);
            Dance(k + 1);
            for (int j = L[i]; j != i; j = L[j])
                resume(j);
            resume(i);
        }
        return 0;
    }
    void initL(int x) ///col is 1~x,row start from 1
    {
        for (int i = 0; i <= x; ++i)
        {
            S[i] = 0;
            D[i] = U[i] = i;
            L[i + 1] = i; R[i] = i + 1;
        }///对列表头初始化
        R[x] = 0;
        size = x + 1; ///真正的元素从m+1开始
        memset (H, -1, sizeof(H));
        ///mark每个位置的名字
    }
};
```

例：POJ 1084 Square Destroyer

```
/*
1、题意：给你一个n*n(n<=5)的完全由火柴棍组成的正方形，已经去掉了一些火柴棍，问最少去掉多少根火柴棍使得所有1*1、2*2......n*n的正方形均被破坏掉？
2、方法：矩阵的一行代表一根火柴棍，矩阵的一列代表一个正方形
3、处理去掉的火柴棍：先计算不去掉火柴棍的矩阵，对去掉的火柴棍对应的正方形加标记并在DLX里面标记它们已经访问过，然后在添加link时忽略这些标记过的正方形
*/
// #pragma comment(linker, "/STACK:102400000,102400000")
#include <cstdio>
#include <iostream>
#include <cstring>
#include <string>
```

```
#include <cmath>
#include <set>
#include <list>
#include <map>
#include <iterator>
#include <cstdlib>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <functional>
using namespace std;
typedef long long LL;
#define pb push_back
const int maxn = 110;
const int inf = 0x3f3f3f3f;

const int maxnode = 360000;
const int maxc = 500;
const int maxr = 500;
// const int inf = 0x3f3f3f3f;
struct DLX
{
    int L[maxnode], R[maxnode], D[maxnode], U[maxnode], C[maxnode];
    int S[maxc], H[maxr], size;
    int ans;
    ///不需要S域
    void Link(int r, int c)
    {
        S[c]++; C[size] = c;
        U[size] = U[c]; D[U[c]] = size;
        D[size] = c; U[c] = size;
        if (H[r] == -1) H[r] = L[size] = R[size] = size;
        else
        {
            L[size] = L[H[r]]; R[L[H[r]]] = size;
            R[size] = H[r]; L[H[r]] = size;
        }
        size++;
    }
    void remove(int c)
    {
        for (int i = D[c]; i != c; i = D[i])
            L[R[i]] = L[i], R[L[i]] = R[i];
    }
```

```
void resume(int c)
{
    for (int i = U[c]; i != c; i = U[i])
        L[R[i]] = R[L[i]] = i;
}
int h() ///用精确覆盖去估算剪枝
{
    int ret = 0;
    bool vis[maxc];
    memset (vis, false, sizeof(vis));
    for (int i = R[0]; i; i = R[i])
    {
        if (vis[i])continue;
        ret++;
        vis[i] = true;
        for (int j = D[i]; j != i; j = D[j])
            for (int k = R[j]; k != j; k = R[k])
                vis[C[k]] = true;
    }
    return ret;
}
//根据具体问题选择限制搜索深度或直接求解。
bool Dance(int k)
{
    if (k + h() >= ans) return 0;
    if (!R[0])
    {
        if (k < ans)ans = k;
        return 1;
    }
    int c = R[0];
    for (int i = R[0]; i; i = R[i])
        if (S[i] < S[c])c = i;
    for (int i = D[c]; i != c; i = D[i])
    {
        remove(i);
        for (int j = R[i]; j != i; j = R[j])
            remove(j);
        Dance(k + 1);
        for (int j = L[i]; j != i; j = L[j])
            resume(j);
        resume(i);
    }
    return 0;
}
```

```
    void initL(int x) ///col is 1~x,row start from 1
    {
        for (int i = 0; i <= x; ++i)
        {
            S[i] = 0;
            D[i] = U[i] = i;
            L[i + 1] = i; R[i] = i + 1;
        }///对列表头初始化
        R[x] = 0;
        size = x + 1; ///真正的元素从m+1开始
        memset (H, -1, sizeof(H));
        ///mark每个位置的名字
    }
} dlx;

int kase;
int n;
vector<int> vec;
bool mtx[maxn][maxn];
int row, col;
bool vis[maxn];
void init()
{
    kase++;
    vec.clear();
    memset(mtx, 0, sizeof(mtx));
    memset(vis, 0, sizeof(vis));
}
void input()
{
    scanf("%d", &n);
    int k;
    scanf("%d", &k);
    while (k--)
    {
        int x;
        scanf("%d", &x);
        vec.pb(x);
    }
}
void debug()
{
    //
}
void calmtx()
```

```
{
    row = 2 * n * (n + 1);
    col = 0;
    for (int i = 1; i <= n; i++)
        col += i * i;

    int cnt = 1;
    for (int si = 1; si <= n; si++)
    {
        for (int i = 1; i <= n - si + 1; i++)
        {
            for (int j = 1; j <= n - si + 1; j++)
            {
                for (int k = 0; k < si; k++)
                {
                    mtx[(i - 1) * (2 * n + 1) + j + k][cnt] = 1;
                    mtx[(i - 1 + si) * (2 * n + 1) + j + k][cnt] = 1;
                    mtx[i * n + (i - 1) * (n + 1) + j + k * (2 * n + 1)][cnt] = 1;
                    mtx[i * n + (i - 1) * (n + 1) + j + k * (2 * n + 1) + si][cnt] = 1;
                }
                cnt++;
            }
        }
    }
}
void build()
{
    calmtx();

    dlx.initL(col);
    for (int i = 0; i < vec.size(); i++)
    {
        int x = vec[i];
        for (int j = 1; j <= col; j++)
            if (mtx[x][j] && !vis[j])
            {
                vis[j] = 1;
                dlx.R[dlx.L[j]] = dlx.R[j];
                dlx.L[dlx.R[j]] = dlx.L[j];
                dlx.R[j] = dlx.L[j] = 0;
            }
    }
    for (int i = 1; i <= row; i++)
    {
        for (int j = 1; j <= col; j++)
```

```
        {
            if (mtx[i][j] && !vis[j])
                dlx.Link(i, j);
        }
    }
}
void solve()
{
    build();
    dlx.ans = inf;
    dlx.Dance(0);
    printf("%d\n", dlx.ans);
}
void output()
{
    //
}
int main()
{
#ifdef xysmlx
    freopen("in.cpp", "r", stdin);
#endif

    kase = 0;
    int T;
    scanf("%d", &T);
    while (T--)
    {
        init();
        input();
        solve();
        output();
    }
    return 0;
}
```

DLX标记已经提前选取过的列

```
dlx.R[dlx.L[j]] = dlx.R[j];
dlx.L[dlx.R[j]] = dlx.L[j];
dlx.R[j] = dlx.L[j] = 0;
```

§ 15.4 基本图算法

15.4.1 判断是否存在奇环、是否是二分图：交叉染色法($O(V+E)$)

```
/**
*判断是否存在奇环、是否是二分图：交叉染色法($O(V+E)$)
*使用方法：找一个连通分量用一次(因为fflag[])
*输入：图(链式前向星),now,fflag[](是否在同一连通分量)
*输出：true(是奇环、不是二分图),false(是二分图、不是奇环)
*/
const int maxn=0;
bool fflag[maxn];
bool toColor(int now)
{
    queue<int> Q;
    int color[maxn];
    memset(color,0,sizeof(color));
    color[now]=1;
    Q.push(now);
    while(!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for(int i=head[u]; i!=-1; i=edge[i].next)
        {
            int v=edge[i].v;
            if(!fflag[v]) continue;/* 不是同一连通分量*/
            if(!color[v])
            {
                color[v]=-color[u];
                Q.push(v);
            }
            else if(color[v] == color[u]) return true;
        }
    }
    return false;
}
```

15.4.2 2-SAT 问题($O(V+E)$)

理论

建图 把所有的输入都转化为$(x_1||y_1)\&\&(x_2||y_2)\&\&\cdots\&\&(x_i||y_i)\cdots$的形式：那么$(xi||yi)$ 就可以转化为$!xi->yi$,和$!yi->xi$两条边。

具体转化过程：$a\&\&b=(a||b)\&\&(!a||b)\&\&(!b||a)$

1. $a\ AND\ b=1$： 这个等价于$(a||b)\&\&(!a||b)\&\&(!b||a)$,于是在图中增加六条边$!a->$

b，$!b->a$，$a->b$，$!b->!a$，$b->a$，$!a->!b$

2. $a\ AND\ b=0$：这个等价于$!a||!b$,于是在原图中增加两条边$a->!b$，$b->!a$

3. $a\ OR\ b=0$：这个等价于$(!a||!b)\&\&(!a||b)\&\&(!b||a)$,于是在图中增加六条边$a->!b$，$b->!a$，$a->b$，$!b->!a$，$b->a$，$!a->!b$

4. $a\ OR\ b=1$：这个等价于$a||b$,于是在图中增加两条边$!a->b$，$!b->a$

5. $a\ XOR\ b=0$：这个等价于$(a||!b)\&\&(!a||b)$,于是在图中增加四条边$!a->!b$，$b->a$，$a->b$，$!b->!a$

6. $a\ XOR\ b1$：这个等价于$(a||b)\&\&(!a||!b)$,于是在图中增加四条边$!a->b$，$!b->a$，$a->!b$，$b->!a$

7. $a->b$：这个等价于$(a->b)\&\&(!b->!a)$,于是在图中增加两条边$a->b$，$!b->!a$

8. $a=0$：这个等价于加边$!a->a$

9. $a=1$：这个等价于加边$a->!a$

可行性判定 检查所有的变量a：!a和a不能够在同一个连通分量中,否则无解.

构造解 强连通分量缩点,在DAG图中按照逆拓扑序,依次选择,每一次选择,同时把和被选择点矛盾矛盾的点及其连通分量标记为不选,直到所有的点都作出了选择.

实现

Modified Edition of LRJ

```
/**
*2-SAT模板，Modified Edition of LRJ 按字典序排列结果
*输入：按照法则添加边(参数为2*i或者2*i+1)
*运行：先init(n),再add(),再solve()
*注意：add(2*i,2*j)才行
*输出：mark[](1表示选中),solve()(是否有解)
*/
const int maxn = 0;
struct TwoSAT
{
    int n;
    vector<int> G[maxn*2];
    bool mark[maxn*2];
    int S[maxn*2], c;

    bool dfs(int x)
    {
        if (mark[x^1]) return false;
        if (mark[x]) return true;
        mark[x] = true;
```

```
    S[c++] = x;
    for (int i = 0; i < G[x].size(); i++)
        if (!dfs(G[x][i])) return false;
    return true;
}

void init(int n)
{
    this->n = n;
    for (int i = 0; i < n*2; i++) G[i].clear();
    memset(mark, 0, sizeof(mark));
}

/// x AND y = 1
void add_and_one(int x,int y)
{
    G[x^1].push_back(y);
    G[y^1].push_back(x);
    G[x].push_back(y);
    G[y^1].push_back(x^1);
    G[y].push_back(x);
    G[x^1].push_back(y^1);
}

/// x AND y = 0
void add_and_zero(int x,int y)
{
    G[x].push_back(y^1);
    G[y].push_back(x^1);
}

/// x OR y = 1
void add_or_one(int x,int y)
{
    G[x^1].push_back(y);
    G[y^1].push_back(x);
}

/// x OR y = 0
void add_or_zero(int x,int y)
{
    G[x].push_back(y^1);
    G[y].push_back(x^1);
    G[x].push_back(y);
    G[y^1].push_back(x^1);
```

```
        G[x^1].push_back(y^1);
        G[y].push_back(x);
    }

    /// x XOR y = 1
    void add_xor_one(int x,int y)
    {
        G[x^1].push_back(y);
        G[y^1].push_back(x);
        G[x].push_back(y^1);
        G[y].push_back(x^1);
    }

    /// x XOR y = 0
    void add_xor_zero(int x,int y)
    {
        G[x^1].push_back(y^1);
        G[y].push_back(x);
        G[x].push_back(y);
        G[y^1].push_back(x^1);
    }

    /// x -> y
    void add_to(int x,int y)
    {
        G[x].push_back(y);
        G[y^1].push_back(x^1);
    }

    bool solve()
    {
        for(int i = 0; i < n*2; i += 2)
            if(!mark[i] && !mark[i+1])
            {
                c = 0;
                if(!dfs(i))
                {
                    while(c > 0) mark[S[--c]] = false;
                    if(!dfs(i+1)) return false;
                }
            }
        return true;
    }
};
```

链式前向星写法

```
/**
*2-SAT模板: 按字典序排列结果
*输入: 按照法则添加边(参数为2*i或者2*i+1)
*运行: 先init(n),再add(),再solve()
*注意: add(2*i,2*j)才行
*输出: vis[](1表示选中),solve()(是否有解)
*/
const int maxn = 0;
const int maxm = 0;
struct Edge
{
    int u, v;
    int next;
};
struct TwoSAT
{
    int n, en;
    Edge edge[maxm];
    int head[maxn];
    int vis[maxn], S[maxn];
    int cnt;

    void init(int _n = 0)
    {
        n = _n;
        memset(head, -1, sizeof(head));
        en = 0;
        memset(vis, 0, sizeof(vis));
    }

    void addse(int u, int v)
    {
        edge[en].u = u;
        edge[en].v = v;
        edge[en].next = head[u];
        head[u] = en++;
    }

    bool dfs(int u)
    {
        if (vis[u ^ 1])return 0;
        if (vis[u])return 1;
        vis[u] = 1;
        S[cnt++] = u;
```

```
    for (int i = head[u]; i != -1; i = edge[i].next)
    {
        if (!dfs(edge[i].v))return 0;
    }
    return 1;
}

bool solve()
{
    for (int i = 0; i < 2 * n; i += 2)
    {
        if (vis[i] || vis[i ^ 1])continue;
        cnt = 0;
        if (!dfs(i))
        {
            while (cnt)vis[S[--cnt]] = 0;
            if (!dfs(i ^ 1))return 0;
        }
    }
    return 1;
}

/// x AND y = 1
void add_and_one(int x, int y)
{
    addse(x ^ 1, y);
    addse(y ^ 1, x);
    addse(x, y);
    addse(y ^ 1, x ^ 1);
    addse(y, x);
    addse(x ^ 1, y ^ 1);
}

/// x AND y = 0
void add_and_zero(int x, int y)
{
    addse(x, y ^ 1);
    addse(y, x ^ 1);
}

/// x OR y = 1
void add_or_one(int x, int y)
{
    addse(x ^ 1, y);
    addse(y ^ 1, x);
```

```
    }

    /// x OR y = 0
    void add_or_zero(int x, int y)
    {
        addse(x, y ^ 1);
        addse(y, x ^ 1);
        addse(x, y);
        addse(y ^ 1, x ^ 1);
        addse(x ^ 1, y ^ 1);
        addse(y, x);
    }

    /// x XOR y = 1
    void add_xor_one(int x, int y)
    {
        addse(x ^ 1, y);
        addse(y ^ 1, x);
        addse(x, y ^ 1);
        addse(y, x ^ 1);
    }

    /// x XOR y = 0
    void add_xor_zero(int x, int y)
    {
        addse(x ^ 1, y ^ 1);
        addse(y, x);
        addse(x, y);
        addse(y ^ 1, x ^ 1);
    }

    /// x -> y
    void add_to(int x, int y)
    {
        addse(x, y);
        addse(y ^ 1, x ^ 1);
    }
};
```

按字典序排列结果的2-sat

```
/**
*2-SAT模板: 按字典序排列结果
*输入: 按照法则添加边(参数为2*i或者2*i+1)
*运行: 先init(n),再add(),再solve()
*注意: add(2*i,2*j)才行
```

```
*输出: color[](R表示选中),solve()/solvable()(是否有解)
*/
const int maxn = 0;
struct TwoSAT
{
    char color[maxn];//染色
    bool visit[maxn];
    queue<int>q1, q2;
    //vector建图方法很妙
    vector<vector<int> >adj; //原图    //中间一定要加空格把两个'>' 隔开
    vector<vector<int> >radj;//逆图
    vector<vector<int> >dag;//缩点后的逆向DAadj图
    int id[maxn], order[maxn], ind[maxn]; //强连通分量, 访问顺序, 入度
    int n, cnt;

    void init(int _n = 0)
    {
        n = _n;
        adj.assign(2 * n, vector<int>());
        radj.assign(2 * n, vector<int>());
    }

    void dfs(int u)
    {
        visit[u] = true;
        int i, len = adj[u].size();
        for (i = 0; i < len; i++)
            if (!visit[adj[u][i]])
                dfs(adj[u][i]);
        order[cnt++] = u;
    }

    void rdfs(int u)
    {
        visit[u] = true;
        id[u] = cnt;
        int i, len = radj[u].size();
        for (i = 0; i < len; i++)
            if (!visit[radj[u][i]])
                rdfs(radj[u][i]);
    }

    void korasaju()
    {
        int i;
```

```
    memset(visit, false, sizeof(visit));
    for (cnt = 0, i = 0; i < 2 * n; i++)
        if (!visit[i])
            dfs(i);
    memset(id, 0, sizeof(id));
    memset(visit, false, sizeof(visit));
    for (cnt = 0, i = 2 * n - 1; i >= 0; i--)
        if (!visit[order[i]])
        {
            cnt++;//这个一定要放前面来
            rdfs(order[i]);
        }
}

bool solvable()
{
    korasaju();
    for (int i = 0; i < n; i++)
        if (id[2 * i] == id[2 * i + 1])
            return false;
    return true;
}

void topsort()
{
    int i, j, len, now, p, pid;
    while (!q1.empty())
    {
        now = q1.front();
        q1.pop();
        if (color[now] != 0)continue;
        color[now] = 'R';
        ind[now] = -1;
        for (i = 0; i < 2 * n; i++)
        {
            if (id[i] == now)
            {
                //p=(i%2)?i+1:i-1;//点的编号从0开始以后这一定要修改
                p = i ^ 1;
                pid = id[p];
                q2.push(pid);
                while (!q2.empty())
                {
                    pid = q2.front();
                    q2.pop();
```

```
                    if (color[pid] == 'B')continue;
                    color[pid] = 'B';
                    len = dag[pid].size();
                    for (j = 0; j < len; j++)
                        q2.push(dag[pid][j]);
                }
            }
        }
        len = dag[now].size();
        for (i = 0; i < len; i++)
        {
            ind[dag[now][i]]--;
            if (ind[dag[now][i]] == 0)
                q1.push(dag[now][i]);
        }
    }
}

bool solve()
{
    if (!solvable()) return false;
    dag.assign(cnt + 1, vector<int>());
    memset(ind, 0, sizeof(ind));
    memset(color, 0, sizeof(color));
    for (int i = 0; i < 2 * n; i++)
    {
        for (int j = 0; j < adj[i].size(); j++)
            if (id[i] != id[adj[i][j]])
            {
                dag[id[adj[i][j]]].push_back(id[i]);
                ind[id[i]]++;
            }
    }
    for (int i = 1; i <= cnt; i++)
        if (ind[i] == 0) q1.push(i);
    topsort();
    return true;
}

/// x AND y = 1
void add_and_one(int x, int y)
{
    adj[x ^ 1].push_back(y);
    adj[y ^ 1].push_back(x);
    adj[x].push_back(y);
```

```
    adj[y ^ 1].push_back(x ^ 1);
    adj[y].push_back(x);
    adj[x ^ 1].push_back(y ^ 1);

    radj[y].push_back(x ^ 1);
    radj[x].push_back(y ^ 1);
    radj[y].push_back(x);
    radj[x ^ 1].push_back(y ^ 1);
    radj[x].push_back(y);
    radj[y ^ 1].push_back(x ^ 1);
}

/// x AND y = 0
void add_and_zero(int x, int y)
{
    adj[x].push_back(y ^ 1);
    adj[y].push_back(x ^ 1);

    radj[y ^ 1].push_back(x);
    radj[x ^ 1].push_back(y);
}

/// x OR y = 1
void add_or_one(int x, int y)
{
    adj[x ^ 1].push_back(y);
    adj[y ^ 1].push_back(x);

    radj[y].push_back(x ^ 1);
    radj[x].push_back(y ^ 1);
}

/// x OR y = 0
void add_or_zero(int x, int y)
{
    adj[x].push_back(y ^ 1);
    adj[y].push_back(x ^ 1);
    adj[x].push_back(y);
    adj[y ^ 1].push_back(x ^ 1);
    adj[x ^ 1].push_back(y ^ 1);
    adj[y].push_back(x);

    radj[y ^ 1].push_back(x);
    radj[x ^ 1].push_back(y);
    radj[y].push_back(x);
```

```
        radj[x ^ 1].push_back(y ^ 1);
        radj[y ^ 1].push_back(x ^ 1);
        radj[x].push_back(y);
    }

    /// x XOR y = 1
    void add_xor_one(int x, int y)
    {
        adj[x ^ 1].push_back(y);
        adj[y ^ 1].push_back(x);
        adj[x].push_back(y ^ 1);
        adj[y].push_back(x ^ 1);

        radj[y].push_back(x ^ 1);
        radj[x].push_back(y ^ 1);
        radj[y ^ 1].push_back(x);
        radj[x ^ 1].push_back(y);
    }

    /// x XOR y = 0
    void add_xor_zero(int x, int y)
    {
        adj[x ^ 1].push_back(y ^ 1);
        adj[y].push_back(x);
        adj[x].push_back(y);
        adj[y ^ 1].push_back(x ^ 1);

        radj[y ^ 1].push_back(x ^ 1);
        radj[x].push_back(y);
        radj[y].push_back(x);
        radj[x ^ 1].push_back(y ^ 1);
    }

    /// x -> y
    void add_to(int x, int y)
    {
        adj[x].push_back(y);
        adj[y ^ 1].push_back(x ^ 1);

        radj[y].push_back(x);
        radj[x ^ 1].push_back(y ^ 1);
    }
} sat;
```

15.4.3 Euler 路径(未知复杂度)

判定

无向图

- G有欧拉通路的充分必要条件为：G 连通，G中只有两个奇度顶点(它们分别是欧拉通路的两个端点)。
- G有欧拉回路(G为欧拉图)：G连通，G中均为偶度顶点。

有向图

- D有欧拉通路：D连通，除两个顶点外，其余顶点的入度均等于出度，这两个特殊的顶点中，一个顶点的入度比出度大1，另一个顶点的入度比出度小1。
- D有欧拉回路(D为欧拉图)：D连通，D中所有顶点的入度等于出度。

混合图(有向+无向)

1. 把该图的无向边随便定向，计算每个点的入度和出度。如果有某个点出入度之差为奇数，那么肯定不存在欧拉回路。因为欧拉回路要求每点入度= 出度，也就是总度数为偶数，存在奇数度点必不能有欧拉回路。
2. 现在每个点入度和出度之差均为偶数。将这个偶数除以2，得x。即是说，对于每一个点，只要将x条边反向（入¿出就是变入，出¿入就是变出），就能保证出= 入。如果每个点都是出= 入，那么很明显，该图就存在欧拉回路。
3. 构造网络流模型。有向边不能改变方向，直接删掉。开始已定向的无向边，定的是什么向，就把网络构建成什么样，边长容量上限1。另新建s和t。对于入> 出的点u，连接边(u, t)、容量为x，对于出> 入的点v，连接边(s, v)，容量为x （注意对不同的点x 不同。当初由于不小心，在这里错了好几次）。之后，察看是否有满流的分配。有就是能有欧拉回路，没有就是没有。查看流值分配，将所有流量非0（上限是1，流值不是0就是1）的边反向，就能得到每点入度= 出度的欧拉图。

有向图

非递归

```
/**
*Euler 路径(未知复杂度)
*首先计算出每个点的度数degree，然后调用euler函数，返回值为欧拉路径长度
*输入：n,g[][](邻接矩阵),deg[]
*输出：euler(),在相应地方处理
*/
const int maxn=0;
int n;
int deg[maxn];
int g[maxn][maxn];
void euler()
{
```

```
    for(int i=0; i<n; i++)
    {
        if(deg[i])
        {
            int u=i;
            while(1)
            {
                for(int j=0; j<n; j++)
                {
                    if(g[u][j]&&g[j][u])
                    {
                        g[j][u]=0;/// 欧拉路径的边，在这里处理
                        deg[u]--,deg[i]--;
                        u=j;
                        break;
                    }
                }
                if(u==i) break;
            }
        }
    }
}
```

递归(不建议用)

```
/**
*Euler 路径(未知复杂度)
*首先计算出每个点的度数degree，然后调用euler函数，返回值为欧拉路径长度
*输入: n,graph[][](邻接矩阵)
*输出: euler(int n)欧拉路径长度
*/
const int maxn=0;//顶点
const int maxm=0;//边
int graph[maxn][maxn];
int degree[maxn],n;
int ans[maxm];

void dfs(int k,int& top)
{
    int i;
    for(i=1; i<=500; ++i)
    {
        if(graph[k][i]>0)
        {
            --graph[k][i];
            --graph[i][k];
```

```
            dfs(i,top);
        }
    }
    ans[top++]=k;
}
int euler(int n)
{
    int ret=0;
    for(int i=1; i<=500; ++i)
    {
        if(degree[i]&1)
        {
            dfs(i,ret);
            return ret;
        }
    }
    dfs(1,ret);
    return ret;
}
```

混合图(POJ 1637)

```
#include <cstdio>
#include <cstring>
#include <queue>
#include <algorithm>
using namespace std;
typedef long long LL;
const int maxn = 2010;
const int maxm = 4010;
const int inf = 0x3f3f3f3f;
struct Edge
{
    int u, v;
    int cap, flow;
    int next;
} edge[maxm];
int head[maxn], edgeNum; //需初始化
int n, m, d[maxn], cur[maxn];
int st, ed;
bool vis[maxn];
void addSubEdge(int u, int v, int cap, int flow)
{
    edge[edgeNum].u = u;
    edge[edgeNum].v = v;
```

```
    edge[edgeNum].cap = cap;
    edge[edgeNum].flow = flow;
    edge[edgeNum].next = head[u];
    head[u] = edgeNum++;
    cur[u] = head[u];
}
void addEdge(int u, int v, int cap)
{
    addSubEdge(u, v, cap, 0);
    addSubEdge(v, u, 0, 0); //注意加反向0 边
}
bool BFS()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    Q.push(st);
    d[st] = 0;
    vis[st] = 1;
    while (!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0 && !vis[v])
            {
                vis[v] = 1;
                Q.push(v);
                d[v] = d[u] + 1;
                if (v == ed) return 1;
            }
        }
    }
    return false;
}
int Aug(int u, int a)
{
    if (u == ed) return a;
    int aug = 0, delta;
    for (int &i = cur[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].v;
        int w = edge[i].cap - edge[i].flow;
```

```
        if (w > 0 && d[v] == d[u] + 1)
        {
            delta = Aug(v, min(a, w));
            if (delta)
            {
                edge[i].flow += delta;
                edge[i ^ 1].flow -= delta;
                aug += delta;
                if (!(a -= delta)) break;
            }
        }
    }
    if (!aug) d[u] = -1;
    return aug;
}
int Dinic(int NdFlow)
{
    int flow = 0;
    while (BFS())
    {
        memcpy(cur, head, sizeof(int) * (n + 1));
        flow += Aug(st, inf);
        /*如果超过指定流量就return 掉*/
        if (NdFlow == inf) continue;
        if (flow > NdFlow) break;
    }
    return flow;
}

int in[maxn], out[maxn], a[maxn], b[maxn], c[maxn];
int N, M;
void init()
{
    memset(in, 0, sizeof(in));
    memset(out, 0, sizeof(out));
    memset(head, -1, sizeof(head));
    edgeNum = 0;
}
void input()
{
    scanf("%d%d", &N, &M);
    for (int i = 0; i < M; i++)
    {
        scanf("%d%d%d", &a[i], &b[i], &c[i]);
        out[a[i]]++;
```

```
        in[b[i]]++;
    }
}
void solve()
{
    int flag = 1;
    for (int i = 1; i <= N; i++)
        if ((out[i] - in[i] + 1000) & 1) flag = 0;
    if (!flag)
    {
        puts("impossible");
        return;
    }
    st = 0, ed = N + 1, n = N + 2;
    for (int i = 0; i < M; i++)
    {
        if (a[i] != b[i] && !c[i]) addEdge(a[i], b[i], 1);
    }
    int ans = 0;
    for (int i = 1; i <= N; i++)
    {
        if (out[i] > in[i])
        {
            addEdge(st, i, (out[i] - in[i]) / 2);
            ans += (out[i] - in[i]) / 2;
        }
        else if (out[i] < in[i]) addEdge(i, ed, (in[i] - out[i]) / 2);
    }
    if (Dinic(inf) == ans) puts("possible");
    else puts("impossible");
}
int main()
{
    int T;
    scanf("%d", &T);
    while (T--)
    {
        init();
        input();
        solve();
    }
    return 0;
}
```

15.4.4 Hamilton 回路

15.4.5 弦图判断

弦 连接环中不相邻的两个点的边

弦图 一个无向图称为弦图，当图中任意长度大于3的环都至少有一个弦。

```
/**
*弦图判断
*输入: g[][]置为邻接矩阵, n(从0到n-1)
*调用: mcs(n); peo(n);
*输出: peo(n)
*/
int g[maxn]maxn], order[maxn], inv[maxn], tag[maxn];
void mcs(int n)
{
    int i, j, k;
    memset(tag, 0, sizeof(tag));
    memset(order, -1, sizeof(order));
    for (i = n - 1; i >= 0; i--)   // vertex: 0 ~ n-1
    {
        for (j = 0; order[j] >= 0; j++) ;
        for (k = j + 1; k < n; k++)
            if (order[k] < 0 && tag[k] > tag[j]) j = k;
        order[j] = i, inv[i] = j;
        for (k = 0; k < n; k++) if (g[j][k]) tag[k]++;
    }
}
int peo(int n)
{
    int i, j, k, w, min;
    for (i = n - 2; i >= 0; i--)
    {
        j = inv[i], w = -1, min = n;
        for (k = 0; k < n; k++)
            if (g[j][k] && order[k] > order[j] &&
                    order[k] < min)
                min = order[k], w = k;
        if (w < 0) continue;
        for (k = 0; k < n; k++)
            if (g[j][k] && order[k] > order[w] && !g[k][w])
                return 0; // no
    }
    return 1; // yes
```

```
}
```

§ 15.5 无向图

BCC 双连通分量

15.5.1 割点($O(V+E)$)

```
/**
*割点(O(V+E))
*输入: 图(链式前向星),n(顶点数)(从0到n-1)
*输出: iscut[](标记是否为割点)
*/
const int maxn = 0;
const int maxm = 0;
struct Edge
{
    int v;
    int next;
} edge[maxm];
int head[maxn], edgeNum;
void addSubEdge(int u, int v)
{
    edge[edgeNum].v = v;
    edge[edgeNum].next = head[u];
    head[u] = edgeNum++;
}
void addEdge(int u, int v)
{
    addSubEdge(u, v);
    addSubEdge(v, u);
}
void init()
{
    memset(head, -1, sizeof(head));
    edgeNum = 0;
}
int n;
int low[maxn];
int mark[maxn];
int iscut[maxn];
int dfn;
int dfs(int k, int p)
{
    mark[k] = low[k] = dfn++;
```

```
    int son = 0, tag = 0;
    for (int i = head[k]; i != -1; i = edge[i].next)
    {
        int j = edge[i].v;
        if (j == p)continue;
        if (mark[j] == 0)
        {
            dfs(j, k);
            ++son;
            low[k] = min(low[k], low[j]);
            if (low[j] >= mark[k])tag = 1;
        }
        else low[k] = min(low[k], mark[j]);
    }
    if (p != -1 && tag || p == -1 && son > 1)iscut[k] = 1;
    return 0;
}
/*Dfs调用*/
void cutpoint()
{
    memset(mark, 0, sizeof(mark));
    memset(iscut, 0, sizeof(iscut));
    dfn = 1;
    for (int i = 0; i < n; ++i) if (mark[i] == 0)dfs(i, -1);
}
```

15.5.2 割边(桥)($O(V+E)$)

```
/**
*割边(桥)(不带判重)(O(V+E))
*判重方法：用map判重
*输入：图(链式前向星),n(顶点数)(从0到n-1)
*输出：按需输出
*/
const int maxn = 0;
const int maxm = 0;
struct Edge
{
    int v;
    int next;
} edge[maxm];
int head[maxn], edgeNum;
void addSubEdge(int u, int v)
{
    edge[edgeNum].v = v;
```

```
    edge[edgeNum].next = head[u];
    head[u] = edgeNum++;
}
void addEdge(int u, int v)
{
    addSubEdge(u, v);
    addSubEdge(v, u);
}
void init()
{
    memset(head, -1, sizeof(head));
    edgeNum = 0;
}
int n;
int low[maxn];
int mark[maxn];
int dfn;
void dfs(int k, int p)
{
    mark[k] = low[k] = dfn++;
    for (int i = head[k]; i != -1; i = edge[i].next)
    {
        int j = edge[i].v;
        if (j == p) continue;
        if (mark[j] == 0)
        {
            dfs(j, k);
            low[k] = min(low[k], low[j]);
            if (low[j] == mark[j])
            {
                //发现(k,j)为桥
            }
        }
        else low[k] = min(low[k], mark[j]);
    }
}
void bridge()
{
    memset(mark, 0, sizeof(mark));
    dfn = 1;
    for (int i = 0; i < n; i++) if (mark[i] == 0) dfs(i, -1);
}
```

15.5.3 双连通分量：Tarjan算法($O(V+E)$)

至少添加几条边,使无向图边成双连通图：将无向图求双连通分量(BCC),缩点后变成一棵树!根据缩点后的新图,统计度为1的结点(假设有a个)! 则$(a+1)/2$ 就是答案!

```
/**
*双连通分量：Tarjan算法($O(V+E)$)
*输入：图(链式前向星),n
*输出：bcc[](双连通分量)
*/
const int maxn = 0;
const int maxm = 0;
struct Edge
{
    int v;
    int next;
} edge[maxm];
int head[maxn], edgeNum;
void addSubEdge(int u, int v)
{
    edge[edgeNum].v = v;
    edge[edgeNum].next = head[u];
    head[u] = edgeNum++;
}
void addEdge(int u, int v)
{
    addSubEdge(u, v);
    addSubEdge(v, u);
}
void init()
{
    memset(head, -1, sizeof(head));
    edgeNum = 0;
}
int n;
int mark[maxn], low[maxn];  //mark初始化为0
int sstack[maxn], stop;     //top初始化为0
int dfn;                    //初始化为1
int bcc[maxn], bid;         //bid初始化为1
int dfs(int k, int p)
{
    int i, j;
    low[k] = mark[k] = dfn++;
    sstack[stop++] = k;
    for (i = head[k]; i != -1; i = edge[i].next)
    {
```

```
        j = edge[i].v;
        if (mark[j] == 0)
        {
            dfs(j, k);
            low[k] = min(low[k], low[j]);
        }
        else if (j != p) low[k] = min(low[k], mark[j]);
    }
    if (mark[k] > low[k])return 0;
    while (sstack[--stop] != k) // 导出一个双连通分量
    {
        bcc[sstack[stop]] = bid;
    }
    bcc[k] = bid;
    ++bid;
    return 0;
}
/*Dfs调用*/
void tarjan()
{
    memset(mark, 0, sizeof(mark));
    dfn = bid = 1;
    stop = 0;
    for (int i = 1; i <= n; ++i) if (mark[i] == 0) dfs(i, -1);
}
```

15.5.4 无向图的最小环($O(N^3)$)

修改版Floyd

```
const int MAX = 110;
int g[MAX][MAX];
int dist[MAX][MAX], s[MAX][MAX];
const int INF = 1000000000;
int path[MAX], ct;
int solve(int i, int j, int k)
{
    ct = 0;
    while (j != i)
    {
        path [ct++] = j;
        j = s[i][j];
    }
    path [ct++] = i, path [ct++] = k;
    return 0;
}
```

```
int min_circle(int graph[MAX][MAX], int n)
{
    memmove(dist, g, sizeof(dist));
    int ret = INF;
    for (int k = 0; k < n; ++k)
    {
        for (int i = 0; i < k; ++i)
        {
            if (g[k][i] == INF)continue;
            for (int j = i + 1; j < k; ++j)
                if (dist[i][j] < INF && g[k][j] < INF
                        && ret > dist[i][j] + g[k][i] + g[k][j])
                {
                    ret = dist[i][j] + g[k][i] + g[k][j];
                    solve(i, j, k);
                }
        }
        for (int i = 0; i < n; ++i)
        {
            if (dist[i][k] == INF)continue;
            for (int j = 0; j < n; ++j)
                if (dist[k][j] < INF && dist[i][j] > dist[i][k] + dist[k][j])
                {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    s[i][j] = s[k][j];
                }
        }
    }
    return ret;
}
```

15.5.5 无向图最小割：Stoer-Wagner算法($O(N^3)$)

求解最小割集普遍采用Stoer-Wagner算法：

1. min=MAXINT，固定一个顶点P
2. 从点P用类似prim的s算法扩展出“最大生成树”，记录最后扩展的顶点和最后扩展的边
3. 计算最后扩展到的顶点的切割值（即与此顶点相连的所有边权和），若比min小更新min
4. 合并最后扩展的那条边的两个端点为一个顶点（当然他们的边也要合并，这个好理解吧？）
5. 转到2，合并N-1次后结束
6. min即为所求，输出min

```
/*
```

```
prim本身复杂度是O(n^2)，合并n-1次，算法复杂度即为O(n^3)
如果在prim中加堆优化，复杂度会降为O((n^2)logn)
*/
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

const int NN = 505;
const int INF = 0x3fffffff;

int n, m, g[NN][NN], node[NN], dist[NN];
bool used[NN];
int mincut()
{
    int maxj, pre, ret = INF;
    for (int i = 0; i < n; i++) node[i] = i;
    while (n > 1)
    {
        memset(used, 0, sizeof(used));
        used[node[0]] = 1;
        maxj = 1;                           //记录'最远'的结点
        for (int i = 1; i < n; i++)
        {
            dist[node[i]] = g[node[0]][node[i]]; //固定定点P为node[0],这里初始化dist
            if (dist[node[i]] > dist[node[maxj]]) maxj = i;
        }
        pre = 0;
        for (int i = 1; i < n; i++)
        {
            if (i == n - 1)                 //生成树的最后一条边
            {
                ret = min(ret, dist[node[maxj]]); //更新最小割
                for (int k = 0; k < n; k++)     //合并pre和maxj两点
                {
                    g[node[k]][node[pre]] += g[node[k]][node[maxj]];
                    g[node[pre]][node[k]] = g[node[k]][node[pre]];
                }
                node[maxj] = node[--n];         //删掉maxj结点
            }
            used[node[maxj]] = 1;
            pre = maxj;
            maxj = -1;
            for (int j = 1; j < n; j++)
                if (!used[node[j]])
```

```
                {
                    dist[node[j]] += g[node[pre]][node[j]]; //更新到树的和距离
                    if (maxj == -1 || dist[node[maxj]] < dist[node[j]]) maxj = j;
                }
        }
    }
    return ret;
}

int main()
{
    while (scanf("%d%d", &n, &m) != -1)
    {
        memset(g, 0, sizeof(g));
        for (int i = 1; i <= m; i++)
        {
            int a, b, c;
            scanf("%d%d%d", &a, &b, &c);
            g[a][b] += c;
            g[b][a] += c;
        }
        printf("%d\n", mincut());
    }
    return 0;
}
```

§ 15.6 有向图

DAG 有向无环图

SCC 强连通分量

15.6.1 拓扑排序($O(V + E)$)

高效版，默认字典序最小，可改最大

```
/**
*拓扑排序：($O(V+E)$)
*高效版，默认字典序最小方案，可改最大(循环从n-1到0即可)
*输入：g[][](有向图), in[](每个点的入度), n
*/
const int maxn = 0;
int in[maxn];
int g[maxn][maxn];
void TopoOrder(int n)
{
```

```
    int i, top = -1;
    for (i = 0; i < n; ++i)
    {
        if (in[i] == 0) // 下标模拟堆栈
        {
            in[i] = top;
            top = i;
        }
    }
    for (i = 0; i < n; ++i)
    {
        if (top == -1)
        {
            printf("存在回路\n");
            return ;
        }
        else
        {
            int j = top;
            top = in[top];
            printf("%d", j);
            for (int k = 0; k < n; ++k)
            {
                if (g[j][k] && (--in[k]) == 0)
                {
                    in[k] = top;
                    top = k;
                }
            }
        }
    }
}
```

输出所有序列

```
/**
*拓扑排序：($O(V+E)$)
*输出所有序列
*POJ 1270 Following Orders
*给出变量列表和一组约束关系，按字典序输出所有满足关系的拓扑序列
*/
#include <iostream>
#include <stdio.h>
#include <string>
```

```
#include <cstring>
#include <algorithm>
using namespace std;
int node[30], num;  //num为变量的个数，node存储变量对应的整型值
int edge[30][30];  //edge[i][j]=1表示i<j。
int into[30];  //表示i的入度
//u表示此次选的是第u个变量，idx表示目前选了idx个变了，s是输出的结果字符串
void topo_dfs(int u, int idx, string s)
{
    if (u != -1)
        s += char(node[u] + 'a');
    if (idx == num)
    {
        cout << s << endl;
        return;
    }
    for (int i = 0; i < num; i++)
    {
        //选出入度为0的变量，将与它相连的点的入度-1。
        if (into[node[i]] == 0)
        {
            into[node[i]] = -1;
            for (int j = 0; j < 26; j++)
            {
                if (edge[node[i]][j])
                {
                    into[j]--;
                }
            }
            //一开始第一个参数传了node[i]。。。
            topo_dfs(i, idx + 1, s);
            //最后别忘了恢复
            into[node[i]] = 0;
            for (int j = 0; j < 26; j++)
            {
                if (edge[node[i]][j])
                {
                    into[j]++;
                }
            }
        }
    }
}
int main()
{
```

```
    char str1[100], str2[300];
    int u, v, len1, len2;
    while (gets(str1))
    {
        gets(str2);
        memset(edge, 0, sizeof(edge));
        memset(into, 0, sizeof(into));
        num = 0;
        len1 = strlen(str1);
        len2 = strlen(str2);
        for (int i = 0; i < len1; i += 2)
        {
            u = str1[i];
            node[num++] = u - 'a';
        }
        //这里排序，是为了之后dfs枚举的顺序按照字典顺序
        sort(node, node + num);

        for (int i = 0; i < len2; i += 4)
        {
            u = str2[i] - 'a';
            v = str2[i + 2] - 'a';
            edge[u][v] = 1;
            into[v]++;  //注意啦，这里into的下标是v，不是node数组中的索引!!! 一开始dfs中into就是用的索引，导致样例一直不过。。。
        }
        topo_dfs(-1, 0, "");
        puts("");
    }
    return 0;
}
```

其他版

```
/**
*拓扑排序
*输入：g[maxn][maxn](图，1~N)
*输出：topo[maxn](拓扑排序顺序)
*/
const int maxn=0;
int n,mk[maxn],topo[maxn],g[maxn][maxn],ps,topook;
void dfs(int u)
{
    if(mk[u]<0)
    {
```

```
        topook=0;
        return;
    }
    if(mk[u]>0) return;
    else mk[u]=-1;
    for(int v=1;topook&&v<=n;v++) if(g[u][v]) dfs(v);
    topo[ps--]=u;
    mk[u]=1;
}
void toposort()
{
    topook=1;
    ps=n;
    memset(mk,0,sizeof(mk));
    for(int i=1;topook&&i<=n;i++) if(!mk[i]) dfs(i);
}
```

15.6.2 强连通分量：Tarjan算法($O(V+E)$)

至少添加几条边,使有向图边成强连通图：将有向图求强连通分量(SCC),缩点后变成有向无环图(DAG)!根据缩点后的新图,分别统计入度为0的结点个数(假设有a个),出度为0的结点个数(假设有b个)! 则$max(a,b)$就是答案!

给定一个有向图,问有多少个点由任意顶点出发都能到达：将有向图求强连通分量(SCC),缩点后变成有向无环图(DAG)!统计新图中入度为0的结点个数，如果只有一个则输出该结点所代表的强连通分量下的顶点个数！ 否则无解，输出0!

给定一个有向图,求出sink点(sink点：如果v能够到的点,反过来可以到达v点) 并按升序输出：将有向图求强连通分量(SCC),缩点后变成有向无环图(DAG)!统计出度为0的结点个数,升序输出该结点所代表的强连通分量下的顶点

有向无环图(DAG)性质：

1. 任何DAG都有一个始点(我们假定入度为0的结点为始点，出度为0 的结点为终点)
2. 如果一个有向图的DAG只有一个始点，则由该始点出发可以到达DAG中的任意结点
3. 如果一个有向图的DAG只有一个终点，则由图中的任意结点都可以到达这个终点

```
/**
*有向图强连通分量：Tarjan算法($O(V+E)$)
*输入：图(从0到n-1)
*输出：sid[](强连通分量标号)
*/
const int maxn=0;
const int maxm=0;
struct Edge
{
    int v;
    int next;
```

```
}edge[maxm];
int head[maxn],edgeNum;
void addSubEdge(int u,int v)
{
    edge[edgeNum].v=v;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void addEdge(int u,int v)
{
    addSubEdge(u,v);
    addSubEdge(v,u);
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
int sid[maxn];
int mark[maxn],low[maxn];
int check[maxn];
int sstack[maxn],top;
int dfn,ssn;
int n,m;
void dfs(int k)
{
    int i,j;
    check[k]=1;
    low[k]=mark[k]=dfn++;
    sstack[top++]=k;
    for(int i=head[k]; i!=-1; i=edge[i].next)
    {
        int j=edge[i].v;
        if(mark[j]==0)
        {
            dfs(j);
            low[k]=min(low[k],low[j]);
        }
        else if(check[j])
            low[k]=min(low[k],mark[j]);
    }
    if(mark[k]==low[k])
    {
        while(sstack[--top]!=k)
        {
```

```
            check[sstack[top]]=0;
            sid[sstack[top]]=ssn;
        }
        sid[k]=ssn;
        check[k]=0;
        ++ssn;
    }
    return;
}
void tarjan()
{
    ssn=1;
    dfn=1;
    top=0;
    memset(check,0,sizeof(check));
    memset(mark,0,sizeof(mark));
    for(int i=0; i<n; ++i) if(mark[i]==0) dfs(i);
}
```

15.6.3 弱连通分量：Tarjan算法($O(V+E)$)

首先用Tarjan对原图进行缩点，对于缩点后的图，统计每个点的入度和出度，如果入度为0的点和出度为0的点都只存在1个，则判定这个图是弱连通图。

15.6.4 有向图的最小环($O(N^3)$)

直接由Floyd算法可以求得$ans = min(ans, edge[u][v] + dis[v][u])$

直接用flody算法求到到个点得最短路，最后取$i == j$中的最小值或最大值即为最小环和最大环的值

路径的求法：用一个$pre[i][j]$记录j前面的一个顶点，初始化为i，当出现需要更新的时候则将$pre[i][j] = pre[k][j]$; 若$i == j$的时候则表示找全了路径，最后将k点加入路径中

15.6.5 有向图最小权点基

1. 求强连通缩点
2. 求入度为0的点
3. 入度为0的点所在强连通分量的最小权值的点即为所求

§ 15.7 树

15.7.1 树的直径

两次DFS：第一次DFS任意起点，得到第二次DFS的起点；第二次DFS以第一次DFS 得出的最远的点为起点，找最长路径长度。

权值为1的树的直径

```
/**
*树的直径(权值为1)
*输入: 链式前向星(顶点标号从1到N)
*输出: diameter()(树的直径)
*/
const int maxn=0;
const int maxm=0;
int head[maxn],en;
Edge edge[maxm];
typedef pair<int,int> Result;
Result visit(int p,int u)
{
    Result r(0,u);
    for(int i=head[u];i!=-1;i=edge[i].next)
    {
        int v=edge[i].v;
        if(v==p) continue;
        Result t=visit(u,v);
        t.first+=1;
        if(r.first<t.first) r=t;
    }
    return r;
}
int diameter()
{
    Result r=visit(0,1);
    Result t=visit(0,r.second);
    return t.first;
}
```

任意权值的树的直径

```
/**
*树的直径(任意权值)
*输入: 链式前向星(顶点标号从1到N)
*输出: diameter()(树的直径)
*/
const int maxn=0;
const int maxm=0;
int head[maxn],en;
Edge edge[maxm];
typedef int Weight;
typedef pair<Weight,int> Result;
Result visit(int p,int u)
```

```
{
    Result r(0,u);
    for(int i=head[u];i!=-1;i=edge[i].next)
    {
        int v=edge[i].v;
        int w=edge[i].w;
        if(v==p) continue;
        Result t=visit(u,v);
        t.first+=w;
        if(r.first<t.first) r=t;
    }
    return r;
}
Weight diameter()
{
    Result r=visit(0,1);
    Result t=visit(0,r.second);
    return t.first;
}
```

15.7.2 LCA & RMQ

算法名称	针对问题	时间消耗	空间消耗
ST算法	一般RMQ问题	$O(N\log_2 N)$-$O(1)$	$O(N\log_2 N)$
Tarjan算法	LCA问题	$O(N\alpha(N) + Q)$	$O(N)$
±1RMQ算法	±1RMQ问题	$O(N)$-$O(1)$	$O(N)$

注：N表示问题规模，Q表示询问次数

dfs+ST在线算法

```
// POJ 1330
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <set>
#include <map>
#include <string>
#include <math.h>
#include <stdlib.h>
#include <time.h>
```

```
using namespace std;
/*
 * LCA  (POJ 1330)
 * 在线算法 DFS + ST
 */
//***************************
//ST算法，里面含有初始化init(n)和query(s,t)函数
//点的编号从1开始，从1到n.返回最小值的下标
//***************************
const int MAXN = 10010;
int rmq[2 * MAXN]; //rmq数组，就是欧拉序列对应的深度序列
struct ST
{
    int mm[2 * MAXN];
    int dp[2 * MAXN][20]; //最小值对应的下标
    void init(int n)
    {
        mm[0] = -1;
        for (int i = 1; i <= n; i++)
        {
            mm[i] = ((i & (i - 1)) == 0) ? mm[i - 1] + 1 : mm[i - 1];
            dp[i][0] = i;
        }
        for (int j = 1; j <= mm[n]; j++)
            for (int i = 1; i + (1 << j) - 1 <= n; i++)
                dp[i][j] = rmq[dp[i][j - 1]] < rmq[dp[i + (1 << (j - 1))][j - 1]] ?
                           dp[i][j - 1] : dp[i + (1 << (j - 1))][j - 1];
    }
    int query(int a, int b) //查询[a,b]之间最小值的下标
    {
        if (a > b)swap(a, b);
        int k = mm[b - a + 1];
        return rmq[dp[a][k]] <= rmq[dp[b - (1 << k) + 1][k]] ?
               dp[a][k] : dp[b - (1 << k) + 1][k];
    }
};
//边的结构体定义
struct Edge
{
    int to, next;
};
Edge edge[MAXN * 2];
int tot, head[MAXN];

int F[MAXN * 2]; //欧拉序列，就是dfs遍历的顺序，长度为2*n-1,下标从1开始
```

```
int P[MAXN];//P[i]表示点i在F中第一次出现的位置
int cnt;

ST st;
void init()
{
    tot = 0;
    memset(head, -1, sizeof(head));
}
void addedge(int u, int v) //加边，无向边需要加两次
{
    edge[tot].to = v;
    edge[tot].next = head[u];
    head[u] = tot++;
}
void dfs(int u, int pre, int dep)
{
    F[++cnt] = u;
    rmq[cnt] = dep;
    P[u] = cnt;
    for (int i = head[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].to;
        if (v == pre)continue;
        dfs(v, u, dep + 1);
        F[++cnt] = u;
        rmq[cnt] = dep;
    }
}
void LCA_init(int root, int node_num) //查询LCA前的初始化
{
    cnt = 0;
    dfs(root, root, 0);
    st.init(2 * node_num - 1);
}
int query_lca(int u, int v) //查询u,v的lca编号
{
    return F[st.query(P[u], P[v])];
}
bool flag[MAXN];
int main()
{
    //freopen("in.txt","r",stdin);
    //freopen("out.txt","w",stdout);
    int T;
```

```
    int N;
    int u, v;
    scanf("%d", &T);
    while (T--)
    {
        scanf("%d", &N);
        init();
        memset(flag, false, sizeof(flag));
        for (int i = 1; i < N; i++)
        {
            scanf("%d%d", &u, &v);
            addedge(u, v);
            addedge(v, u);
            flag[v] = true;
        }
        int root;
        for (int i = 1; i <= N; i++) // 找根
            if (!flag[i])
            {
                root = i;
                break;
            }
        LCA_init(root, N); // LCA初始化
        scanf("%d%d", &u, &v);
        printf("%d\n", query_lca(u, v)); // LCA查询
    }
    return 0;
}
```

LCA转化为RMQ的问题($O(N)$)

```
/* *****************************************
LCA转化为RMQ的问题
MAXN为最大结点数。ST的数组 和 F，edge要设置为2*MAXN

F是欧拉序列，rmq是深度序列，P是某点在F中第一次出现的下标
*******************************************/

struct LCA2RMQ
{
    int n;//结点个数
    Node edge[2 * MAXN]; //树的边,因为是建无向边，所以是两倍
    int tol;//边的计数
    int head[MAXN];//头结点
```

```
bool vis[MAXN];//访问标记
int F[2 * MAXN]; //F是欧拉序列，就是DFS遍历的顺序
int P[MAXN];//某点在F中第一次出现的位置
int cnt;

ST st;
void init(int n)//n为所以点的总个数，可以从0开始，也可以从1开始
{
    this->n = n;
    tol = 0;
    memset(head, -1, sizeof(head));
}
void addedge(int a, int b) //加边
{
    edge[tol].to = b;
    edge[tol].next = head[a];
    head[a] = tol++;
    edge[tol].to = a;
    edge[tol].next = head[b];
    head[b] = tol++;
}

int query(int a, int b) //传入两个节点，返回他们的LCA编号
{
    return F[st.query(P[a], P[b])];
}

void dfs(int a, int lev)
{
    vis[a] = true;
    ++cnt;//先加，保证F序列和rmq序列从1开始
    F[cnt] = a; //欧拉序列，编号从1开始，共2*n-1个元素
    rmq[cnt] = lev; //rmq数组是深度序列
    P[a] = cnt;
    for (int i = head[a]; i != -1; i = edge[i].next)
    {
        int v = edge[i].to;
        if (vis[v])continue;
        dfs(v, lev + 1);
        ++cnt;
        F[cnt] = a;
        rmq[cnt] = lev;
    }
}
```

```
    void solve(int root)
    {
        memset(vis, false, sizeof(vis));
        cnt = 0;
        dfs(root, 0);
        st.init(2 * n - 1);
    }
};
```

离线Tarjan算法($O(N + Q)$)

```
// POJ 1470
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <set>
#include <map>
#include <string>
#include <math.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
/*
 * POJ 1470
 * 给出一颗有向树，Q个查询
 * 输出查询结果中每个点出现次数
 */
/*
 * LCA离线算法，Tarjan
 * 复杂度O(n+Q);
 */
const int MAXN = 1010;
const int MAXQ = 500010;//查询数的最大值

//并查集部分
int F[MAXN];//需要初始化为-1
int find(int x)
{
    if (F[x] == -1)return x;
    return F[x] = find(F[x]);
}
void bing(int u, int v)
```

```
{
    int t1 = find(u);
    int t2 = find(v);
    if (t1 != t2)
        F[t1] = t2;
}
//************************
bool vis[MAXN];//访问标记
int ancestor[MAXN];//祖先
struct Edge
{
    int to, next;
} edge[MAXN * 2];
int head[MAXN], tot;
void addedge(int u, int v)
{
    edge[tot].to = v;
    edge[tot].next = head[u];
    head[u] = tot++;
}

struct Query
{
    int q, next;
    int index;//查询编号
} query[MAXQ * 2];
int answer[MAXQ];//存储最后的查询结果，下标0~Q-1
int h[MAXQ];
int tt;
int Q;

void add_query(int u, int v, int index) //u,v,第几组查询
{
    query[tt].q = v;
    query[tt].next = h[u];
    query[tt].index = index;
    h[u] = tt++;
    query[tt].q = u;
    query[tt].next = h[v];
    query[tt].index = index;
    h[v] = tt++;
}

void init()
{
```

```
    tot = 0;
    memset(head, -1, sizeof(head));
    tt = 0;
    memset(h, -1, sizeof(h));
    memset(vis, false, sizeof(vis));
    memset(F, -1, sizeof(F));
    memset(ancestor, 0, sizeof(ancestor));
}

void LCA(int u)
{
    ancestor[u] = u;
    vis[u] = true;
    for (int i = head[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].to;
        if (vis[v])continue;
        LCA(v);
        bing(u, v);
        ancestor[find(u)] = u;
    }
    for (int i = h[u]; i != -1; i = query[i].next)
    {
        int v = query[i].q;
        if (vis[v])
        {
            answer[query[i].index] = ancestor[find(v)];
        }
    }
}

bool flag[MAXN];
int Count_num[MAXN];
int main()
{
    //freopen("in.txt","r",stdin);
    //freopen("out.txt","w",stdout);
    int n;
    int u, v, k;
    while (scanf("%d", &n) == 1)
    {
        init();
        memset(flag, false, sizeof(flag));
        for (int i = 1; i <= n; i++)
        {
```

```
            scanf("%d:(%d)", &u, &k);
            while (k--)
            {
                scanf("%d", &v);
                flag[v] = true;
                addedge(u, v);
                addedge(v, u);
            }
        }
        scanf("%d", &Q);
        for (int i = 0; i < Q; i++)
        {
            char ch;
            cin >> ch;
            scanf("%d %d)", &u, &v);
            add_query(u, v, i); //增加一组查询
        }
        int root;
        for (int i = 1; i <= n; i++) //找根
            if (!flag[i])
            {
                root = i;
                break;
            }
        LCA(root);
        memset(Count_num, 0, sizeof(Count_num));
        for (int i = 0; i < Q; i++)
            Count_num[answer[i]]++;
        for (int i = 1; i <= n; i++)
            if (Count_num[i] > 0)
                printf("%d:%d\n", i, Count_num[i]);
    }
    return 0;
}
```

倍增算法，在线算法

```
// POJ 1330
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>
#include <set>
```

```
#include <map>
#include <string>
#include <math.h>
#include <stdlib.h>
#include <time.h>
using namespace std;
/*
 * POJ 1330
 * LCA 在线算法
 */
const int MAXN = 10010;
const int DEG = 20;

struct Edge
{
    int to, next;
} edge[MAXN * 2];
int head[MAXN], tot;
void addedge(int u, int v)
{
    edge[tot].to = v;
    edge[tot].next = head[u];
    head[u] = tot++;
}
void init()
{
    tot = 0;
    memset(head, -1, sizeof(head));
}
int fa[MAXN][DEG];//fa[i][j]表示结点i的第2^j个祖先
int deg[MAXN];//深度数组

void BFS(int root)
{
    queue<int>que;
    deg[root] = 0;
    fa[root][0] = root;
    que.push(root);
    while (!que.empty())
    {
        int tmp = que.front();
        que.pop();
        for (int i = 1; i < DEG; i++)
            fa[tmp][i] = fa[fa[tmp][i - 1]][i - 1];
        for (int i = head[tmp]; i != -1; i = edge[i].next)
```

```
        {
            int v = edge[i].to;
            if (v == fa[tmp][0])continue;
            deg[v] = deg[tmp] + 1;
            fa[v][0] = tmp;
            que.push(v);
        }

    }
}
int LCA(int u, int v)
{
    if (deg[u] > deg[v])swap(u, v);
    int hu = deg[u], hv = deg[v];
    int tu = u, tv = v;
    for (int det = hv - hu, i = 0; det ; det >>= 1, i++)
        if (det & 1)
            tv = fa[tv][i];
    if (tu == tv)return tu;
    for (int i = DEG - 1; i >= 0; i--)
    {
        if (fa[tu][i] == fa[tv][i])
            continue;
        tu = fa[tu][i];
        tv = fa[tv][i];
    }
    return fa[tu][0];
}
bool flag[MAXN];
int main()
{
    //freopen("in.txt","r",stdin);
    //freopen("out.txt","w",stdout);
    int T;
    int n;
    int u, v;
    scanf("%d", &T);
    while (T--)
    {
        scanf("%d", &n);
        init();
        memset(flag, false, sizeof(flag));
        for (int i = 1; i < n; i++)
        {
            scanf("%d%d", &u, &v);
```

```
            addedge(u, v);
            addedge(v, u);
            flag[v] = true;
        }
        int root;
        for (int i = 1; i <= n; i++)
            if (!flag[i])
            {
                root = i;
                break;
            }
        BFS(root);
        scanf("%d%d", &u, &v);
        printf("%d\n", LCA(u, v));
    }
    return 0;
}
```

15.7.3 最小斯坦纳树(Steiner Tree)($O(n3^k + cE2^k)$)

斯坦纳树(Steiner Tree) 使得指定集合中的点连通的树。

最小斯坦纳树 将指定点集合中的所有点连通，且边权总和最小的生成树。

最小斯坦纳树解法 可以用DP求解，$dp[i][state]$表示以i为根，指定集合中的点的连通状态为$state$的生成树的最小总权值。

- 第一重(枚举子树的形态)：$dp[i][state] = mindp[i][state], dp[i][subset1] + dp[i][subset2]$；枚举子集的技巧可以用$for(sub = (state - 1)\&state; sub; sub = (sub - 1)\&state)$。
- 第二重(按照边进行松弛)：$dp[i][state] = mindp[i][state], dp[j][state] + e[i][j]$

模板

```
/*
 *  Steiner Tree：求，使得指定K个点连通的生成树的最小总权值
 *  st[i] 表示顶点i的标记值，如果i是指定集合内第m(0<=m<K)个点，则st[i]=1<<m
 *  endSt=1<<K
 *  dptree[i][state] 表示以i为根，连通状态为state的生成树值
 *  输入：图(链式前向星)、st[i]、K
 */
struct Edge
{
    int u, v;
    int w;
    int next;
} edge[maxm];
```

```
int head[maxn];
int en;

int dptree[N][1 << K], st[N], endSt;
bool vis[N][1 << K];
queue<int> que;

int input()
{
    /*
    *    输入，并且返回指定集合元素个数K
    *    因为有时候元素个数需要通过输入数据处理出来，所以单独开个输入函数。
    */
}

void initSteinerTree()
{
    while (!que.empty()) que.pop();
    memset(dptree, -1, sizeof(dptree));
    memset(st, 0, sizeof(st));
    for (int i = 1; i <= n; i++)
        memset(vis[i], 0, sizeof(vis[i]));
    endSt = 1 << input();
    for (int i = 1; i <= n; i++)
        dptree[i][st[i]] = 0;
}

void update(int &a, int x)
{
    a = (a > x || a == -1) ? x : a;
}

void SPFA(int state)
{
    while (!que.empty())
    {
        int u = que.front();
        que.pop();
        vis[u][state] = false;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            if (dptree[v][st[v] | state] == -1 ||
                    dptree[v][st[v] | state] > dptree[u][state] + edge[i].w)
            {
```

```
                dptree[v][st[v] | state] = dptree[u][state] + edge[i].w;
                if (st[v] | state != state || vis[v][state])
                    continue; //只更新当前连通状态
                vis[v][state] = true;
                que.push(v);
            }
        }
    }
}

void steinerTree()
{
    for (int j = 1; j < endSt; j++)
    {
        for (int i = 1; i <= n; i++)
        {
            if (st[i] && (st[i]&j) == 0) continue;
            for (int sub = (j - 1)&j; sub; sub = (sub - 1)&j)
            {
                int x = st[i] | sub, y = st[i] | (j - sub);
                if (dptree[i][x] != -1 && dptree[i][y] != -1)
                    update(dptree[i][j], dptree[i][x] + dptree[i][y]);
            }
            if (dptree[i][j] != -1)
                que.push(i), vis[i][j] = true;
        }
        SPFA(j);
    }
}
```

例:

§ 15.8 最小生成树

15.8.1 Prim 算法($O(N^2)$)

```
/**
*最小生成树Prim 算法
*输入: 邻接矩阵mtx[MAXN][MAXN]
*输出: prim()(最小权值),path[](最小生成树的边)
*/
#include<cstdio>
#include<cstring>
#include<iostream>
```

```
#include<string>
#include<algorithm>
#include<functional>
#include<cmath>
#include<queue>
#include<stack>
#include<vector>
using namespace std;
const int MAXN=1010;//Max size of the problem
const double INF=1e30;//Infinity
const double EPS=1e-6;//Epsilon
double mtx[MAXN][MAXN];//Matrix of the graph
int clovtx[MAXN];
double lowwei[MAXN];
int n;//Number of the vertexes
int st;//Start vertex
struct EDGE//Edge of the graph
{
    int st;
    int ed;
    double wei;
    EDGE(int stx=0,int edx=0,double weight=0):st(stx),ed(edx),wei(weight) {}
    void setedge(int stx=0,int edx=0,double weight=0)
    {
        st=stx;
        ed=edx;
        wei=weight;
    }
    ~EDGE() {}
};
vector<EDGE> path;
void initial()//Initial the problem
{
    for(int i=0; i<MAXN; i++)
    {
        for(int j=0; j<MAXN; j++)
        {
            if(i==j) mtx[i][j]=0;
            else mtx[i][j]=INF;
        }
    }
    n=0;
    st=-1;
}
void input()//Input the data
```

```
{
    int ne;
    cout<<"Input the number of the vertexes: ";
    cin>>n;
    cout<<"Input the number of the edges: ";
    cin>>ne;
    cout<<"Input the edges(start, end, weight):"<<endl;
    for(int i=0; i<ne; i++)
    {
        int a,b;
        double c;
        cin>>a>>b>>c;
        mtx[a][b]=c;
        mtx[b][a]=c;
    }
    cout<<"Input the start vertex: ";
    cin>>st;
}
double prim()//Prim Algorithm
{
    path.clear();
    double sum=0;
    for(int i=1; i<=n; i++)
    {
        lowwei[i]=mtx[st][i];
        clovtx[i]=st;
    }
    lowwei[st]=0;
    clovtx[st]=-1;
    for(int i=1; i<n; i++)
    {
        double MinCost=INF;
        int v=-1;
        for(int j=1; j<=n; j++)
        {
            if(clovtx[j]!=-1&&lowwei[j]<MinCost)
            {
                MinCost=lowwei[j];
                v=j;
            }
        }
        if(v!=-1)
        {
            EDGE temp=EDGE(clovtx[v],v,lowwei[v]);
            path.push_back(temp);//Get the MST
```

```
            clovtx[v]=-1;
            sum+=lowwei[v];
            for(int j=1; j<=n; j++)
            {
                if(clovtx[j]!=-1&&mtx[v][j]<lowwei[j])
                {
                    lowwei[j]=mtx[v][j];
                    clovtx[j]=v;
                }
            }
        }
    }
    return sum;
}
void solve()
{
    double ans=prim();
    cout<<"The cost of the MST is: "<<ans<<endl;
    cout<<"The MST is:"<<endl;
    vector<EDGE>::iterator ite;
    for(ite=path.begin();ite!=path.end();ite++)//Output the MST
    {
        cout<<ite->st<<"---"<<ite->ed<<" "<<ite->wei<<endl;
    }
}
int main()
{
    initial();
    input();
    solve();
    return 0;
}
```

15.8.2 Kruskal 算法(稀疏图)($O(E \lg E)$)

```
/**
*最小生成树Kruskal 算法
*输入: edge[](边)
*输出: kruskal()(最小权值),path[](最小生成树的边)
*/
#include<cstdio>
#include<cstring>
#include<iostream>
#include<string>
#include<algorithm>
```

```
#include<functional>
#include<cmath>
#include<queue>
#include<stack>
#include<vector>
using namespace std;
const int MAXN=1010;//Max size of the problem
const double INF=1e30;//Infinity
const double EPS=1e-6;//Epsilon
struct EDGE//Edge of the graph
{
    int st;
    int ed;
    double wei;
    EDGE(int stx=0,int edx=0,double weight=0):st(stx),ed(edx),wei(weight) {}
    void setedge(int stx=0,int edx=0,double weight=0)
    {
        st=stx;
        ed=edx;
        wei=weight;
    }
    ~EDGE() {}
    bool operator<(const EDGE &temp) const
    {
        return wei<temp.wei;
    }
};
vector<EDGE> edge;//The set of edges
vector<EDGE> path;//Store the MST
int pnt[MAXN];//Parents vector
int n;//Number of the vertexes
void initial()//Initial the problem
{
    edge.clear();
    n=0;
    for(int i=0;i<MAXN;i++) pnt[i]=i;
}
void input()//Input the data
{
    int ne;
    cout<<"Input the number of the vertexes: ";
    cin>>n;
    cout<<"Input the number of the edges: ";
    cin>>ne;
    cout<<"Input the edges(start, end, weight):"<<endl;
```

```
    for(int i=0; i<ne; i++)
    {
        int a,b;
        double c;
        cin>>a>>b>>c;
        EDGE temp=EDGE(a,b,c);
        edge.push_back(temp);
    }
}
int UFind(int x)//Union-Find Algorithm
{
    if(x!=pnt[x]) pnt[x]=UFind(pnt[x]);
    return pnt[x];
}
double kruskal()//Kruskal Algorithm
{
    path.clear();
    double sum=0;
    sort(edge.begin(),edge.end());
    vector<EDGE>::iterator ite;
    for(ite=edge.begin();ite!=edge.end();ite++)
    {
        if(UFind(ite->st)!=UFind(ite->ed))
        {
            sum+=ite->wei;
            path.push_back(*ite);
            pnt[UFind(ite->ed)]=UFind(ite->st);
        }
    }
    return sum;
}
void solve()
{
    double ans=kruskal();
    cout<<"The cost of the MST is: "<<ans<<endl;
    cout<<"The MST is:"<<endl;
    vector<EDGE>::iterator ite;
    for(ite=path.begin();ite!=path.end();ite++)//Output the MST
    {
        cout<<ite->st<<"---"<<ite->ed<<" "<<ite->wei<<endl;
    }
}
int main()
{
    initial();
```

```
    input();
    solve();
    return 0;
}
```

15.8.3 增量最小生成树

增量最小生成树 从n个点的空图开始，依次加入m条带权边，每加入一条边，输出一次权值。

算法1($O(NM \lg N)$) 每生成一个最小生成树，将其余的所有非生成树的边删除。然后每次做最小生成树即可。

算法2($O(NM)$) 每次加入一条边后，图中恰好包含一个环，删除该回路上权值最大的边即可。每次$O(N)$搜索，复杂度$O(NM)$。

15.8.4 最小瓶颈路与最小瓶颈生成树

最小瓶颈生成树 给出加权无向图，求一颗生成树，使得最大边权最小。即为最小生成树。

最小瓶颈路 给定加权无向图的两个节点u,v，求从u到v的一条路径，使路径最长边最短。先计算最小生成树，最小瓶颈路即在最小生成树上。计算任意两对点的最小瓶颈路：$f(x,u) = max(f(x,v), w(u,v))$

15.8.5 次小生成树($O(V^2)$)

在prim算法的同时,计算出任意两点间在生成树路径上的最大边,这个计算的复杂度为$O(V^2)$.然后再枚举不在生成树上的边,做可行交换,复杂度为$O(V^2)$.总的时间复杂度为$O(V^2)$.

```
/**
*次小生成树(O(V^2))
*输入: n(点,从0到n-1),m(边数),graph[][](邻接矩阵)
*输出: tag(是否单一),
*/
const int maxn=0;
const int inf=0x3f3f3f3f;
int graph[maxn][maxn];
int dp[maxn][maxn];
int mark[maxn];
int s[maxn];
int d[maxn];
int n,m;
void lessMST()
{
    d[0]=0;
    s[0]=-1;
    mark[0]=1;
    for(int i=1; i<n; ++i)
```

```
{
    if(graph[i][0]==0) d[i]=inf;
    else d[i]=graph[i][0];
    s[i]=0;
}
int ans=0;
for(int i=1; i<n; ++i)
{
    int k=-1;
    for(int j=0; j<n; ++j)
    {
        if(mark[j]==1)continue;
        if(k==-1||d[j]<d[k]) k=j;
    }
    if(k==-1||d[k]==inf)break;
    ans+=d[k];
    for(int j=0; j<n; ++j)
    {
        if(mark[j]==0)continue;
        dp[j][k]=max(dp[j][s[k]],d[k]);
        dp[k][j]=dp[j][k];
    }
    mark[k]=1;
    for(int j=0; j<n; ++j)
    {
        if(mark[j]==1)continue;
        if(graph[k][j]&&graph[k][j]<d[j])
        {
            d[j]=graph[k][j];
            s[j]=k;
        }
    }
}
int tag=0;
for(int i=0; i<n; ++i)
{
    for(int j=0; j<n; ++j)
    {
        if(graph[i][j]==0)continue;
        if(s[i]==j||s[j]==i)continue;
        if(graph[i][j]==dp[i][j]) tag=1;
    }
}
if(tag) printf("Not Unique!\n");//不单一
else printf("%d\n",ans);// 单一
```

```
}
void input()
{
    scanf("%d%d",&n,&m);
    memset(graph,0,sizeof(graph));
    memset(mark,0,sizeof(mark));
    memset(dp,0,sizeof(dp));
    while(m--)
    {
        int i,j,k;
        scanf("%d%d%d",&i,&j,&k);
        --i,--j;//从0到n-1
        if(graph[i][j]==0) graph[i][j]=k;
        else graph[i][j]=min(graph[i][j],k);// 带重边
        graph[j][i]=graph[i][j];
    }
}
```

15.8.6 第k小生成树

15.8.7 最优比例生成树(未知复杂度)

Dinkelbach 版(优于二分)

```
/**
*最优比例生成树(未知复杂度)(优于二分)
*从1到n
*输入: n(顶点数),a[][](收益),b[][](费用)
*输出: dinkelbach()(最优比率)
*/
const int maxn=0;
const double INF=1e30;
const double eps=1e-6;// 控制精度
double a[maxn][maxn],b[maxn][maxn],mtx[maxn][maxn];
int clovtx[maxn];
double lowwei[maxn];
double prim()//Prim Algorithm
{
    double p=0,q=0;
    int st=1;
    double sum=0;
    for(int i=1; i<=n; i++)
    {
        lowwei[i]=mtx[st][i];
```

```
        clovtx[i]=st;
    }
    lowwei[st]=0;
    clovtx[st]=-1;
    for(int i=1; i<n; i++)
    {
        double MinCost=INF;
        int v=-1;
        for(int j=1; j<=n; j++)
        {
            if(clovtx[j]!=-1&&lowwei[j]<MinCost)
            {
                MinCost=lowwei[j];
                v=j;
            }
        }
        if(v!=-1)
        {
            p+=a[clovtx[v]][v];
            q+=b[clovtx[v]][v];
            clovtx[v]=-1;
            sum+=lowwei[v];
            for(int j=1; j<=n; j++)
            {
                if(clovtx[j]!=-1&&mtx[v][j]<lowwei[j])
                {
                    lowwei[j]=mtx[v][j];
                    clovtx[j]=v;
                }
            }
        }
    }
    return p/q;
}
double dinkelbach()
{
    double L=0.5;
    double ans;
    do
    {
        ans=L;
        for(int i=1; i<n; i++)
            for(int j=i+1; j<=n; j++)
                mtx[i][j]=mtx[j][i]=a[i][j]-L*b[i][j];
        for(int i=1; i<=n; i++) mtx[i][i]=INF;
```

```
        L=prim();
    }
    while(fabs(ans-L)>=eps);
    return ans;
}
```

二分版

```
/**
*最优比例生成树(未知复杂度)
*从0到n-1
*输入: n(顶点数),c[][](费用),w[][](收益)
*输出: opt_mst()(最优比率)
*/
const int maxn=0;
const double INF=1e30;
const double eps=1e-6;
int n;
double g[maxn][maxn],c[maxn][maxn],w[maxn][maxn];
double mst()
{
    double minD[maxn];
    int mark[maxn];
    double ans=0.0;
    memset(mark,0,sizeof(mark));
    for(int i=0; i<n; ++i)
        minD[i]=INF;
    minD[0]=0;
    for(int i=0; i<n; ++i)
        if(g[0][i]<minD[i])
            minD[i]=g[0][i];
    mark[0]=1;
    for(int i=1; i<n; ++i)
    {
        int k=-1;
        for(int j=0; j<n; ++j)
        {
            if(mark[j]==0)
            {
                if(k==-1||minD[j]<minD[k])
                    k=j;
            }
        }
        ans+=minD[k];
        mark[k]=1;
```

```
        for(int i=0; i<n; ++i)
        {
            if(mark[i]==0&&g[k][i]<minD[i])
                minD[i]=g[k][i];
        }
    }
    return ans;
}
double opt_mst()//二分法求解
{
    double low=0.0,high=100;
    while(fabs(low-high)>eps)
    {
        double mid=(low+high)/2;
        for(int i=0; i<n; ++i)
            for(int j=0; j<n; ++j)
                g[i][j]=c[i][j]-mid*w[i][j];
        double ans=mst();
        if(fabs(ans)<eps)
        {
            low=mid;
            break;
        }
        else if(ans>-eps) low=mid;
        else high=mid;
    }
    return low;
}
```

15.8.8 有向图最小树形图($O(VE)$)

```
/**
*有向图最小树形图(O(VE))
*输入: edge置为边表; res 置为0; cp[i] 置为i;N(顶点数,从0到n-1)
*调用: dirtree(root, nv, ne)
*输出: res是结果,cp[](记录父子节点);
*/
const int maxn=0;
const int maxm=0;
int res,dis[maxn],N;
int to[maxn],cp[maxn],tag[maxn],en;
struct Edge
{
    int u,v,next,w;
};
```

```
Edge edge[maxm];
int iroot(int i)
{
    if (cp[i] == i) return i;
    return cp[i] = iroot(cp[i]);
}
bool dirtree(int root)  // root: 树根
{
// vertex: 0 ~ n-1
    int i, j, k, circle = 0;
    memset(tag, -1, sizeof(tag));
    memset(to, -1, sizeof(to));
    for (i = 0; i < N; ++i) dis[i] = inf;
    for (j = 0; j < en; ++j)
    {
        i = iroot(edge[j].u);
        k = iroot(edge[j].v);
        if (k != i && dis[k] > edge[j].w)
        {
            dis[k] = edge[j].w;
            to[k] = i;
        }
    }
    to[root] = -1;
    dis[root] = 0;
    tag[root] = root;
    for (i = 0; i < N; ++i) if (cp[i] == i && -1 == tag[i])
        {
            j = i;
            for ( ; j != -1 && tag[j] == -1; j = to[j])
                tag[j] = i;
            if (j == -1) return 0;
            if (tag[j] == i)
            {
                circle = 1;
                tag[j] = -2;
                for (k = to[j]; k != j; k = to[k]) tag[k] = -2;
            }
        }
    if (circle)
    {
        for (j = 0; j < en; ++j)
        {
            i = iroot(edge[j].u);
            k = iroot(edge[j].v);
```

```
            if (k != i && tag[k] == -2) edge[j].w -= dis[k];
        }
        for (i = 0; i < N; ++i) if (tag[i] == -2)
            {
                res += dis[i];
                tag[i] = 0;
                for (j = to[i]; j != i; j = to[j])
                {
                    res += dis[j];
                    cp[j] = i;
                    tag[j] = 0;
                }
            }
        if (0 == dirtree(root)) return 0;
    }
    else
    {
        for (i = 0; i < N; ++i) if (cp[i] == i) res += dis[i];
    }
    return 1; // 若返回0 代表原图不连通
}
```

15.8.9 最小度限制生成树

15.8.10 最小生成森林(k颗树)：改进Kruskal($O(E \lg E)$)

数据结构 并查集

算法 改进Kruskal

根据Kruskal算法思想，图中的生成树在连完第$n-1$条边前，都是一个最小生成森林，每次贪心的选择两个不属于同一连通分量的树（如果连接一个连通分量，因为不会减少块数，那么就是不合算的）且用最“便宜”的边连起来，连接$n-1$次后就形成了一棵MST，$n-2$次就形成了一个两棵树的最小生成森林，$n-3, \cdots, n-k$此后就形成了k颗树的最小生成森林，就是题目要求求解的。

15.8.11 平面点的欧几里德最小生成树($O(V^2)$)

欧几里德距离：$dis(A,B) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$
先生成所有点之间的欧几里德距离的邻接矩阵，再用Prim算法即可。

15.8.12 平面点的曼哈顿最小生成树与莫队算法

曼哈顿距离：$dis(A,B) = |A_x - B_x| + |A_y - B_y|$

朴素算法($O(V^2)$)

先生成所有点之间的曼哈顿距离的邻接矩阵，再用Prim算法即可。

莫队算法中的最小生成树($O(VlgV)$)

每个点出发向8个方向的范围内最多只与一个点相连，那么我们只要想这8个点连边建图，这样只会有8n条边（去重了还剩4n），把这个图建出来，最小生成树就很好搞了。

(http://wenku.baidu.com/view/1e4878196bd97f192279e941.html)

POJ 3241 Object Clustering

```
/*
POJ 3241 Object Clustering
给你N个点，让你计算出K最小生成森林(K=1时为最小生成树)
*/
#include <cstdio>
#include <iostream>
#include <algorithm>
using namespace std;
#define INF 0x3f3f3f3f
#define eps 1e-8
#define pi acos(-1.0)
typedef long long ll;
const int maxn = 100100;
struct Point
{
    int x, y, id;
    bool operator < (const Point p) const
    {
        if (x != p.x)return x < p.x;
        return y < p.y;
    }
} p[maxn];
struct BIT
{
    int min_val, pos;
    void init()
    {
        min_val = INF;
        pos = -1;
    }
} bit[maxn << 2];
struct Edge
{
    int u, v, d;
    bool operator < (const Edge p) const
    {
        return d < p.d;
    }
```

```
} edge[maxn << 2];
int tot, n, F[maxn];
int find(int x)
{
    return F[x] = (F[x] == x ? x : find(F[x]));
}
void addedge(int u, int v, int d)
{
    edge[tot].u = u; edge[tot].v = v; edge[tot].d = d; tot++;
}
void update(int i, int val, int pos)
{
    while (i > 0)
    {
        if (val < bit[i].min_val)
        {
            bit[i].min_val = val;
            bit[i].pos = pos;
        }
        i -= i & (-i);
    }
}
int ask(int i, int m)
{
    int min_val = INF, pos = -1;
    while (i <= m)
    {
        if (bit[i].min_val < min_val)
        {
            min_val = bit[i].min_val;
            pos = bit[i].pos;
        }
        i += i & (-i);
    }
    return pos;
}
int dist(Point a, Point b)
{
    return abs(a.y - b.y) + abs(a.x - b.x);
}
int MHT(int n, Point *p, int k)
{
    int a[maxn], b[maxn];
    tot = 0;
    for (int dir = 0; dir < 4; dir++)
```

```
    {
        if (dir == 1 || dir == 3)
        {
            for (int i = 0; i < n; i++)
                swap(p[i].x, p[i].y);
        }
        if (dir == 2)
        {
            for (int i = 0; i < n; i++)
                p[i].x = -p[i].x;
        }
        sort(p, p + n);
        for (int i = 0; i < n; i++)
            a[i] = b[i] = p[i].y - p[i].x;
        sort(b, b + n);
        int m = unique(b, b + n) - b;
        for (int i = 1; i <= m; i++)bit[i].init();
        for (int i = n - 1; i >= 0; i--)
        {
            int pos = lower_bound(b, b + m, a[i]) - b + 1;
            int ans = ask(pos, m);
            if (ans != -1)
                addedge(p[i].id, p[ans].id, dist(p[i], p[ans]));
            update(pos, p[i].x + p[i].y, i);
        }
    }
    sort(edge, edge + tot);
    for (int i = 0; i < n; i++)F[i] = i;
    for (int i = 0; i < tot; i++)
    {
        int u = edge[i].u, v = edge[i].v;
        int fa = find(u), fb = find(v);
        if (fa != fb)
        {
            k--;
            F[fa] = fb;
            if (k == 0)return edge[i].d;
        }
    }
}
int main()
{
    int n, k;
    while (~scanf("%d%d", &n, &k))
    {
```

```
        for (int i = 0; i < n; i++)
            scanf("%d%d", &p[i].x, &p[i].y), p[i].id = i;
        cout << MHT(n, p, n - k) << endl;
    }
    return 0;
}
```

莫队算法($O(N^{1.5})$)(用于无修改区间查询)

莫队算法 对于两个区间的查询[l1,r1] ,[l2,r2]如果每增加一个区间元素或者删除，都能做到O(1)的话，那么从[l1,r1]转移到[l2,r2]，暴力可以做到—l1-l2—+—r1-r2—，就是manhattan距离。

曼哈顿最小生成树版本([2009国家集训队]小Z的袜子(hose),3284ms)

```
/*
[2009国家集训队]小Z的袜子(hose)
输入文件第一行包含两个正整数N和M。N为袜子的数量，M为小Z所提的询问的数量。
接下来一行包含N个正整数Ci，其中Ci表示第i只袜子的颜色，相同的颜色用相同的数字表示。
再接下来M行，每行两个正整数L，R表示一个询问。
求[L,R]区间取两只袜子相同颜色的概率
*/
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <vector>
#include <cstring>
#define lowbit(x) (x&(-x))
#define LL long long
using namespace std;
const int N = 50005;
struct Point
{
    int x, y, id;
    bool operator<(const Point p)const
    {
        return x != p.x ? x < p.x : y < p.y;
    }
} p[N], pp[N];
//数状数组，找(y-x)大于当前的，但是y+x最小的
struct BIT
{
    int min_val, pos;
    void init()
    {
        min_val = (1 << 30);
```

```
        pos = -1;
    }
} bit[N];
//所有有效边，Kruskal
struct Edge
{
    int u, v, d;
    bool operator<(const Edge e)const
    {
        return d < e.d;
    }
} e[N << 2];
//前向星
struct Graph
{
    int v, next;
} edge[N << 1];
int n, m, tot, pre[N];
int total, start[N];
int find(int x)
{
    return pre[x] = (x == pre[x] ? x : find(pre[x]));
}
inline int dist(int i, int j)
{
    return abs(p[i].x - p[j].x) + abs(p[i].y - p[j].y);
}
inline void addedge(int u, int v, int d)
{
    e[tot].u = u;
    e[tot].v = v;
    e[tot++].d = d;
}
inline void _add(int u, int v)
{
    edge[total].v = v;
    edge[total].next = start[u];
    start[u] = total++;
}
inline void update(int x, int val, int pos)
{
    for (int i = x; i >= 1; i -= lowbit(i))
        if (val < bit[i].min_val)
            bit[i].min_val = val, bit[i].pos = pos;
}
```

```
inline int ask(int x, int m)
{
    int min_val = (1 << 30), pos = -1;
    for (int i = x; i <= m; i += lowbit(i))
        if (bit[i].min_val < min_val)
            min_val = bit[i].min_val, pos = bit[i].pos;
    return pos;
}
inline void Manhattan_minimum_spanning_tree(int n, Point *p)
{
    int a[N], b[N];
    for (int dir = 0; dir < 4; dir++)
    {
        //4种坐标变换
        if (dir == 1 || dir == 3)
        {
            for (int i = 0; i < n; i++)
                swap(p[i].x, p[i].y);
        }
        else if (dir == 2)
        {
            for (int i = 0; i < n; i++)
            {
                p[i].x = -p[i].x;
            }
        }
        sort(p, p + n);
        for (int i = 0; i < n; i++)
        {
            a[i] = b[i] = p[i].y - p[i].x;
        }
        sort(b, b + n);
        int m = unique(b, b + n) - b;
        for (int i = 1; i <= m; i++)
            bit[i].init();
        for (int i = n - 1; i >= 0; i--)
        {
            int pos = lower_bound(b, b + m, a[i]) - b + 1; //BIT中从1开始
            int ans = ask(pos, m);
            if (ans != -1)
                addedge(p[i].id, p[ans].id, dist(i, ans));
            update(pos, p[i].x + p[i].y, i);
        }
    }
    sort(e, e + tot);
```

```
    for (int i = 0; i < n; i++)
        pre[i] = i;
    for (int i = 0; i < tot; i++)
    {
        int u = e[i].u, v = e[i].v;
        int fa = find(u), fb = find(v);
        if (fa != fb)
        {
            pre[fa] = fb;
            _add(u, v);
            _add(v, u);
        }
    }
}
LL gcd(LL a, LL b)
{
    return b == 0 ? a : gcd(b, a % b);
}
LL up[N], down[N];
LL ans;
int col[N], vis[N] = {0};
int cnt[N] = {0}; //记录每种颜色出现的次数
inline void add(int l, int r)
{
    for (int i = l; i <= r; i++)
    {
        int c = col[i];
        ans -= (LL)cnt[c] * (cnt[c] - 1) / 2;
        cnt[c]++;
        ans += (LL)cnt[c] * (cnt[c] - 1) / 2;
    }
}
inline void del(int l, int r)
{
    for (int i = l; i <= r; i++)
    {
        int c = col[i];
        ans -= (LL)cnt[c] * (cnt[c] - 1) / 2;
        cnt[c]--;
        ans += (LL)cnt[c] * (cnt[c] - 1) / 2;
    }
}
//[l1,r1]前一个区间 [l2,r2]当前区间
void dfs(int l1, int r1, int l2, int r2, int idx, int pre)
{
```

```
    if (l2 < l1) add(l2, l1 - 1);
    if (r2 > r1) add(r1 + 1, r2);
    if (l2 > l1) del(l1, l2 - 1);
    if (r2 < r1) del(r2 + 1, r1);
    up[pp[idx].id] = ans;
    vis[idx] = 1;
    for (int i = start[idx]; i != -1; i = edge[i].next)
    {
        int v = edge[i].v;
        if (vis[v]) continue;
        dfs(l2, r2, pp[v].x, pp[v].y, v, idx);
    }
    if (l2 < l1) del(l2, l1 - 1);
    if (r2 > r1) del(r1 + 1, r2);
    if (l2 > l1) add(l1, l2 - 1);
    if (r2 < r1) add(r2 + 1, r1);
}
int main()
{
    //freopen("input.txt","r",stdin);
    scanf("%d%d", &n, &m);
    tot = total = 0;
    memset(start, -1, sizeof(start));
    for (int i = 1; i <= n; i++)
        scanf("%d", &col[i]);
    for (int i = 0; i < m; i++)
    {
        scanf("%d%d", &p[i].x, &p[i].y);
        down[i] = (LL)(p[i].y - p[i].x + 1) * (p[i].y - p[i].x) / 2;
        p[i].id = i;
        pp[i] = p[i]; //副本一份，便于后面DFS，或者之后按id排序
    }
    Manhattan_minimum_spanning_tree(m, p);
    for (int i = 0; i < m; i++)
    {
        p[i].y = -p[i].y;
    }
    dfs(2, 1, pp[0].x, pp[0].y, 0, -1);
    for (int i = 0; i < m; i++)
    {
        LL g = gcd(up[i], down[i]);
        printf("%lld/%lld\n", up[i] / g, down[i] / g);
    }
    return 0;
```

```
}
```

无生成树版本([2009国家集训队]小Z的袜子(hose),884ms)

```
/*
[2009国家集训队]小Z的袜子(hose)
输入文件第一行包含两个正整数N和M。N为袜子的数量，M为小Z所提的询问的数量。
接下来一行包含N个正整数Ci，其中Ci表示第i只袜子的颜色，相同的颜色用相同的数字表示。
再接下来M行，每行两个正整数L，R表示一个询问。
求[L,R]区间取两只袜子相同颜色的概率

直接把x轴分块，每一块中按y升序排列就好了。
这样每个块内做一遍暴力的莫队，时间算下来O(n^1.5)。
具体实现是按照x/s,y排序(x是根号m)，
然后对于x/s相同的先把第一个求出来，然后暴力转移后面的。
*/
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cmath>
#include <cstring>
#define maxn 55000
#define inf 2147483647
using namespace std;
struct query
{
    int l, r, s, w;
} a[maxn];
int c[maxn];
long long col[maxn], size[maxn], ans[maxn];
int n, m, cnt, len;

long long gcd(long long x, long long  y)
{
    return (!x) ? y : gcd(y % x, x);
}

bool cmp(query a, query b)
{
    return (a.w == b.w) ? a.r < b.r : a.w < b.w;
}

int main()
{
    //freopen("hose.in","r",stdin);
    scanf("%d%d", &n, &m);
```

```
    for (int i = 1; i <= n; i++) scanf("%d", &c[i]);
    len = (int)sqrt(m);
    cnt = (len * len == m) ? len : len + 1;
    for (int i = 1; i <= m; i++)
    {
        scanf("%d%d", &a[i].l, &a[i].r);
        if (a[i].l > a[i].r) swap(a[i].l, a[i].r);
        size[i] = a[i].r - a[i].l + 1;
        a[i].w = a[i].l / len + 1;
        a[i].s = i;
    }
    sort(a + 1, a + m + 1, cmp);
    int i = 1;
    while (i <= m)
    {
        int now = a[i].w;
        memset(col, 0, sizeof(col));
        for (int j = a[i].l; j <= a[i].r; j++)
            ans[a[i].s] += 2 * (col[c[j]]++);
        i++;
        for (; a[i].w == now; i++)
        {
            ans[a[i].s] = ans[a[i - 1].s];
            for (int j = a[i - 1].r + 1; j <= a[i].r; j++)
                ans[a[i].s] += 2 * (col[c[j]]++);
            if (a[i - 1].l < a[i].l)
                for (int j = a[i - 1].l; j < a[i].l; j++)
                    ans[a[i].s] -= 2 * (--col[c[j]]);
            else
                for (int j = a[i].l; j < a[i - 1].l; j++)
                    ans[a[i].s] += 2 * (col[c[j]]++);
        }
    }
    long long all, num;
    for (int i = 1; i <= m; i++)
    {
        if (size[i] == 1) all = 1; else all = size[i] * (size[i] - 1);
        num = gcd(ans[i], all);
        printf("%lld/%lld\n", ans[i] / num, all / num);
    }
    return 0;
}
```

15.8.13 最小平衡生成树

15.8.14 生成树计数(Matrix-Tree定理)

Matrix-Tree定理(Kirchhoff矩阵-树定理)

- 1. G的度数矩阵D[G]是一个n*n的矩阵，并且满足：当i≠j 时,dij=0；当i=j 时，dij等于vi的度数。

  2. G的邻接矩阵A[G]也是一个n*n的矩阵，并且满足：如果vi、vj之间有边直接相连，则aij=1，否则为0。

- 我们定义G的Kirchhoff矩阵(也称为拉普拉斯算子)C[G]为C[G]=D[G]-A[G]，则Matrix-Tree定理可以描述为：G的所有不同的生成树的个数等于其Kirchhoff矩阵C[G]任何一个n-1阶主子式的行列式的绝对值。所谓n-1阶主子式，就是对于r(1⩽r⩽n)，将C[G]的第r行、第r列同时去掉后得到的新矩阵，用Cr[G]表示。

SPOJ HIGH Highways

```
/*
SPOJ HIGH Highways
N点、M条边的图，问生成树有多少个
*/
#include <cmath>
#include <cstdio>
#include <cstring>
using namespace std;
#define zero(x)((x>0? x:-x)<1e-15)
int const maxn = 100;
double a[maxn][maxn];
double b[maxn][maxn];
int g[53][53];
int N, M;
double det(double a[maxn][maxn], int n)
{
    int i, j, k, sign = 0;
    double ret = 1, t;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            b[i][j] = a[i][j];
    for (i = 0; i < n; i++)
    {
        if (zero(b[i][i]))
        {
            for (j = i + 1; j < n; j++)
                if (!zero(b[j][i]))
```

```
                break;
            if (j == n)
                return 0;
            for (k = i; k < n; k++)
                t = b[i][k], b[i][k] = b[j][k], b[j][k] = t;
            sign++;
        }
        ret *= b[i][i];
        for (k = i + 1; k < n; k++)
            b[i][k] /= b[i][i];
        for (j = i + 1; j < n; j++)
            for (k = i + 1; k < n; k++)
                b[j][k] -= b[j][i] * b[i][k];
    }
    if (sign & 1)
        ret = -ret;
    return ret;
}
int main()
{
#ifdef xysmlx
    freopen("in.cpp", "r", stdin);
#endif
    int T;
    scanf("%d", &T);//T组数据
    while (T--)
    {
        scanf("%d%d", &N, &M);//N,M
        memset(g, 0, sizeof(g));
        while (M--)//M条边，建图
        {
            int a, b;
            scanf("%d%d", &a, &b);
            g[a - 1][b - 1] = g[b - 1][a - 1] = 1;
        }
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++) a[i][j] = 0;
        }
        for (int i = 0; i < N; i++)
        {
            int d = 0;
            for (int j = 0; j < N; j++) if (g[i][j]) d++;
            a[i][i] = d;
        }
```

```
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                if (g[i][j]) a[i][j] = -1;
        double ans = det(a, N - 1);
        printf("%0.0lf\n", ans);
    }
    return 0;
}
```

15.8.15 最小生成树计数(BZOJ 1016)

```
/*
 *题目地址:
 *http://www.lydsy.com/JudgeOnline/problem.php?id=1016
 *
 *题目大意:
 *给出一个简单无向加权图,求这个图中有多少个不同的最小生成树;
 *由于不同的最小生成树可能很多,所以只需输出方案数对31011的模就可以了;
 *
 *算法思想:
 *Kruskal+Matrix_Tree定理;
 *
 *先按照任意顺序对等长的边进行排序;
 *然后利用并查集将所有长度为L0的边的处理当作一个阶段来整体看待;
 *可以定义一个数组的vector向量来保存每一个连通块的边的信息;
 *即将原图划分成多个连通块,每个连通块里面的边的权值都相同;
 *针对每一个连通块构建对应的Kirchhoff矩阵C,利用Matrix_Tree定理求每一个连通块的生成
树个数;
 *最后把他们的值相乘即可;
 *
 *Matrix_Tree定理:
 *G的所有不同的生成树的个数等于其Kirchhoff矩阵C[G]任何一个n-1阶主子式的行列式的绝
对值;
 *n-1阶主子式就是对于r(1≤r≤n),将C[G]的第r行,第r列同时去掉后得到的新矩阵,用Cr[G] 表
示;
**/
#include <cstdio>
#include <cmath>
#include <cstring>
#include <cstdlib>
#include <algorithm>
#include <vector>
using namespace std;
const int N = 111;
const int M = 1111;
```

```
const int mod = 31011;
struct Edges
{
    int a, b, c;
    bool operator<(const Edges &x)const
    {
        return c < x.c;
    }
} edge[M];

int n, m;
int f[N], U[N], vist[N]; //f,U都是并查集，U是每组边临时使用
int G[N][N], C[N][N]; //G顶点之间的关系，C为生成树计数用的Kirchhoff矩阵

vector<int>V[N];//记录每个连通分量

int Find(int x, int f[])
{
    if (x == f[x])
        return x;
    else
        return Find(f[x], f);
}

int det(int a[][N], int n) //生成树计数:Matrix-Tree定理
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            a[i][j] %= mod;
    int ret = 1;
    for (int i = 1; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
            while (a[j][i])
            {
                int t = a[i][i] / a[j][i];
                for (int k = i; k < n; k++)
                    a[i][k] = (a[i][k] - a[j][k] * t) % mod;
                for (int k = i; k < n; k++)
                    swap(a[i][k], a[j][k]);
                ret = -ret;
            }
        if (a[i][i] == 0)
            return 0;
        ret = ret * a[i][i] % mod;
```

```
    }
    if (ret < 0)
        ret = -ret;
    return (ret + mod) % mod;
}

void Solve()
{
    sort(edge, edge + m); //按权值排序
    for (int i = 1; i <= n; i++) //初始化并查集
    {
        f[i] = i;
        vist[i] = 0;
    }

    int Edge = -1; //记录相同的权值的边
    int ans = 1;
    for (int k = 0; k <= m; k++)
    {
        if (edge[k].c != Edge || k == m) //一组相等的边,即权值都为Edge的边加完
        {
            for (int i = 1; i <= n; i++)
            {
                if (vist[i])
                {
                    int u = Find(i, U);
                    V[u].push_back(i);
                    vist[i] = 0;
                }
            }
            for (int i = 1; i <= n; i++) //枚举每个连通分量
            {
                if (V[i].size() > 1)
                {
                    for (int a = 1; a <= n; a++)
                        for (int b = 1; b <= n; b++)
                            C[a][b] = 0;
                    int len = V[i].size();
                    for (int a = 0; a < len; a++) //构建Kirchhoff矩阵C
                        for (int b = a + 1; b < len; b++)
                        {
                            int a1 = V[i][a];
                            int b1 = V[i][b];
                            C[a][b] = (C[b][a] -= G[a1][b1]);
                            C[a][a] += G[a1][b1]; //连通分量的度
```

```
                        C[b][b] += G[a1][b1];
                    }
                int ret = (int)det(C, len);
                ans = (ans * ret) % mod; //对V中的每一个连通块求生成树个数再
相乘
                for (int a = 0; a < len; a++)
                    f[V[i][a]] = i;
            }
        }
        for (int i = 1; i <= n; i++)
        {
            U[i] = f[i] = Find(i, f);
            V[i].clear();
        }
        if (k == m)
            break;
        Edge = edge[k].c;
    }

    int a = edge[k].a;
    int b = edge[k].b;
    int a1 = Find(a, f);
    int b1 = Find(b, f);
    if (a1 == b1)
        continue;
    vist[a1] = vist[b1] = 1;
    U[Find(a1, U)] = Find(b1, U); //并查集操作
    G[a1][b1]++;
    G[b1][a1]++;
}

int flag = 0;
for (int i = 2; i <= n && !flag; i++)
    if (U[i] != U[i - 1])
        flag = 1;
if (m == 0)
    flag = 1;
printf("%d\n", flag ? 0 : ans % mod);

}

int main()
{
    while (~scanf("%d%d", &n, &m))
    {
```

```
        memset(G, 0, sizeof(G));
        for (int i = 1; i <= n; i++)
            V[i].clear();
        for (int i = 0; i < m; i++)
            scanf("%d%d%d", &edge[i].a, &edge[i].b, &edge[i].c);
        Solve();
    }
    return 0;
}
```

§ 15.9 最短路径

最长路径可用SPFA或Bellman-Ford做

15.9.1 有向无环图的最短路径：拓扑排序($O(N+E)$)

```
/**
*拓扑排序
*输入: g[maxn][maxn](图, 1~N)
*输出: topo[maxn](拓扑排序顺序)
*/
const int maxn=0;
int n,mk[maxn],topo[maxn],g[maxn][maxn],ps,topook;
void dfs(int u)
{
    if(mk[u]<0)
    {
        topook=0;
        return;
    }
    if(mk[u]>0) return;
    else mk[u]=-1;
    for(int v=1;topook&&v<=n;v++) if(g[u][v]) dfs(v);
    topo[ps--]=u;
    mk[u]=1;
}
void toposort()
{
    topook=1;
    ps=n;
    memset(mk,0,sizeof(mk));
    for(int i=1;topook&&i<=n;i++) if(!mk[i]) dfs(i);
}
```

15.9.2 非负权值加权图的最短路径：朴素Dijkstra算法(适用稠密图)($O(V^2)$)

```
/*
Dijkstra 数组实现 O(V^2)
Dijkstra --- 数组实现(在此基础上可直接改为STL的Queue实现)
d[] --- st到其他点的最近距离
path[] -- st为根展开的树,记录父亲结点
*/
const int maxn = 0;
const int inf = 0x3f3f3f3f;
int path[maxn];
bool vis[maxn];
int cost[maxn][maxn];
int d[maxn];
int n;
void dijkstra(int st)
{
    int i, j, minx;
    memset(vis, 0, sizeof(vis));
    vis[st] = 1;
    for (i = 0 ; i < n ; i++)
    {
        d[i] = cost[st][i];
        path[i] = st;
    }
    d[st] = 0;
    path[st] = -1;
    int pre = st;
    for (i = 1 ; i < n ; i++)
    {
        minx = inf;
        for (j = 0 ; j < n ; j++)//下面的加法可能导致溢出,INF不能取太大
        {
            if (vis[j] == 0 && d[pre] + cost[pre][j] < d[j] )
            {
                d[j] = d[pre] + cost[pre][j];
                path[j] = pre;
            }
        }
        for (j = 0 ; j < n ; j++)
        {
            if ( vis[j] == 0 && d[j] < minx )
            {
                minx = d[j];
                pre = j;
```

```
            }
        }
        vis[pre] = 1;
    }
}
```

15.9.3 非负权值加权图的最短路径：Dijkstra算法(二叉堆优化)($O((E+V)\lg V)$)

```
/**
*(不建议使用)
*非负权值加权图的最短路径：Dijkstra 算法(二叉堆优化)(From SJTU)(不建议使用)
*注：从1到n
*将dst去掉，将后面的while改为while(1) 就可以得到某点到其他所有点的最短距离了
*输入：用input() 函数输入和建立nbs[],ev[],ew[],next[],n,m; 输入src 和dst;
*输出：value[]
*/
const int maxn=0;
const int inf=0x3f3f3f3f;

int n,m,num,len,next[maxn],ev[maxn],ew[maxn];
int value[maxn],mk[maxn],nbs[maxn],ps[maxn],heap[maxn];

void update(int r)
{
    int q=ps[r],p=q>>1;
    while(p&&value[heap[p]]>value[r])
    {
        ps[heap[p]]=q;
        heap[q]=heap[p];
        q=p;
        p=q>>1;
    }
    heap[q]=r;
    ps[r]=q;
}

int getmin()
{
    int ret=heap[1],p=1,q=2,r=heap[len--];
    while(q<=len)
    {
        if(q<len&&value[heap[q+1]]<value[heap[q]]) q++;
        if(value[heap[q]]<value[r])
        {
```

```
            ps[heap[q]]=p;
            heap[p]=heap[q];
            p=q;
            q=p<<1;
        }
        else break;
    }
    heap[p]=r;
    ps[r]=p;
    return ret;
}

void dijkstra(int src,int dst)
{
    int u,v;
    for(int i=1;i<=n;i++)
    {
        value[i]=inf;
        mk[i]=ps[i]=0;
    }
    value[src]=0;
    heap[len=1]=src;
    ps[src]=1;
    while(!mk[dst])
    {
        if(len==0) return;
        u=getmin();
        mk[u]=1;
        for(int j=nbs[u];j;j=next[j])
        {
            v=ev[j];
            if(!mk[v]&&value[u]+ew[j]<value[v])
            {
                if(ps[v]==0)
                {
                    heap[++len]=v;
                    ps[v]=len;
                }
                value[v]=value[u]+ew[j];
                update(v);
            }
        }
    }
}
```

```
void input()
{
    int i,u,v,w;
    cin>>n>>m;//n 个顶点, m 条边
    num=0;
    memset(nbs,0,sizeof(nbs));
    while(m--)
    {
        cin>>u>>v>>w;
        next[++num]=nbs[u];
        nbs[u]=num;
        ev[num]=v;
        ew[num]=w;
    }
    dijkstra(1,n);
}
```

15.9.4 非负权值加权图的最短路径：Dijkstra 算法(优先队列优化)

```
/**
*Dijkstra 算法(优先队列优化)(From WJMZBMR)
*输入: 图(链式前向星),n(顶点数)(从0到n-1),st(起点)
*输出: Dist[](某点到其他所有点的距离)
*/
const int maxn=0;
const int maxm=0;
const int inf=0x3f3f3f3f;
int n;
struct Edge
{
    int v,w,id,next;//t:to, c:value, num:id
}edge[maxm];
int head[maxn],edgeNum;
void addSubEdge(int u,int v,int w,int id)
{
    edge[edgeNum].v=v;
    edge[edgeNum].w=w;
    edge[edgeNum].id=id;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void addEdge(int u,int v,int w,int id)
{
    addSubEdge(u,v,w,id);
    addSubEdge(v,u,w,id);
```

```
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
int Dist[maxn];
struct State
{
    int p,c;//p: 点   c: 值
    State(int _p,int _c):p(_p),c(_c) {}
    bool operator<(const State&o)const
    {
        return c>o.c;
    }
};
void Dijstra(int st)
{
    priority_queue<State> Q;
    fill(Dist,Dist+n,inf);
    Dist[st]=0;
    Q.push(State(st,0));
    while(!Q.empty())
    {
        State t=Q.top();
        Q.pop();
        if(t.c>Dist[t.p])continue;
        int ncost;
        for(int i=head[t.p];i!=-1;i=edge[i].next)
            if((ncost=t.c+edge[i].w)<Dist[edge[i].v])
            {
                Dist[edge[i].v]=ncost;
                Q.push(State(edge[i].v,Dist[edge[i].v]));
            }
    }
}
```

15.9.5 含负权值加权图的单源最短路径：Bellman-Ford 算法(适用负环未知)($O(VE)$)

```
/**
*含负权值加权图的单源最短路径：Bellman-Ford 算法(适用负环未知)($O(VE)$)
*输入：图(链式前向星),n(顶点数,从1到n)
*输出：d[](距离),是否有负环
*/
```

```
const int maxn=0;
const int maxm=0;
struct Edge
{
    int v,w;
    int next;
} edge[maxm];
int head[maxn],edgeNum;
void addSubEdge(int u,int v,int w)
{
    edge[edgeNum].v=v;
    edge[edgeNum].w=w;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
int n,m;
int d[maxn];
bool bellman_ford(int s)
{
    for(int i=1; i<n; i++)
    {
        bool flag=1;
        for(int u=1; u<=n; u++)
        {
            for(int j=head[u]; j!=-1; j=edge[j].next)
            {
                int v=edge[j].v;
                int w=edge[j].w;
                if(d[u]+w<d[v])//最长路改为>即可
                {
                    d[v]=d[u]+w;
                    flag=0;
                }
            }
        }
        if(flag) break;//return 1;(没有负环)
    }
    ///判断负环
    for(int u=1; u<=n; u++)
    {
```

```
        for(int j=head[u]; j!=-1; j=edge[j].next)
        {
            int v=edge[j].v;
            int w=edge[j].w;
            if(d[u]+w<d[v]) return 0;//有负环
        }
    }
    return 1;
}
```

15.9.6 含负权值加权图的单源最短路径：Bellman-Ford 算法(栈优化，适用负环未知)($O(VE)$)

```
/**
*含负权值加权图的单源最短路径：Bellman-Ford 算法(栈优化)(O(VE))
*输入：图(链式前向星),n(顶点数,从1到n)
*输出：d[](距离)
*/
const int maxn=0;
const int maxm=0;
struct Edge
{
    int v,w;
    int next;
}edge[maxm];
int head[maxn],edgeNum;
void addSubEdge(int u,int v,int w)
{
    edge[edgeNum].v=v;
    edge[edgeNum].w=w;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
int n,m;
int d[maxn];
void bellman_ford(int s)
{
    int mark[maxn];
    int q[maxn],top;//栈
    for(int i=1; i<=n; ++i) d[i]=inf;
```

```
    memset(mark,0,sizeof(mark));
    d[s]=0;
    mark[s]=1;
    top=0;
    q[top++]=s;
    while(top>0)
    {
        int k=q[--top];
        mark[k]=0;
        for(int i=head[k]; i!=-1; i=edge[i].next)
        {
            int s=edge[i].v;
            int w=edge[i].w;
            if(d[k]+w<d[s])
            {
                d[s]=d[k]+w;
                if(mark[s]==0)
                {
                    mark[s]=1;
                    q[top++]=s;
                }
            }
        }
    }
}
```

15.9.7 含负权值加权图的单源最短路径：Spfa 算法(稀疏图)($O(KE)$)

朴素SPFA

```
/**
*含负权值加权图的单源最短路径：Spfa 算法(稀疏图)(O(KE))(不适用分层图)
*输入：图(链式前向星),n(顶点数,从1到n)
*输出：d[](距离)
*/
const int maxn = 0;
const int maxm = 0;
struct Edge
{
    int u, v, w;
    int next;
} edge[maxm];
int head[maxn], en;
int n, m;
int d[maxn];
int pre[maxn];//用于解析路径
```

```
int num[maxn];//最短路径数量
int cnt[maxn];
bool mark[maxn];
queue<int> Q;
void addse(int u, int v, int w)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].w = w;
    edge[en].next = head[u];
    head[u] = en++;
}
void init()
{
    memset(head, -1, sizeof(head));
    en = 0;
}

/*DFS找负环
bool cir[maxn];
void dfs(int u)
{
    cir[u]=true;
    for(int i=head[u]; i!=-1; i=edge[i].next)
        if(!cir[edge[i].v]) dfs(edge[i].v);
}
*/
bool spfa(int s)
{
    memset(d, 0x3f, sizeof(int) * (n + 1));
    for (int i = 1; i <= n; ++i) pre[i] = i; //用于解析路径
    memset(mark, 0, sizeof(bool) * (n + 1));
    memset(cnt, 0, sizeof(int) * (n + 1));
    d[s] = 0;
    Q.push(s);
    mark[s] = 1;
    num[s] = 1;//最短路径数量
    cnt[s]++;
    while (Q.size())
    {
        int u = Q.front();
        Q.pop();
        mark[u] = 0;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
```

```
            int v = edge[i].v;
            int w = edge[i].w;
            if (d[u] + w < d[v])
            {
                pre[v] = u; // 用于解析路径
                d[v] = d[u] + w;
                num[v] = num[u];//最短路径数量
                if (mark[v] == 0)
                {
                    mark[v] = 1;
                    Q.push(v);
                    if (++cnt[v] > n) return false; //有负环，可以用DFS找
                }
            }
            else if (d[u] + w == d[v])//最短路径数量
            {
                num[v] += num[u];
            }
        }
    }
    return true;
}
```

SLF优化的SPFA

```
/**
*含负权值加权图的单源最短路径：Spfa 算法(稀疏图)(O(KE))(不适用分层图)(SLF优化的SPFA)
*输入：图(链式前向星),n(顶点数,从1到n)
*输出：d[](距离)
*/
const int maxn = 0;
const int maxm = 0;
struct Edge
{
    int u, v, w;
    int next;
} edge[maxm];
int head[maxn], en;
int n, m;
int d[maxn];
int pre[maxn];//用于解析路径
int cnt[maxn];
bool mark[maxn];
deque<int> Q;
void addse(int u, int v, int w)
```

```
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].w = w;
    edge[en].next = head[u];
    head[u] = en++;
}
void init()
{
    memset(head, -1, sizeof(head));
    en = 0;
}
/*DFS找负环
bool cir[maxn];
void dfs(int u)
{
    cir[u]=true;
    for(int i=head[u]; i!=-1; i=edge[i].next)
        if(!cir[edge[i].v]) dfs(edge[i].v);
}
*/
bool spfa(int s)
{
    while (!Q.empty()) Q.pop_front();
    memset(d, 0x3f, sizeof(int) * (n + 1));
    for (int i = 1; i <= n; ++i) pre[i] = i; //用于解析路径
    memset(mark, 0, sizeof(bool) * (n + 1));
    memset(cnt, 0, sizeof(int) * (n + 1));
    d[s] = 0;
    Q.push_back(s);
    mark[s] = 1;
    cnt[s]++;
    while (Q.size())
    {
        int u = Q.front();
        Q.pop_front();
        mark[u] = 0;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].w;
            if (d[u] + w < d[v])
            {
                pre[v] = u; // 用于解析路径
                d[v] = d[u] + w;
```

```
                if (mark[v] == 0)
                {
                    mark[v] = 1;
                    if (!Q.empty())
                    {
                        if (d[v] > d[Q.front()]) Q.push_back(v);
                        else Q.push_front(v);
                    }
                    else Q.push_back(v);
                    if (++cnt[v] > n) return false; //有负环，可以用DFS找
                }
            }
        }
    }
    return true;
}
```

SPFA求包含原点闭环的最短路(邻接矩阵版)

要计算从出发点出发的闭环的路径长度。所以要在普通SPFA的基础上做点变化。

就是把dist[start]设为INF。同时一开始并不是让出发点入队，而是让出发点能够到达的点入队。

```
/**
*含负权值加权图的单源最短路径：Spfa 算法(稀疏图)(O(KE))(不适用分层图)
*输入：图(链式前向星),n(顶点数,从1到n)
*输出：d[](距离)
*/
const int maxn = 0;
int n;
int d[maxn];
int pre[maxn];//用于解析路径
int num[maxn];//最短路径数量
int cnt[maxn];
bool mark[maxn];
queue<int> Q;

/*DFS找负环
bool cir[maxn];
void dfs(int u)
{
    cir[u]=true;
    for(int i=head[u]; i!=-1; i=edge[i].next)
        if(!cir[edge[i].v]) dfs(edge[i].v);
}
*/
```

```
bool spfa(int s)
{
    while (!Q.empty()) Q.pop();
    memset(d, 0x3f, sizeof(int) * (n + 1));
    for (int i = 1; i <= n; ++i) pre[i] = i; //用于解析路径
    memset(mark, 0, sizeof(bool) * (n + 1));
    memset(cnt, 0, sizeof(int) * (n + 1));

    for (int v = 0; v < n; v++)
    {
        if (v == s)
        {
            d[v] = inf;
            mark[v] = 0;
        }
        else if (g[s][v] != inf)
        {
            d[v] = g[s][v];
            Q.push(v);
            mark[v] = 1;
            num[v] = 1;//最短路径数量
            cnt[v]++;
        }
        else
        {
            d[v] = inf;
            mark[v] = 0;
        }
    }

    while (Q.size())
    {
        int u = Q.front();
        Q.pop();
        mark[u] = 0;
        for (int v = 0; v < n; v++)
        {
            if (g[u][v] == inf) continue;//不邻接
            int w = g[u][v];
            if (d[u] + w < d[v])
            {
                pre[v] = u; // 用于解析路径
                d[v] = d[u] + w;
                num[v] = num[u];//最短路径数量
                if (mark[v] == 0)
```

```
                {
                    mark[v] = 1;
                    Q.push(v);
                    if (++cnt[v] > n) return false; //有负环，可以用DFS找
                }
            }
            else if (d[u] + w == d[v])//最短路径数量
            {
                num[v] += num[u];
            }
        }
    }
    return true;
}
```

15.9.8 全源最短路径：Floyd 算法($O(V^3)$)

```
/**
*全源最短路径：Floyd 算法
*输入：mtx[][](从0到n-1)
*输出：mtx[][](最短路径长度),path[][](从后往前的最短路径)
*/
const int maxn=0;
int mtx[maxn][maxn];
int path[maxn][maxn];
int n;

void floyd()
{
    for(int i=0;i<n;i++) for(int j=0;j<n;j++) path[i][j]=i;
    for(int k=0;k<n;k++)
    {
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<n;j++)
            {
                if(mtx[i][k]+mtx[k][j]<mtx[i][j])
                {
                    mtx[i][j]=mtx[i][k]+mtx[k][j];
                    path[i][j]=path[k][j];// 从后往前的，要用栈得到正向路径
                }
            }
        }
    }
}
```

15.9.9 全源最短路径：Johnson 算法(稀疏图)($O(EV\lg V)$)

15.9.10 次短路径

```
/*
POJ 3255 (次短路经)
*/
#include <iostream>
#include <queue>
using namespace std;
const int maxn = 5010;
const int maxm = 200010;
const int inf = 0x3f3f3f3f;

typedef struct
{
    int v, w, next;
} Edge;
Edge edge[maxm];

int d[maxn], dr[maxn];
int n, m, en;
int head[maxn];
bool vis[maxn];

void init()
{
    memset(head, -1, sizeof(head));
    en = 0;
    for (int i = 1; i <= n; i++) d[i] = dr[i] = inf;
}

void addedge(int u, int v, int w)
{
    edge[en].v = v;
    edge[en].w = w;
    edge[en].next = head[u];
    head[u] = en++;
}

void spfa(int st, int dt[])
{
    int i, v, u;
    queue<int>q;
```

```
    memset(vis, 0, sizeof(vis));
    dt[st] = 0;
    vis[st] = 1;
    q.push(st);
    while (!q.empty())
    {
        v = q.front(); q.pop();
        vis[v] = 0;
        for (i = head[v]; i != -1; i = edge[i].next)
        {
            u = edge[i].v;
            if (dt[v] + edge[i].w < dt[u])
            {
                dt[u] = dt[v] + edge[i].w;
                if (!vis[u])
                {
                    vis[u] = 1;
                    q.push(u);
                }
            }
        }
    }
}

int main()
{
    int a, b, c;
    int ans, tmp, i;
    while (~scanf("%d%d", &n, &m))
    {
        init();
        while (m--)
        {
            scanf("%d%d%d", &a, &b, &c);
            addedge(a, b, c);
            addedge(b, a, c);
        }

        spfa(1, d);
        spfa(n, dr);

        ans = inf;
        for (i = 1; i <= n; i++)
        {
            for (int j = head[i]; j != -1; j = edge[j].next)
```

```
            {
                b = edge[j].v;
                c = edge[j].w;
                tmp = d[i] + dr[b] + c;
                if (tmp > d[n] && ans > tmp)
                    ans = tmp;
            }
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

15.9.11 第k短路径

第k短路径：A*

```
/*
* 第K短路径(A*)
* POJ 2449
*/
#include <queue>
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
const int maxn = 1005;
const int maxm = 200005;

int n, m;
int st, ed, K;
struct Edge
{
    int v, w, next;
    bool rev;
} edge[maxm];
int en, pnt[maxn][2];

void addse(int x, int y, int z, bool r = false)
{
    edge[en].v = y;
    edge[en].w = z;
    edge[en].rev = r;
    edge[en].next = pnt[x][r];
    pnt[x][r] = en++;
}
```

```
void init()
{
    en = 0;
    for (int i = 1; i <= n; i++) pnt[i][0] = pnt[i][1] = -1;
    while (m--)
    {
        int x, y, z;
        scanf("%d%d%d", &x, &y, &z);
        addse(x, y, z);
        addse(y, x, z, true);
    }
}

//reversed edges
int f[maxn];
bool inq[maxn];
void spfa()
{
    for (int i = 1; i <= n; i++)
    {
        f[i] = -1;
        inq[i] = false;
    }
    queue<int> q;
    f[ed] = 0;
    q.push(ed);
    inq[ed] = true;
    while (!q.empty())
    {
        int cx = q.front();
        q.pop();
        inq[cx] = false;
        for (int i = pnt[cx][1]; i >= 0; i = edge[i].next)
        {
            int nx = edge[i].v;
            if (f[nx] < 0 || f[nx] > f[cx] + edge[i].w)
            {
                f[nx] = f[cx] + edge[i].w;
                if (!inq[nx])
                {
                    q.push(nx);
                    inq[nx] = true;
                }
            }
```

```
        }
    }
}

struct Node
{
    int v, f;
    Node() {}
    Node(int x, int y)
    {
        v = x;
        f = y;
    }
    bool operator < (const Node &x)const
    {
        return f > x.f;
    }
};
int cnt[maxn];
int astar()
{
    if (f[st] < 0) return -1;
    for (int i = 1; i <= n; i++) cnt[i] = 0;
    priority_queue<Node> q;
    q.push(Node(st, f[st]));
    while (!q.empty())
    {
        Node tmp = q.top();
        q.pop();
        int cx = tmp.v;
        int cy = tmp.f;
        cnt[cx] += 1;
        if (cnt[ed] == K) return cy;
        if (cnt[cx] > K) continue;
        for (int i = pnt[cx][0]; i >= 0; i = edge[i].next)
        {
            q.push(Node(edge[i].v, cy + edge[i].w + f[edge[i].v] - f[cx]));
        }
    }
    return -1;
}

void work()
{
    scanf("%d%d%d", &st, &ed, &K);
```

```
    K += (st == ed);
    spfa();
    printf("%d\n", astar());
}

int main()
{
    while (~scanf("%d%d", &n, &m))
    {
        init();
        work();
    }
    return 0;
}
```

前k短路径: Dijkstra变形

```
/*
*前K短路径O(VK*(lg(V^2K)+V))
*/
struct Heap
{
    int x;
    int dis;
    Heap(int _x, int _dis): x(_x), dis(_dix) {}
    Heap() {}
    bool operator < (cosnt Heap &o) const
    {
        return dis > o.dis;
    }
};

int n, m;
int tot;
int st, ed, K;
int dis[maxn][maxn];
int cnt[maxn];
int head[maxn];
priority_queue<Heap> pq;
int dijkstra(int st)
{
    Heap u = Heap(st, 0);
    pq.push(u);
    while (!pq.empty())
    {
```

```
        u = pq.top();
        pq.pop();
        dis[u.x][++cnt[u.x]] = u.dis;
        for (int i = head[u.x]; ~i; i = edge[i].next)
        {
            Heap v = Heap(edge[i].v, u.dis + edge[i].w);
            if (cnt[v.x] < K)
            {
                pq.push(v);
            }
        }
    }
    return -1;
}
```

15.9.12 差分约束系统：SPFA($O(KE)$)

Theorm

$A_x \leq b$给出的约束条件是m个差分约束集合，其中包含n个未知量，对应的线性规划矩阵A为m 行n 列。每个约束条件为如下形式的简单线性不等式：$x_j - x_i \leq b_k$。其中$1 \leq i, j \leq n 1 \leq k \leq m$。

在一个差分约束系统$A_x \leq b$ 中，mXn 的线性规划矩阵A可被看做是n顶点，m条边的图的关联矩阵。对于$i = 1, 2, \cdots, n$，图中的每一个顶点v_i对应着n 个未知量的一个x_i。图中的每个有向边对应着关于两个未知量的m个不等式中的一个。

顶点集合V由对应于每个未知量x_i的顶点v_i和附加的顶点v_0组成。边的集合E由对应于每个差分约束条件的边与对应于每个未知量x_i的边(v_0, v_i) 构成。如果$x_j - x_i \leq b_k$是一个差分约束，则边(v_i, v_j)的权$w(v_i, v_j) = b_k$ （注意i和j 不能颠倒），从v_0出发的每条边的权值均为0。

给定一差分约束系统$A_x \leq b$，设$G = (V, E)$ 为其相应的约束图。如果G不包含负权回路，那么$x = (d(v_0, v_1), d(v_0, v_2), \cdots, d(v_0, v_n))$是此系统的一可行解，其中$d(v_0, v_i)$是约束图中$v_0$到$v_i$的最短路径($i = 1, 2, \cdots, n$)。如果$G$包含负权回路，那么此系统不存在可行解。

$A_x \geq b$可转化成最长路径

最短路解得在某个变量确定的情况下，其他所有变量都取到所能取的最大值。

最长路解得在某个变量确定的情况下，其他所有变量都取到所能取的最小值。

模板：SPFA($O(KE)$)

邻接表版

```
//负环要用普通的bellman-ford
//类似，求解时可以用各种最短距离算法，有时TLE有时AC
//有时求解最长路径，将'<'改成'>'，将inf改成-inf
struct Edge
{
    int v,w;
    int next;
}edge[maxm];
```

```
int head[maxn],edgeNum;
void addSubEdge(int u,int v,int w)
{
    edge[edgeNum].v=v;
    edge[edgeNum].w=w;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
int n,m;
int d[maxn];
void spfa(int s)
{
    int mark[maxn];
    queue<int> Q;
    for(int i=1; i<=n; ++i) d[i]=inf;//-inf
    memset(mark,0,sizeof(mark));
    d[s]=0;
    Q.push(s);
    mark[s]=1;
    while(Q.size())
    {
        int k=Q.front();
        Q.pop();
        mark[k]=0;
        for(int i=head[k]; i!=-1; i=edge[i].next)
        {
            int s=edge[i].v;
            int w=edge[i].w;
            if(d[k]+w<d[s])//>
            {
                d[s]=d[k]+w;
                if(mark[s]==0)
                {
                    mark[s]=1;
                    Q.push(s);
                }
            }
        }
    }
}
```

带负环的情况：Bellman-Ford

一道例题：HDU 3776 Task

```
/**
*注意：xi-xj>=d，最长路处理，注意要加xi-xj>=0的约束
*/
#include<cstdio>
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
typedef long long LL;
const int maxn=110;
const int maxm=100010;
const int inf=1000000;
struct Edge
{
    int v,w;
    int next;
} edge[maxm];
int head[maxn],edgeNum;
void addSubEdge(int u,int v,int w)
{
    edge[edgeNum].v=v;
    edge[edgeNum].w=w;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
int n,m;
int d[maxn];
bool bellman_ford(int s)
{
    fill(d,d+n+1,1);
    for(int i=1; i<=n-1; i++)
    {
        int flag=1;
        for(int u=1; u<=n; u++)
        {
            for(int j=head[u]; j!=-1; j=edge[j].next)
            {
```

```
                int v=edge[j].v;
                int w=edge[j].w;
                if(d[u]+w>d[v])
                {
                    flag=0;
                    d[v]=d[u]+w;
                }
            }
        }
        if(flag) break;
    }
    for(int u=1; u<=n; u++)
    {
        if(d[u]>=inf) return false;// 处理环
        for(int j=head[u]; j!=-1; j=edge[j].next)
        {
            int v=edge[j].v;
            int w=edge[j].w;
            if(d[u]+w>d[v]) return false;
        }
    }
    return true;
}
void input()
{
    scanf("%d",&m);
    char ts[1010];
    int a,b,c;
    while(m--)
    {
        scanf("%*s%d%*s%s",&a,ts);
        if(ts[0]=='a')
        {
            scanf("%*s%d%*s%*s%*s%*s%d", &c, &b);
            addSubEdge(b,a,c);
        }
        else
        {
            scanf("%d%*s%*s%*s%*s%*s%*s%*s%d", &c, &b);
            addSubEdge(a,b,-c);
            addSubEdge(b,a,0);
        }
    }
}
void solve()
```

```
{
    int flag=bellman_ford(1);
    if(!flag)
    {
        puts("Impossible.");
        return;
    }
    for(int i=1; i<=n; i++)
    {
        printf("%d",d[i]);
        if(i<n) printf(" ");
    }
    puts("");
}
int main()
{
    while(~scanf("%d",&n))
    {
        if(!n) return 0;
        init();
        input();
        solve();
    }
    return 0;
}
```

上题另解

```
/**
*注意: xi-xj>=d, 最长路处理, 注意要加xi-xj>=0的约束
*/
#include<cstdio>
#include<iostream>
#include<cstring>
#include<queue>
#include<algorithm>
using namespace std;
const int maxn=110;
const int maxm=100010;
const int inf=1000000;
struct Edge
{
    int v,w;
    int next;
} edge[maxm];
int head[maxn],edgeNum;
```

```
void addSubEdge(int u,int v,int w)
{
    edge[edgeNum].v=v;
    edge[edgeNum].w=w;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
}
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
int n,m;
int d[maxn];
int cnt[maxn];
bool spfa(int s)
{
    int mark[maxn];
    queue<int> Q;
    memset(mark,0,sizeof(mark));
    memset(cnt,0,sizeof(cnt));
    d[s]=1;
    Q.push(s);
    mark[s]=1;
    ++cnt[s];
    while(Q.size())
    {
        int k=Q.front();
        Q.pop();
        mark[k]=0;
        for(int i=head[k]; i!=-1; i=edge[i].next)
        {
            int s=edge[i].v;
            int w=edge[i].w;
            if(d[k]+w<d[s])//>
            {
                d[s]=d[k]+w;
                if(mark[s]==0)
                {
                    mark[s]=1;
                    Q.push(s);
                    if(++cnt[s]>n) return false;
                }
            }
        }
```

```
    }
    return true;
}
void input()
{
    scanf("%d",&m);
    char ts[1010];
    int a,b,c;
    while(m--)
    {
        scanf("%*s%d%*s%s",&a,ts);
        if(ts[0]=='a')
        {
            scanf("%*s%d%*s%*s%*s%*s%d", &c, &b);
            addSubEdge(b,a,c);
        }
        else
        {
            scanf("%d%*s%*s%*s%*s%*s%*s%*s%d", &c, &b);
            addSubEdge(a,b,-c);
            addSubEdge(b,a,0);
        }
    }
}
void solve()
{
    fill(d,d+n+1,inf);
    for(int i=1; i<=n; i++)
    {
        if(d[i]>=inf)
        {
            if(!spfa(i))
            {
                puts("Impossible.");
                return;
            }
        }
    }
    int minx=inf;
    for(int i=1; i<=n; i++) minx=min(minx,d[i]);
    minx--;
    for(int i=1; i<=n; i++)
    {
        printf("%d",d[i]-minx);
        if(i<n) printf(" ");
```

```
    }
    puts("");
}
int main()
{
    while(~scanf("%d",&n))
    {
        if(!n) return 0;
        init();
        input();
        solve();
    }
    return 0;
}
```

邻接矩阵版

15.9.13 平面点对的最短路径(优化)

15.9.14 双标准限制最短路径

§ 15.10 匹配

15.10.1 二分图最大匹配：Hungary算法($O(VE)$)

主框架

```
bool 寻找从k出发的对应项出的可增广路
{
    while (从邻接表中列举k能关联到顶点j)
    {
        if (j不在增广路上)
        {
            把j加入增广路;
            if (j是未盖点 或者 从j 的对应项出发有可增广路)
            {
                修改j的对应项为k;
                则从k的对应项出有可增广路,返回true;
            }
        }
    }
    则从k的对应项出没有可增广路,返回false;
}
```

```
void 匈牙利hungary()
{
    for i->1 to n
    {
        if (则从i的对应项出有可增广路)
            匹配数++;
    }
    输出 匹配数;
}
```

邻接矩阵实现

```
/**
*二分图匹配：匈牙利算法的DFS实现(O(VE))
*适于稠密图，DFS找增广路快
*输入：g[][]两边定点划分的情况
*输出：hungary()(最大匹配数),cx[]cy[](匹配)
*字典序最大：在hungary()中从n-1到0地扫
*/
const int maxn = 0;
int uN, vN; //u,v数目
int g[maxn][maxn];//编号是0~n-1 的
int cx[maxn], cy[maxn];
bool used[maxn];
bool dfs(int u)
{
    int v;
    for (v = 0; v < vN; v++)
    {
        if (g[u][v] && !used[v])
        {
            used[v] = true;
            if (cy[v] == -1 || dfs(cy[v]))
            {
                cx[u] = v;
                cy[v] = u;
                return true;
            }
        }
    }
    return false;
}
int hungary()
{
    int res = 0;
```

```
    int u;
    memset(cx, -1, sizeof(cx));
    memset(cy, -1, sizeof(cy));
    for (u = 0; u < uN; u++)//默认最小字典序，在这里uN-1->0扫描使得字典序最大
    {
        memset(used, 0, sizeof(used));
        if (dfs(u)) res++;
    }
    return res;
}
```

链式前向星实现

```
/**
*二分图匹配：匈牙利算法的DFS实现(O(VE))
*适于稠密图，DFS找增广路快
*输入：链式前向星
*输出：hungary()(最大匹配数),cx[]cy[](匹配)
*字典序最大：在hungary()中从n-1到0地扫
*/
const int maxn = 0;
const int maxm = 0;
struct Edge
{
    int u, v;
    int next;
} edge[maxm];
int en, head[maxn];
int uN, vN; //u,v数目
int cx[maxn], cy[maxn];
bool used[maxn];
bool dfs(int u)
{
    for (int i = head[u]; ~i; i = edge[i].next)
    {
        int v = edge[i].v;
        if (!used[v])
        {
            used[v] = true;
            if (cy[v] == -1 || dfs(cy[v]))
            {
                cx[u] = v;
                cy[v] = u;
                return true;
            }
```

```
        }
    }
    return false;
}
int hungary()
{
    int res = 0;
    int u;
    memset(cx, -1, sizeof(cx));
    memset(cy, -1, sizeof(cy));
    for (u = 0; u < uN; u++)//默认最小字典序，在这里uN-1->0扫描使得字典序最大
    {
        memset(used, 0, sizeof(used));
        if (dfs(u)) res++;
    }
    return res;
}
```

15.10.2 大数据二分图最大匹配：Hopcroft-Karp($O(\sqrt{V}E)$)

```
/**
*大数据二分图匹配：Hopcroft-Karp($O(\sqrt{v}E)$)
*适用于数据较大的二分匹配(从0到n-1)
*输入：Nx,Ny,g[][]
*输出：res=MaxMatch();Mx[]My[]
*/
const int maxn = 0;
const int inf = 0x3f3f3f3f;
int g[maxn][maxn], Mx[maxn], My[maxn], Nx, Ny;
int dx[maxn], dy[maxn], dis;
bool vst[maxn];
bool searchP()
{
    queue<int>Q;
    dis = inf;
    memset(dx, -1, sizeof(dx));
    memset(dy, -1, sizeof(dy));
    for (int i = 0; i < Nx; i++)
        if (Mx[i] == -1)
        {
            Q.push(i);
            dx[i] = 0;
        }
    while (!Q.empty())
    {
```

```
        int u = Q.front();
        Q.pop();
        if (dx[u] > dis)  break;
        for (int v = 0; v < Ny; v++)
            if (g[u][v] && dy[v] == -1)
            {
                dy[v] = dx[u] + 1;
                if (My[v] == -1)  dis = dy[v];
                else
                {
                    dx[My[v]] = dy[v] + 1;
                    Q.push(My[v]);
                }
            }
    }
    return dis != inf;
}
bool DFS(int u)
{
    for (int v = 0; v < Ny; v++)
        if (!vst[v] && g[u][v] && dy[v] == dx[u] + 1)
        {
            vst[v] = 1;
            if (My[v] != -1 && dy[v] == dis) continue;
            if (My[v] == -1 || DFS(My[v]))
            {
                My[v] = u;
                Mx[u] = v;
                return 1;
            }
        }
    return 0;
}
int MaxMatch()
{
    int res = 0;
    memset(Mx, -1, sizeof(Mx));
    memset(My, -1, sizeof(My));
    while (searchP())
    {
        memset(vst, 0, sizeof(vst));
        for (int i = 0; i < Nx; i++)
            if (Mx[i] == -1 && DFS(i))  res++;
    }
    return res;
```

```
}
```

15.10.3 二分图多重匹配：Hungary算法改($O(VE)$)

```
/*
二分图的多重匹配：匈牙利算法
输入：cap[](y图的匹配数限制),g[][](图)
输出：mulmatch(),link[][]
*/
const int maxn = 0;
int cap[maxn], g[maxn][maxn], vlink[maxn], link[maxn][maxn];
bool vis[maxn];
int nx, ny;
int path(int s)
{
    for (int i = 0; i < ny; i++)
    {
        if (g[s][i] && !vis[i])
        {
            vis[i] = true;
            if (vlink[i] < cap[i])
            {
                link[i][vlink[i]++] = s;
                return 1;
            }
            for (int j = 0; j < vlink[i]; j++)
            {
                if (path(link[i][j]))
                {
                    link[i][j] = s;
                    return 1;
                }
            }
        }
    }
    return 0;
}
bool mulmatch()
{
    memset(vlink, 0, sizeof(vlink));
    for (int i = 0; i < nx; i++)
    {
        memset(vis, 0, sizeof(vis));
        if (!path(i))
            return 0;
```

```
    }
    return 1;
}
```

15.10.4 二分图的几个等价

最大边独立集 边集导出子图不含公共点叫独立集，最大的叫最大边独立集

最大独立集 点集导出子图不含边叫独立集，最大的叫最大独立集

最小支配集 点集，原图任意顶点要么属于此点集，要么与此点集的点邻接，最小的叫最小支配集

最大团 点集导出子集中任意两点均有边，最大的叫最大团

最小点覆盖 边集，边的两端点的集合是原图的点集，最小的叫最小点覆盖

最小路径覆盖 点集，点是所有边的两端点之一，最小的叫最小路径覆盖

等价

- 最小点覆盖 $= |V| -$ 最大独立集
- 最大独立点集 = 最大完全子图
- 二分图最小点覆盖集 = 二分图最大匹配
- 二分图最大独立点集 $= |V|$二分图最小点覆盖集
- 最大团 = 补图的最大点独立集
- 有向图最小路径覆盖 $= |V|$最大匹配
- 无向图最小路径覆盖 $= |V|$最大匹配/2

15.10.5 二分图带权(最大/最小)完备匹配：Kuhn-Munkras算法($O(N^3)$)

```
/**
*二分图带权(最大/最小)完备匹配：Kuhn-Munkras算法($O(N^3)$)
*lx[],ly[]为顶标，nx,ny为x,y顶点数，sx[],sy[]表示visx,visy
*默认最大，若求最小则把权值取反即可
*输入：g[][],nx,ny
*输出：cx[],cy[],KuhnMunkres()(最大匹配)
*/
const int inf = 0x3f3f3f3f;
const int maxn = 0;
int cx[maxn], cy[maxn], nx, ny, match;
bool sx[maxn], sy[maxn];
double lx[maxn], ly[maxn], g[maxn][maxn];
```

```
bool path(int u)
{
    sx[u] = 1;
    for (int v = 1; v <= ny; v++)
        if (g[u][v] == lx[u] + ly[v] && !sy[v])
        {
            sy[v] = 1;
            if (!cy[v] || path(cy[v]))
            {
                cx[u] = v;
                cy[v] = u;
                return 1;
            }
        }
    return 0;
}

int KuhnMunkres()
{
    int i, j, u, minx;
    memset(lx, 0, sizeof (lx));
    memset(ly, 0, sizeof (ly));
    memset(cx, 0, sizeof (cx));
    memset(cy, 0, sizeof (cy));
    for (i = 1; i <= nx; i++)
        for (j = 1; j <= ny; j++)
            lx[i] = max(lx[i], g[i][j]);
    for (match = 0, u = 1; u <= nx; u++)
        if (!cx[u])
        {
            memset(sx, 0, sizeof (sx));
            memset(sy, 0, sizeof (sy));
            while (!path(u))//没找到增广路径
            {
                minx = inf;
                for (i = 1; i <= nx; i++)
                    if (sx[i])
                        for (j = 1; j <= ny; j++)
                            if (!sy[j]) minx = min(minx, lx[i] + ly[j] - g[i][j]);
                for (i = 1; i <= nx; i++)
                    if (sx[i])
                    {
                        lx[i] -= minx;
                        sx[i] = 0;
                    }
```

```
                for (j = 1; j <= ny; j++)
                    if (sy[j])
                    {
                        ly[j] += minx;
                        sy[j] = 0;
                    }
            }
        }
    int ret = 0; //计算最大匹配
    for (int i = 1; i <= ny; i++)
        if (cy[i] > 0) ret += g[cy[i]][i];
    /**与上面等价
    for(int i=1;i<=nx;i++)
        if(cx[i]>0) ret+=g[i][cx[i]];
    */
    return ret;
}
```

15.10.6 一般图最大匹配：带花树算法(未知复杂度)

```
/**
*一般图的最大基数匹配：带花树算法
*输入：g[][],n(输入从0到n-1,用addEdge()加边)
*输出：gao()(最大匹配数),match[](匹配)
*/
const int maxn = 0;
struct Matching
{
    deque<int> Q;
    int n;
    //g[i][j]存放关系图：i,j是否有边,match[i]存放i所匹配的点
    bool g[maxn][maxn], inque[maxn], inblossom[maxn], inpath[maxn];
    int match[maxn], pre[maxn], base[maxn];

    //找公共祖先
    int findancestor(int u, int v)
    {
        memset(inpath, 0, sizeof(inpath));
        while (1)
        {
            u = base[u];
            inpath[u] = true;
            if (match[u] == -1)break;
            u = pre[match[u]];
        }
```

```
    while (1)
    {
        v = base[v];
        if (inpath[v])return v;
        v = pre[match[v]];
    }
}

//压缩花
void reset(int u, int anc)
{
    while (u != anc)
    {
        int v = match[u];
        inblossom[base[u]] = 1;
        inblossom[base[v]] = 1;
        v = pre[v];
        if (base[v] != anc)pre[v] = match[u];
        u = v;
    }
}

void contract(int u, int v, int n)
{
    int anc = findancestor(u, v);
    //SET(inblossom,0);
    memset(inblossom, 0, sizeof(inblossom));
    reset(u, anc);
    reset(v, anc);
    if (base[u] != anc)pre[u] = v;
    if (base[v] != anc)pre[v] = u;
    for (int i = 1; i <= n; i++)
        if (inblossom[base[i]])
        {
            base[i] = anc;
            if (!inque[i])
            {
                Q.push_back(i);
                inque[i] = 1;
            }
        }
}

bool dfs(int S, int n)
{
```

```
        for (int i = 0; i <= n; i++)pre[i] = -1, inque[i] = 0, base[i] = i;
        Q.clear();
        Q.push_back(S);
        inque[S] = 1;
        while (!Q.empty())
        {
            int u = Q.front();
            Q.pop_front();
            for (int v = 1; v <= n; v++)
            {
                if (g[u][v] && base[v] != base[u] && match[u] != v)
                {
                    if (v == S || (match[v] != -1 && pre[match[v]] != -1))contract(u, v, n);
                    else if (pre[v] == -1)
                    {
                        pre[v] = u;
                        if (match[v] != -1)Q.push_back(match[v]), inque[match[v]] = 1;
                        else
                        {
                            u = v;
                            while (u != -1)
                            {
                                v = pre[u];
                                int w = match[v];
                                match[u] = v;
                                match[v] = u;
                                u = w;
                            }
                            return true;
                        }
                    }
                }
            }
        }
        return false;
    }

    void init(int n)
    {
        this->n = n;
        memset(match, -1, sizeof(match));
        memset(g, 0, sizeof(g));
    }

    void addEdge(int a, int b)
```

```
    {
        ++a;
        ++b;
        g[a][b] = g[b][a] = 1;
    }

    int gao()
    {
        int ans = 0;
        for (int i = 1; i <= n; ++i)
            if (match[i] == -1 && dfs(i, n))
                ++ans;
        return ans;
    }
};
```

15.10.7 稳定婚姻匹配($O(N^2)$)

```
/**
*POJ 3487 The Stable Marriage Problem
*男的优先的稳定婚姻匹配
*DATA:
1 // test case
3 // 男女数
a b c A B C //Male name is a lowercase letter, female name is an upper-case letter
a:BAC
b:BAC
c:ACB
A:acb
B:bac
C:cab
*/
#include <iostream>
#include <queue>
using namespace std;
int gg[30][30], mm[30][30];
int a[30], n, ggpre[30], mmpre[30];
queue<int>my;
void stable_marriage()
{
    int i;
    memset(ggpre, 0, sizeof(ggpre)); //gg优先选择.
    memset(mmpre, -1, sizeof(mmpre)); //mm优先选择.
    int pm, pf;
    while (!my.empty())
```

```
    {
        pm = my.front();
        my.pop();
        pf = gg[pm][ggpre[pm]];
        ggpre[pm]++;
        if (mmpre[pf] < 0) mmpre[pf] = pm; //pf是自由的（pm，pf）变成约会状态
        else if (mm[pf][mmpre[pf]] < mm[pf][pm]) //pf更喜欢pm1,pm保持自由.
        {
            my.push(pm);
        }
        else                                //pf更喜欢pm,而不是pm1,(pm,pf)变成约会状
态.
        {
            my.push(mmpre[pf]);
            mmpre[pf] = pm;
        }
    }
    for (i = 0; i < 26; i++)
        if (mmpre[i] > -1) ggpre[mmpre[i]] = i;
    for (i = 0; i < n; i++)
        printf("%c %c\n", a[i] + 'a', ggpre[a[i]] + 'A');
    puts("");
}
int main()
{
    int i, j, t;
    scanf("%d", &t);
    while (t--)
    {
        scanf("%d", &n);
        char temp, str[30];
        while (!my.empty())
            my.pop();
        for (i = 0; i < n; i++)
        {
            scanf(" %c", &temp);
            a[i] = temp - 'a';
            my.push(temp - 'a');
        }
        sort(a, a + n);
        for (i = 0; i < n; i++)
            scanf(" %c", &temp);
        for (i = 0; i < n; i++)
        {
            scanf("%s", str);
```

```
            for (j = 0; j < n; j++)
                gg[str[0] - 'a'][j] = str[j + 2] - 'A';
        }
        for (i = 0; i < n; i++)
        {
            scanf("%s", str);
            for (j = 0; j < n; j++)
                mm[str[0] - 'A'][str[j + 2] - 'a'] = j;
        }
        stable_marriage();
    }
    return 0;
}
```

§ 15.11 网络流

增广路方法的复杂度是通过估计增广次数的上界得到的。对于实际应用中的网络，增广次数往往很少，所以使用范围还是很广的，实用性强。预流推进方法看似比增广路方法在复杂度上快很多，然而实际上，预流推进方法的复杂度的上界是比较紧的。**对于一些稀疏图，预流推进方法的实际效果往往不如增广路方法。**

15.11.1 最大流：Edmonds Karp ($O(V * E^2)$)

无打印路径

```
/**
*最大流: Edmonds Karp ($(V*E^2)$)
*输入: g[][],st=0,ed=1,n(点的个数,编号0-n.n包括了源点和汇点)
*输出: 最大流Edmonds_Karp()
*/
const int maxn=0;
const int inf=0x3f3f3f3f;
int g[maxn][maxn];//存边的容量，没有边的初始化为0
int path[maxn],st,ed;
int n;//点的个数，编号0-n.n包括了源点和汇点。

int bfs()
{
    int i,t;
    queue<int> q;
    int flow[maxn];
    memset(path,-1,sizeof(path));//每次搜索前都把路径初始化成-1
    path[st]=0;
    flow[st]=inf;//源点可以有无穷的流流进
    q.push(st);
    while(!q.empty())
```

```
    {
        t=q.front();
        q.pop();
        if(t==ed)break;
        //枚举所有的点，如果点的编号起始点有变化可以改这里
        for(i=0; i<=n; i++)
        {
            if(i!=st&&path[i]==-1&&g[t][i])
            {
                flow[i]=flow[t]<g[t][i]?flow[t]:g[t][i];
                q.push(i);
                path[i]=t;
            }
        }
    }
    if(path[ed]==-1)return -1;//即找不到汇点上去了。找不到增广路径了
    return flow[ed];
}
int EK(int NdFlow)
{
    int max_flow=0;
    int step,now,pre;
    while((step=bfs())!=-1)
    {
        max_flow+=step;
        now=ed;
        while(now!=st)
        {
            pre=path[now];
            g[pre][now]-=step;
            g[now][pre]+=step;
            now=pre;
        }
        /*如果超过指定流量就return 掉*/
        if(NdFlow==inf) continue;
        if(flow > NdFlow) break;
    }
    return max_flow;
}
```

带打印路径

```
/**
*最大流: Edmonds Karp ($(V*E^2)$)
*输入: cap[][],st=0,ed=1,n(点的个数,编号0-n.n包括了源点和汇点)
```

```
*输出：最大流EK(),flow[][](用于打印路径)
*/
const int maxn=0;
const int inf=0x3f3f3f3f;
int cap[maxn][maxn];//存边的容量，没有边的初始化为0
int flow[maxn][maxn];//记录记录路径
int path[maxn],st,ed;
int n;//点的个数，编号0-n.n包括了源点和汇点。

int bfs()
{
    int i,t;
    int tflow[maxn];
    queue<int> q;
    memset(path,-1,sizeof(path));//每次搜索前都把路径初始化成-1
    path[st]=0;
    tflow[st]=inf;//源点可以有无穷的流流进
    q.push(st);
    while(!q.empty())
    {
        t=q.front();
        q.pop();
        if(t==ed)break;
        //枚举所有的点，如果点的编号起始点有变化可以改这里
        for(i=0; i<=n; i++)
        {
            if(i!=st&&path[i]==-1&&cap[t][i])
            {
                tflow[i]=tflow[t]<cap[t][i]?tflow[t]:cap[t][i];
                q.push(i);
                path[i]=t;
            }
        }
    }
    if(path[ed]==-1)return -1;//即找不到汇点上去了。找不到增广路径了
    return tflow[ed];
}
int EK(int NdFlow)
{
    memset(flow,0,sizeof(flow));
    int max_tflow=0;
    int step,now,pre;
    while((step=bfs())!=-1)
    {
        max_tflow+=step;
```

```
        now=ed;
        while(now!=st)
        {
            pre=path[now];
            cap[pre][now]-=step;
            cap[now][pre]+=step;
            flow[pre][now]+=step;
            flow[now][pre]-=step;
            now=pre;
        }
        /*如果超过指定流量就return 掉*/
        if(NdFlow == inf) continue;
        if(max_tflow > NdFlow) break;
    }
    return max_tflow;
}
```

15.11.2 最大流最小割：加各种优化的Dinic算法($O(V^2E)$)

```
/**
*最大流最小割：加各种优化的Dinic算法($O(V^2E)$)
*输入：图(链式前向星),n(顶点个数,包含源汇),st(源),ed(汇)
*输出：Dinic(NdFlow)(最大流),MinCut()(最小割)(需先求最大流)
*打印路径方法：按反向边(i&1)的flow 找，或者按边的flow找
*/
const int maxn = 0;
const int maxm = 0;
const int inf = 0x3f3f3f3f;
struct DINIC
{
    struct Edge
    {
        int u, v;
        int cap, flow;
        int next;
    } edge[maxm];
    int head[maxn], en; //需初始化
    int n, m, d[maxn], cur[maxn];
    int st, ed;
    bool vis[maxn];
    void init(int _n = 0)
    {
        n = _n;
        memset(head, -1, sizeof(head));
        en = 0;
```

```
}
void addse(int u, int v, int cap, int flow)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].cap = cap;
    edge[en].flow = flow;
    edge[en].next = head[u];
    head[u] = en++;
    cur[u] = head[u];
}
void adde(int u, int v, int cap)
{
    addse(u, v, cap, 0);
    addse(v, u, 0, 0); //注意加反向0 边
}
bool BFS()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    Q.push(st);
    d[st] = 0;
    vis[st] = 1;
    while (!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0 && !vis[v])
            {
                vis[v] = 1;
                Q.push(v);
                d[v] = d[u] + 1;
                if (v == ed) return 1;
            }
        }
    }
    return false;
}
int Aug(int u, int a)
{
    if (u == ed) return a;
```

```
        int aug = 0, delta;
        for (int &i = cur[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0 && d[v] == d[u] + 1)
            {
                delta = Aug(v, min(a, w));
                if (delta)
                {
                    edge[i].flow += delta;
                    edge[i ^ 1].flow -= delta;
                    aug += delta;
                    if (!(a -= delta)) break;
                }
            }
        }
        if (!aug) d[u] = -1;
        return aug;
    }
    int Dinic(int NdFlow)
    {
        int flow = 0;
        while (BFS())
        {
            memcpy(cur, head, sizeof(int) * (n + 1));
            flow += Aug(st, inf);
            /*如果超过指定流量就return 掉*/
            if (NdFlow == inf) continue;
            if (flow > NdFlow) break;
        }
        return flow;
    }
    /*残余网络*/
    void Reduce()
    {
        for (int i = 0; i < en; i++) edge[i].cap -= edge[i].flow;
    }
    /*清空流量*/
    void ClearFlow()
    {
        for (int i = 0; i < en; i++) edge[i].flow = 0;
    }
    /*求最小割*/
    vector<int> MinCut()
```

```
{
    BFS();
    vector<int> ans;
    for (int u = 0; u < n; u++)
    {
        if (!vis[u]) continue;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            if (i & 1) continue; /*忽略反向边*/
            int v = edge[i].v;
            int w = edge[i].cap;
            if (!vis[v] && w > 0) ans.push_back(i);
        }
    }
    return ans;
}
/*判网络流有多解*/
bool no[maxn];
int Stack[maxn], top;
bool dfs(int u, int pre, bool flag)
{
    vis[u] = 1;
    Stack[top++] = u;
    for (int i = head[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].v;
        if (edge[i].cap <= edge[i].flow) continue;
        if (v == pre) continue;
        if (!vis[v])
        {
            if (dfs(v, u, edge[i ^ 1].flow < edge[i ^ 1].cap))
                return true;
        }
        else if (!no[v])return true;
    }
    if (!flag)
    {
        while (1)
        {
            int v = Stack[--top];
            no[v] = true;
            if (v == u)break;
        }
    }
    return false;
```

```
    }
    bool multi()
    {
        memset(vis, 0, sizeof(bool) * (n + 1));
        memset(no, 0, sizeof(bool) * (n + 1));
        return dfs(ed, ed, 0);
    }
} dinic;


/*
另一版本(有时较快，不适合double)
*/
typedef long long LL;
const int maxn = 1010;
const int maxm = 2005010;
const int inf = 0x3f3f3f3f;
int node, s, t, edge;
int to[maxm], flow[maxm], next[maxm];
int head[maxn], work[maxn], dis[maxn], q[maxn];
inline int min(int a, int b)
{
    return a < b ? a : b;
}
inline void init(int nn, int ss, int tt)
{
    node = nn, s = ss, t = tt, edge = 0;
    for (int i = 0; i < node; ++i) head[i] = -1;
}
inline void add(int u, int v, int c1, int c2 = 0)
{
    to[edge] = v, flow[edge] = c1, next[edge] = head[u], head[u] = edge++;
    to[edge] = u, flow[edge] = c2, next[edge] = head[v], head[v] = edge++;
}
bool bfs()
{
    int i, u, v, l, r = 0;
    for (i = 0; i < node; ++i) dis[i] = -1;
    dis[q[r++] = s] = 0;
    for (l = 0; l < r; ++l)
        for (i = head[u = q[l]]; i >= 0; i = next[i])
            if (flow[i] && dis[v = to[i]] < 0)
            {
                dis[q[r++] = v] = dis[u] + 1;
                if (v == t)return 1;
            }
```

```
    return 0;
}
int dfs(int u, int maxf)
{
    if (u == t) return maxf;
    for (int &i = work[u], v, tmp; i >= 0; i = next[i])
        if (flow[i] && dis[v = to[i]] == dis[u] + 1 && (tmp = dfs(v, min(maxf, flow[i]))) > 0)
        {
            flow[i] -= tmp;
            flow[i ^ 1] += tmp;
            return tmp;
        }
    return 0;
}
LL dinic()
{
    int i, delta;
    LL ret = 0;
    while (bfs())
    {
        for (i = 0; i < node; ++i) work[i] = head[i];
        while (delta = dfs(s, inf))ret += delta;
    }
    return ret;
}
```

15.11.3 最大流最小割：加各种优化的ISAP算法($O(V^2E)$)

```
/**
*最大流最小割：加各种优化的ISAP算法($O(V^2E)$)
*输入：图(链式前向星),n(顶点个数,包含源汇),st(源),ed(汇)
*输出：ISAP(NdFlow)(最大流),MinCut()(最小割)(需先求最大流)
*打印路径方法：按反向边(i&1)的flow 找，或者按边的flow找
*/
const int maxn = 0;
const int maxm = 0;
const int inf = 0x3f3f3f3f;
struct ISAP
{
    struct Edge
    {
        int u, v;
        int cap, flow;
        int next;
    } edge[maxm];
```

```
int head[maxn], en;
int st, ed, n;
int d[maxn], p[maxn], num[maxn], cur[maxn];;
bool vis[maxn];
void init(int _n = 0)
{
    n = _n;
    memset(head, -1, sizeof(head));
    en = 0;
}
void addse(int u, int v, int cap, int flow)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].cap = cap;
    edge[en].flow = flow;
    edge[en].next = head[u];
    head[u] = en++;
}
void adde(int u, int v, int cap)
{
    addse(u, v, cap, 0);
    addse(v, u, 0, 0);
}
void bfs()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    d[ed] = 0;
    vis[ed] = 1;
    Q.push(ed);
    while (!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            i ^= 1;
            int v = edge[i].u, cap = edge[i].cap, flow = edge[i].flow;
            if (!vis[v] && cap > flow)
            {
                vis[v] = 1;
                d[v] = d[u] + 1;
                Q.push(v);
            }
```

```
            i ^= 1;
        }
    }
}
int Aug()
{
    int u = ed, a = inf;
    while (u != st)
    {
        a = min(a, edge[p[u]].cap - edge[p[u]].flow);
        u = edge[p[u]].u;
    }
    for (u = ed; u != st; u = edge[p[u]].u)
    {
        edge[p[u]].flow += a;
        edge[p[u] ^ 1].flow -= a;
    }
    return a;
}
int ISAP(int NdFlow)
{
    int flow = 0;
    bfs();
    memset(num, 0, sizeof(num));
    for (int i = 0; i < n; i++) num[d[i]]++;
    memcpy(cur, head, sizeof(int) * (n + 1));
    int u = st;
    while (d[st] < n)
    {
        if (u == ed)
        {
            flow += Aug();
            u = st;
        }
        int ok = 0;
        for (int i = cur[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v, ef = edge[i].flow, cap = edge[i].cap;
            if (d[v] + 1 == d[u] && cap > ef) // Advance
            {
                ok = 1;
                p[v] = i;
                cur[u] = i;
                u = v;
                break;
```

```
            }
        }
        if (!ok) // Retreat
        {
            int tmp = n - 1;
            for (int i = head[u]; i != -1; i = edge[i].next)
                if (edge[i].cap > edge[i].flow)
                    tmp = min(tmp, d[edge[i].v]);
            if (--num[d[u]] == 0) break;
            num[d[u] = tmp + 1]++;
            cur[u] = head[u];
            if (u != st) u = edge[p[u]].u;
        }
        /*如果超过指定流量就return 掉*/
        if (NdFlow == inf) continue;
        if (flow >= NdFlow) break;
    }
    return flow;
}
/*残余网络*/
void Reduce()
{
    for (int i = 0; i < en; i++) edge[i].cap -= edge[i].flow;
}
/*清空流量*/
void ClearFlow()
{
    for (int i = 0; i < en; i++) edge[i].flow = 0;
}
/*求最小割*/
bool bfs2()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    Q.push(st);
    vis[st] = 1;
    while (!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0 && !vis[v])
```

```
                {
                    vis[v] = 1;
                    Q.push(v);
                    if (v == ed) return 1;
                }
            }
        }
        return false;
    }
    vector<int> MinCut()
    {
        bfs2();
        vector<int> ans;
        for (int u = 0; u < n; u++)
        {
            if (!vis[u]) continue;
            for (int i = head[u]; i != -1; i = edge[i].next)
            {
                if (i & 1) continue; /*忽略反向边*/
                int v = edge[i].v;
                int w = edge[i].cap;
                if (!vis[v] && w > 0) ans.push_back(i);
            }
        }
        return ans;
    }
} isap;
```

15.11.4 最大流最小割：加各种优化的HLPP算法($O(V^2\sqrt{E})$)

```
/**
*最大流最小割：加各种优化的HLPP算法($O(V^2\sqrt{E})$)(从1到n)
*输入：图(链式前向星),n(顶点个数,包含源汇),st(源),ed(汇)
*输出:
*打印路径方法：按反向边(i&1)的flow 找，或者按边的flow找
*/
const int maxn = 0;
const int maxm = 0;
const int inf = 0x3f3f3f3f;
struct HLPP
{
    struct Edge
    {
        int u, v;
        int cap, flow;
```

```
    int next;
} edge[maxm];
int head[maxn], en; //需初始化
int n, label_max, st, ed;
int label[maxn], GAP[maxn];
bool vis[maxn];
int in_flow[maxn];
queue<int> active[maxn];
void init(int _n = 0)
{
    n = _n;
    memset(head, -1, sizeof(head));
    en = 0;
}
void addse(int u, int v, int cap, int flow)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].cap = cap;
    edge[en].flow = flow;
    edge[en].next = head[u];
    head[u] = en++;
}
void adde(int u, int v, int cap)
{
    addse(u, v, cap, 0);
    addse(v, u, 0, 0); //注意加反向0 边
}
void bfs()
{
    queue<int> Q;
    for (int i = 0; i <= n; i++) label[i] = n + 1;
    memset(vis, 0, sizeof(vis));
    Q.push(ed);
    label[ed] = 0;
    vis[ed] = 1;
    GAP[0] = 1;
    GAP[n + 1] = n - 1;
    while (!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
```

```
            int w = edge[i].cap - edge[i].flow;
            if (!vis[v]) //不可加w>0
            {
                vis[v] = 1;
                Q.push(v);
                label[v] = label[u] + 1;
                GAP[label[v]]++;
            }
        }
    }
}
void prepare()
{
    for (int i = head[st]; i != -1; i = edge[i].next)
    {
        int v = edge[i].v;
        int w = edge[i].cap - edge[i].flow;
        if (w > 0)
        {
            in_flow[v] += w;
            edge[i].flow += w;
            edge[i ^ 1].flow -= w;
            label_max = max(label_max, label[v]);
            active[label[v]].push(v);
        }
    }
}
void max_flow()
{
    while (label_max)
    {
        if (active[label_max].empty())
        {
            label_max--;
            continue;
        }
        int u = active[label_max].front();
        active[label_max].pop();
        int label_min = n + 1;
        int push_flow;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0)
```

```
            {
                if (label[v] + 1 == label[u])
                {
                    push_flow = min(w, in_flow[u]);
                    edge[i].flow += push_flow;
                    edge[i ^ 1].flow -= push_flow;
                    in_flow[u] -= push_flow;
                    in_flow[v] += push_flow;
                    if (push_flow) active[label[v]].push(v);

                }
            }
            if (edge[i].cap > edge[i].flow)
                label_min = min(label_min, label[v]);
            if (!in_flow[u])
                break;
        }
        if (in_flow[u] && u != ed && label_min < n)
        {
            int tmp = label[u];
            GAP[label[u]]--;
            label[u] = label_min + 1;
            GAP[label[u]]++;
            if (GAP[tmp] == 0)
            {
                for (int i = 1; i <= n; i++)
                    if (label[i] > tmp && label[i] < n + 1)
                    {
                        GAP[label[i]]--;
                        GAP[n + 1]++ ;
                        label[i] = n + 1;
                    }
            }
            active[label[u]].push(u);
            if (label[u] > label_max)
                label_max = label[u];
        }
        //          /*如果超过指定流量就return掉*///此处有问题
        //          if(NdFlow==inf) continue;
        //          if(in_flow[ed]>=NdFlow) break;
    }
}
int HLPP()
{
    memset(in_flow, 0, sizeof(in_flow));
```

```
        for (int i = 0; i < n; i++)
            while (!active[i].empty()) active[i].pop();
        bfs();
        prepare();
        max_flow();
        return in_flow[ed];
    }
} hlpp;
```

15.11.5 贪心预流：用于分层图Dinic预处理(ZOJ 2364)

```
/**
*贪心预流: 用于分层图Dinic预处理(从0到n-1)
*输入: 图(链式前向星),n(顶点个数,包含源汇),st(源),ed(汇),rk[i]=i,level[](分层图的
层次)
*输出: GreedyPreFlow()(预处理Dinic)
*/
const int maxn=0;
const int maxm=0;
const int inf=0x3f3f3f3f;
int in[maxn], out[maxn];
int level[maxn], rk[maxn];
bool cmp(const int &i, const int &j)
{
    return level[i] < level[j];
}
void GreedyPreFlow()
{
    memset(in, 0, sizeof (in));
    memset(out, 0, sizeof (out));
    sort(rk, rk + n, cmp);
    in[st] = inf;
    for (int i = 0; i < n; ++i)
    {
        int u = rk[i];
        for (int j = head[u]; j != -1; j = edge[j].next)
        {
            int v = edge[j].v, w = edge[j].cap - edge[j].flow;
            if (!(j & 1) && in[u] > out[u])
            {
                int f = min(w, in[u] - out[u]);
                in[v] += f, out[u] += f;
            }
        }
    }
```

```
    memset(in, 0, sizeof (in));
    in[ed] = inf;
    for (int i = n - 1; i >= 0; --i)
    {
        int v = rk[i];
        for (int j = head[v]; j != -1; j = edge[j].next)
        {
            int u = edge[j].v, w = edge[j ^ 1].cap - edge[j ^ 1].flow;
            if (j & 1 && out[u] > in[u])
            {
                int f = min(w, min(out[u] - in[u], in[v]));
                in[u] += f, in[v] -= f;
                edge[j].flow -= f, edge[j ^ 1].flow += f;
            }
        }
    }
}
```

示例：ZOJ 2364

```
//#pragma comment(linker, "/STACK:102400000,102400000")
#include<cstdio>
#include<iostream>
#include<cstring>
#include<string>
#include<cmath>
#include<set>
#include<list>
#include<map>
#include<iterator>
#include<cstdlib>
#include<vector>
#include<queue>
#include<stack>
#include<algorithm>
#include<functional>
using namespace std;
typedef long long LL;
#define ROUND(x) round(x)
#define FLOOR(x) floor(x)
#define CEIL(x) ceil(x)
//const int maxn=0;
//const int inf=0x3f3f3f3f;
const LL inf64=0x3f3f3f3f3f3f3f3fLL;
const double INF=1e30;
```

```
const double eps=1e-6;
/**
*最大流最小割：加各种优化的Dinic算法($O(V^2E)$)
*输入：图(链式前向星),n(顶点个数,包含源汇),st(源),ed(汇)
*输出：Dinic(NdFlow)(最大流),MinCut()(最小割)(需先求最大流)
*打印路径方法：按反向边(i&1)的flow 找，或者按边的flow找
*/
const int maxn=1510;
const int maxm=600010;
const int inf=0x3f3f3f3f;
struct Edge
{
    int u,v;
    int cap,flow;
    int next;
} edge[maxm];
int head[maxn],edgeNum;//需初始化
int n,m,d[maxn],cur[maxn];
int st,ed;
bool vis[maxn];
void addSubEdge(int u,int v,int cap,int flow)
{
    edge[edgeNum].u=u;
    edge[edgeNum].v=v;
    edge[edgeNum].cap=cap;
    edge[edgeNum].flow=flow;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
    cur[u]=head[u];
}
void addEdge(int u,int v,int cap)
{
    addSubEdge(u,v,cap,0);
    addSubEdge(v,u,0,0);//注意加反向0 边
}
bool BFS()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    Q.push(st);
    d[st]=0;
    vis[st]=1;
    while (!Q.empty())
    {
        int u=Q.front();
```

```
        Q.pop();
        for(int i=head[u]; i!=-1; i=edge[i].next)
        {
            int v=edge[i].v;
            int w=edge[i].cap-edge[i].flow;
            if(w>0 && !vis[v])
            {
                vis[v]=1;
                Q.push(v);
                d[v]=d[u]+1;
                if(v==ed) return 1;
            }
        }
    }
    return false;
}
int Aug(int u, int a)
{
    if (u==ed) return a;
    int aug=0, delta;
    for(int &i=cur[u]; i!=-1; i=edge[i].next)
    {
        int v=edge[i].v;
        int w=edge[i].cap-edge[i].flow;
        if (w>0 && d[v]==d[u]+1)
        {
            delta = Aug(v, min(a,w));
            if (delta)
            {
                edge[i].flow += delta;
                edge[i^1].flow -= delta;
                aug += delta;
                if (!(a-=delta)) break;
            }
        }
    }
    if (!aug) d[u]=-1;
    return aug;
}
int Dinic(int NdFlow)
{
    int flow=0;
    while (BFS())
    {
        memcpy(cur,head,sizeof(int)*(n+1));
```

```
        flow += Aug(st,inf);
        /*如果超过指定流量就return 掉*/
        if(NdFlow==inf) continue;
        if(flow > NdFlow) break;
    }
    return flow;
}

int in[maxn],out[maxn];
int level[maxn],rk[maxn];
bool cmp(const int &i,const int &j)
{
    return level[i]<level[j];
}
void GreedyPreFlow()
{
    memset(in, 0, sizeof (in));
    memset(out, 0, sizeof (out));
    sort(rk, rk+n, cmp);
    in[st] = inf;
    for (int i = 0; i < n; ++i)
    {
        int u = rk[i];
        for (int j = head[u]; j!=-1; j = edge[j].next)
        {
            int v = edge[j].v, w = edge[j].cap-edge[j].flow;
            if (!(j & 1) && in[u] > out[u])
            {
                int f = min(w, in[u]-out[u]);
                in[v] += f, out[u] += f;
            }
        }
    }
    memset(in, 0, sizeof (in));
    in[ed] = inf;
    for (int i = n-1; i >= 0; --i)
    {
        int v = rk[i];
        for (int j = head[v]; j!=-1; j = edge[j].next)
        {
            int u = edge[j].v, w = edge[j^1].cap-edge[j^1].flow;
            if (j & 1 && out[u] > in[u])
            {
                int f = min(w, min(out[u]-in[u], in[v]));
                in[u] += f, in[v] -= f;
```

```
                edge[j].flow -= f, edge[j^1].flow += f;
            }
        }
    }
}

int N,M,L;
void init()
{
    memset(head,-1,sizeof(head));
    edgeNum=0;
}
void input()
{
    scanf("%d%d%d",&N,&M,&L);
    int x=0;
    int idx=0;
    st=0,ed=0;
    for(int i=0; i<N; i++)
    {
        scanf("%d",&x);
        rk[i]=i;
        level[i]=x;
        if(x==1) st=i;
        if(x==L) ed=i;
    }
    for(int i=0; i<M; i++)
    {
        int u,v,w;
        scanf("%d%d%d",&u,&v,&w);
        u--,v--;
        addEdge(u,v,w);
    }
    n=N;
}
void solve()
{
    GreedyPreFlow();
    Dinic(inf);
//    cout<<HLPP()<<endl;
//    cout<<Dinic(inf)<<endl;
    for(int i=0; i<edgeNum; i+=2) printf("%d\n",edge[i].flow);
}
void output()
{
```

```
    //
}
int main()
{
//    std::ios_base::sync_with_stdio(false);
//    freopen("in.cpp","r",stdin);
    int T;
    scanf("%d",&T);
    while(T--)
    {
        init();
        input();
        solve();
        output();
    }
    return 0;
}
```

15.11.6 有上下界的网络流

无源汇最大流

原理 上界用c_i表示，下界用b_i表示。

下界是必须流满的，那么对于每一条边，去掉下界后，其自由流为$c_i b_i$。

主要思想：每一个点流进来的流=流出去的流

对于每一个点i，令

1. M_i=sum(i点所有流进来的下界流)sum(i点所有流出去的下界流)
2. 新建源点S、汇点T
   - 如果$M_i > 0$，代表此点必须还要流出去M_i的自由流，那么我们从源点连一条M_i的边到该点。
   - 如果$M_i < 0$，代表此点必须还要流进来M_i的自由流，那么我们从该点连一条M_i的边到汇点。
3. 求$S->T$的最大流，看是否满流(S的相邻边都流满)。满流则有解，否则无解。

例：SGU 194 Reactor Cooling

```
/**
*给n个点，及m根pipe，每根pipe用来流躺液体的，单向
*每时每刻每根pipe流进来的物质要等于流出去的物质，要使得m条pipe组成一个循环体，里面
流躺物质
*并且满足每根pipe一定的流量限制，范围为[Li,Ri]
*即要满足每时刻流进来的不能超过Ri(最大流问题)，同时最小不能低于Li
*/
//#pragma comment(linker, "/STACK:102400000,102400000")
```

```
#include<cstdio>
#include<iostream>
#include<cstring>
#include<string>
#include<cmath>
#include<set>
#include<list>
#include<map>
#include<iterator>
#include<cstdlib>
#include<vector>
#include<queue>
#include<stack>
#include<algorithm>
#include<functional>
using namespace std;
typedef long long LL;
#define ROUND(x) round(x)
#define FLOOR(x) floor(x)
#define CEIL(x) ceil(x)
const int maxn=210;
const int maxm=2*210*210;
const int inf=0x3f3f3f3f;
const LL inf64=0x3f3f3f3f3f3f3f3fLL;
const double INF=1e30;
const double eps=1e-6;

/**
*最大流最小割：加各种优化的Dinic算法($O(V^2E)$)
*输入：图(链式前向星),n(顶点个数,包含源汇),st(源),ed(汇)
*输出：Dinic(NdFlow)(最大流),MinCut()(最小割)(需先求最大流)
*打印路径方法：按反向边(i&1)的flow 找，或者按边的flow找
*/
//const int maxn=0;
//const int maxm=0;
//const int inf=0x3f3f3f3f;
struct Edge
{
    int u,v;
    int cap,flow;
    int next;
} edge[maxm];
int head[maxn],edgeNum;//需初始化
int n,m,d[maxn],cur[maxn];
int st,ed;
```

```
bool vis[maxn];
void addSubEdge(int u,int v,int cap,int flow)
{
    edge[edgeNum].u=u;
    edge[edgeNum].v=v;
    edge[edgeNum].cap=cap;
    edge[edgeNum].flow=flow;
    edge[edgeNum].next=head[u];
    head[u]=edgeNum++;
    cur[u]=head[u];
}
void addEdge(int u,int v,int cap)
{
    addSubEdge(u,v,cap,0);
    addSubEdge(v,u,0,0);//注意加反向0 边
}
bool BFS()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    Q.push(st);
    d[st]=0;
    vis[st]=1;
    while (!Q.empty())
    {
        int u=Q.front();
        Q.pop();
        for(int i=head[u]; i!=-1; i=edge[i].next)
        {
            int v=edge[i].v;
            int w=edge[i].cap-edge[i].flow;
            if(w>0 && !vis[v])
            {
                vis[v]=1;
                Q.push(v);
                d[v]=d[u]+1;
                if(v==ed) return 1;
            }
        }
    }
    return false;
}
int Aug(int u, int a)
{
    if (u==ed) return a;
```

```
    int aug=0, delta;
    for(int &i=cur[u]; i!=-1; i=edge[i].next)
    {
        int v=edge[i].v;
        int w=edge[i].cap-edge[i].flow;
        if (w>0 && d[v]==d[u]+1)
        {
            delta = Aug(v, min(a,w));
            if (delta)
            {
                edge[i].flow += delta;
                edge[i^1].flow -= delta;
                aug += delta;
                if (!(a-=delta)) break;
            }
        }
    }
    if (!aug) d[u]=-1;
    return aug;
}
int Dinic(int NdFlow)
{
    int flow=0;
    while (BFS())
    {
        memcpy(cur,head,sizeof(int)*(n+1));
        flow += Aug(st,inf);
        /*如果超过指定流量就return 掉*/
        if(NdFlow==inf) continue;
        if(flow > NdFlow) break;
    }
    return flow;
}

int N,M;
int in[maxn],out[maxn];
int p[maxm],q[maxm];
void init()
{
    memset(head,-1,sizeof(head));
    memset(in,0,sizeof(in));
    memset(out,0,sizeof(out));
    edgeNum=0;
}
void input()
```

```
{
    scanf("%d%d",&N,&M);
    for(int i=0;i<M;i++)
    {
        int u,v;
        scanf("%d%d%d%d",&u,&v,&p[i],&q[i]);
        addEdge(u,v,q[i]-p[i]);
        out[u]+=p[i];
        in[v]+=p[i];
    }
}
void solve()
{
    int ans=0;
    st=0,ed=N+1,n=N+2;
    for(int i=1;i<=N;i++)
    {
        int t=in[i]-out[i];
        if(t>0)
        {
            addEdge(st,i,t);
            ans+=t;
        }
        else if(t<0) addEdge(i,ed,-t);
    }

    if(Dinic(inf)!=ans)
    {
        puts("NO");
        return;
    }
    puts("YES");
    for(int i=0;i<M;i++)
    {
//          cout<<edge[2*i].flow<<endl;
        printf("%d\n",edge[2*i].flow+p[i]);
    }
}
void output()
{
    //
}
int main()
{
//    std::ios_base::sync_with_stdio(false);
```

```
//    freopen("in.cpp","r",stdin);
    int T;
    scanf("%d",&T);
    while(T--)
    {
        init();
        input();
        solve();
        output();
    }
    return 0;
}
```

有源汇最大流

原理 满足所有下界的情况下，判断是否存在可行流，方法可以转化成上面无源汇上下界判断方法：

- 只要连一条$T->S$的边，流量为无穷，没有下界，那么原图就得到一个无源汇的循环流图。
- 原图中的边的流量设成自由流量c_ib_i
- 新建源点SS汇点TT，求M_i，连边
- 然后求$SS->TT$最大流，判是否满流。
- 判定有解之后然后求最大流，信息都在上面求得的残留网络里面。
- 满足所有下界时，从$S->T$的流量为后悔边$S->T$的边权！然后在残留网络中$S->T$可能还有些自由流没有流满，再做一次$S->T$ 的最大流，所得到的最大流就是原问题的最大流(内含两部分：残留的自由流所得到的流+后悔边$S->T$)。

例：ZOJ 3229 Shoot the Bullet

有源汇最小流

1. 同样先转换为无源汇网络流问题
2. 先不加$T->S$边权为无穷的边，求$SS->TT$的最大流
3. 如果还没有流满则再加$T->S$边权为无穷的边，再求一次最大流。得到后悔边$S->T$就是原问题的最小流了。

15.11.7 最小(大)费用最大流：SPFA增广路($O(w*O(SPFA))$)

ZKW版

```
/**
*ZKW最小费用最大流
```

```
*适用于最终流量较大，而费用取值范围不大的图，或者是增广路径比较短的图(如二分图),
zkw算法都会比较快
*/
struct MaxFlow
{
    int size, n;
    int st, en, maxflow, mincost;
    bool vis[maxn];
    int net[maxn], pre[maxn], cur[maxn], dis[maxn];
    std::queue <int> Q;
    struct EDGE
    {
        int v, cap, cost, next;
        EDGE() {}
        EDGE(int a, int b, int c, int d)
        {
            v = a, cap = b, cost = c, next = d;
        }
    } E[maxm << 1];
    void init(int _n)
    {
        n = _n, size = 0;
        memset(net, -1, sizeof(net));
    }
    void add(int u, int v, int cap, int cost)
    {
        E[size] = EDGE(v, cap, cost, net[u]);
        net[u] = size++;
        E[size] = EDGE(u, 0, -cost, net[v]);
        net[v] = size++;
    }
    bool adjust()
    {
        int v, min = inf;
        for (int i = 0; i <= n; i++)
        {
            if (!vis[i]) continue;
            for (int j = net[i]; v = E[j].v, j != -1; j = E[j].next)
                if (E[j].cap)
                    if (!vis[v] && dis[v] - dis[i] + E[j].cost < min)
                        min = dis[v] - dis[i] + E[j].cost;
        }
        if (min == inf) return false;
        for (int i = 0; i <= n; i++)
            if (vis[i])
```

```
            cur[i] = net[i], vis[i] = false, dis[i] += min;
        return true;
    }
    int augment(int i, int flow)
    {
        if (i == en)
        {
            mincost += dis[st] * flow;
            maxflow += flow;
            return flow;
        }
        vis[i] = true;
        for (int j = cur[i], v; v = E[j].v, j != -1; j = E[j].next)
        {
            if (!E[j].cap) continue;
            if (vis[v] || dis[v] + E[j].cost != dis[i]) continue;
            int delta = augment(v, std::min(flow, E[j].cap));
            if (delta)
            {
                E[j].cap -= delta;
                E[j ^ 1].cap += delta;
                cur[i] = j;
                return delta;
            }
        }
        return 0;
    }
    void spfa()
    {
        int u, v;
        for (int i = 0; i <= n; i++)
            vis[i] = false, dis[i] = inf;
        dis[st] = 0;
        Q.push(st);
        vis[st] = true;
        while (!Q.empty())
        {
            u = Q.front(), Q.pop();
            vis[u] = false;
            for (int i = net[u]; v = E[i].v, i != -1; i = E[i].next)
            {
                if (!E[i].cap || dis[v] <= dis[u] + E[i].cost)
                    continue;
                dis[v] = dis[u] + E[i].cost;
                if (!vis[v])
```

```
                {
                    vis[v] = true;
                    Q.push(v);
                }
            }
        }
        for (int i = 0; i <= n; i++)
            dis[i] = dis[en] - dis[i];
    }
    int zkw(int s, int t, int need)
    {
        st = s, en = t;
        spfa();
        mincost = maxflow = 0;
        for (int i = 0; i <= n; i++)
            vis[i] = false, cur[i] = net[i];
        do
        {
            while (augment(st, inf))
                memset(vis, false, sizeof(vis));
        }
        while (adjust());
        if (maxflow < need) return -1;
        return mincost;
    }
} zkw;
```

普通版

```
/**
*最小(大)费用最大流: SPFA增广路($O(w*O(SPFA))$)
*最大费用: 费用取反addEdge(,,,-cost);
*输入: 图(链式前向星),n(顶点个数,包含源汇),s(源),t(汇)
*输出: minCostMaxflow(int s, int t, int &cost)返回流量, cost为费用
*打印路径方法: 按反向边(i&1)的flow 找, 或者按边的flow找
*/
const int maxn = 0;
const int maxm = 0;
const int inf = 0x3f3f3f3f;
struct Edge
{
    int u, v;
    int cap, flow;
    int cost;
    int next;
```

```
} edge[maxm];
int head[maxn], en; //需初始化
int n, m;
bool vis[maxn];
int pre[maxn], dis[maxn];
void addse(int u, int v, int cap, int flow, int cost)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].cap = cap;
    edge[en].flow = flow;
    edge[en].cost = cost;
    edge[en].next = head[u];
    head[u] = en++;
}
void adde(int u, int v, int cap, int cost)
{
    addse(u, v, cap, 0, cost);
    addse(v, u, 0, 0, -cost); //注意加反向0 边
}
bool spfa(int s, int t)
{
    queue<int>q;
    for (int i = 0; i < n; i++)
    {
        dis[i] = inf;
        vis[i] = false;
        pre[i] = -1;
    }
    dis[s] = 0;
    vis[s] = true;
    q.push(s);
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        vis[u] = false;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            if (edge[i].cap > edge[i].flow &&
                    dis[v] > dis[u] + edge[i].cost )
            {
                dis[v] = dis[u] + edge[i].cost;
                pre[v] = i;
```

```
                if (!vis[v])
                {
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
    }
    if (pre[t] == -1)return false;
    else return true;
}
int minCostMaxflow(int s, int t, int &cost)//返回流量，cost为费用
{
    int flow = 0;
    cost = 0;
    while (spfa(s, t))
    {
        int Min = inf;
        for (int i = pre[t]; i != -1; i = pre[edge[i ^ 1].v])
        {
            if (Min > edge[i].cap - edge[i].flow)
                Min = edge[i].cap - edge[i].flow;
        }
        for (int i = pre[t]; i != -1; i = pre[edge[i ^ 1].v])
        {
            edge[i].flow += Min;
            edge[i ^ 1].flow -= Min;
            cost += edge[i].cost * Min;
        }
        flow += Min;
    }
    return flow;
}
```

15.11.8 判网络流有多解

方法1(较慢)： 如果残余网络里有长度大于2的环，则网络流有多解。

方法2(很快)： 一种找环比较正确，而且快速的方法：和Tarjan的思路差不多。就是从汇点开始去dfs, 记录哪些点是回不到end的。然后就一遍dfs就可以解决了。

```
bool vis[maxn], no[maxn];
int Stack[maxn], top;
bool dfs(int u, int pre, bool flag)
{
    vis[u] = 1;
```

```
    Stack[top++] = u;
    for (int i = head[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].v;
        if (edge[i].cap <= edge[i].flow) continue;
        if (v == pre) continue;
        if (!vis[v])
        {
            if (dfs(v, u, edge[i ^ 1].flow < edge[i ^ 1].cap))
                return true;
        }
        else if (!no[v])return true;
    }
    if (!flag)
    {
        while (1)
        {
            int v = Stack[--top];
            no[v] = true;
            if (v == u)break;
        }
    }
    return false;
}
```

例：HDU 4975

```
/*
HDU 4975 A simple Gaussian elimination problem.
*/
#include <cstdio>
#include <iostream>
#include <cstring>
#include <string>
#include <queue>
#include <ctime>
#include <cstdlib>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
const int maxn = 1010;
const int maxm = 600010;
const int inf = 0x3f3f3f3f;
struct DINIC
{
```

```
struct Edge
{
    int u, v;
    int cap, flow;
    int next;
} edge[maxm];
int head[maxn], en;
int n, m, d[maxn], cur[maxn];
int st, ed;
bool vis[maxn];
void init(int _n = 0)
{
    n = _n;
    memset(head, -1, sizeof(int) * (n + 1));
    en = 0;
}
void addse(int u, int v, int cap, int flow)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].cap = cap;
    edge[en].flow = flow;
    edge[en].next = head[u];
    head[u] = en++;
    cur[u] = head[u];
}
void adde(int u, int v, int cap)
{
    addse(u, v, cap, 0);
    addse(v, u, 0, 0);
}
bool BFS()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    Q.push(st);
    d[st] = 0;
    vis[st] = 1;
    while (!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
```

```
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0 && !vis[v])
            {
                vis[v] = 1;
                Q.push(v);
                d[v] = d[u] + 1;
                if (v == ed) return 1;
            }
        }
    }
    return false;
}
int Aug(int u, int a)
{
    if (u == ed) return a;
    int aug = 0, delta;
    for (int &i = cur[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].v;
        int w = edge[i].cap - edge[i].flow;
        if (w > 0 && d[v] == d[u] + 1)
        {
            delta = Aug(v, min(a, w));
            if (delta)
            {
                edge[i].flow += delta;
                edge[i ^ 1].flow -= delta;
                aug += delta;
                if (!(a -= delta)) break;
            }
        }
    }
    if (!aug) d[u] = -1;
    return aug;
}
int Dinic(int NdFlow)
{
    int flow = 0;
    while (BFS())
    {
        memcpy(cur, head, sizeof(int) * (n + 1));
        flow += Aug(st, inf);
        if (NdFlow == inf) continue;
        if (flow > NdFlow) break;
```

```
        }
        return flow;
    }

    bool no[maxn];
    int Stack[maxn], top;
    bool dfs(int u, int pre, bool flag)
    {
        vis[u] = 1;
        Stack[top++] = u;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            if (edge[i].cap <= edge[i].flow) continue;
            if (v == pre) continue;
            if (!vis[v])
            {
                if (dfs(v, u, edge[i ^ 1].flow < edge[i ^ 1].cap))
                    return true;
            }
            else if (!no[v])return true;
        }
        if (!flag)
        {
            while (1)
            {
                int v = Stack[--top];
                no[v] = true;
                if (v == u)break;
            }
        }
        return false;
    }
    bool multi()
    {
        memset(vis, 0, sizeof(bool) * (n + 1));
        memset(no, 0, sizeof(bool) * (n + 1));
        return dfs(ed, ed, 0);
    }
} dinic;
int kase;
int n, m;
int a[510], b[510];
int sum;
inline int read()
```

```
{
    bool flag = 0;
    char ch = getchar();
    int data = 0;
    while (ch < '0' || ch > '9')
    {
        if (ch == '-') flag = 1;
        ch = getchar();
    }
    do
    {
        data = data * 10 + ch - '0';
        ch = getchar();
    }
    while (ch >= '0' && ch <= '9');
    return flag ? -data : data;
}
void init()
{
    kase++;
}
void input()
{
    n = read();
    m = read();
    sum = 0;
    for (int i = 0; i < n; i++)
    {
        a[i] = read();
        sum += a[i];
    }
    for (int i = 0; i < m; i++)
        b[i] = read();
}
void build()
{
    dinic.init(n + m + 2);
    dinic.st = 0;
    dinic.ed = 1;
    for (int i = 0; i < n; i++)
        dinic.adde(dinic.st, i + 2, a[i]);
    for (int i = 0; i < m; i++)
        dinic.adde(i + n + 2, dinic.ed, b[i]);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
```

```
            dinic.adde(i + 2, j + n + 2, 9);
}
void solve()
{
    build();
    int flow = dinic.Dinic(inf);
    if (flow < sum)
    {
        printf("Case #%d: So naive!\n", kase);
        return;
    }
    if (dinic.multi())
    {
        printf("Case #%d: So young!\n", kase);
        return;
    }
    printf("Case #%d: So simple!\n", kase);
}
void output()
{
    //
}
int main()
{
#ifdef xysmlx
    freopen("in.cpp", "r", stdin);
#endif

    int T;
    kase = 0;
    T = read();
    while (T--)
    {
        init();
        input();
        solve();
        output();
    }
    return 0;
}
```

15.11.9 最大权闭合子图

闭合图 有向图的点集，集合中的点的出边都指向点集内部的点，$(u,v)\in E$则当u成立时v成立(即：u蕴含$v(u->v)$)。

最大权闭合子图 点权之和最大的闭合图。

建图 每一条有向边变为容量为∞，源S到正权点v($w_v > 0$)的边容量w_v，负权点v($w_v < 0$)到汇T的边容量$-w_v$，零权点v($w_v = 0$)不与源和汇相连。然后求最小割(SUM-最大流)即为答案。

例：SPOJ PROFIT Maximum Profit

```
/*
n个中转站，每个站建立花费Xi；m个客户，每个客户需要中转站Ai,Bi，获得收益为Ci，问最大
收益
S向客户连边(S,i,Ci)
站向T连边(i,T,Xi)
客户向站连边(i,j,inf)
答案为sum-dinic()
*/
#include <cstdio>
#include <iostream>
#include <cstring>
#include <string>
#include <cmath>
#include <set>
#include <list>
#include <map>
#include <iterator>
#include <cstdlib>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <functional>
using namespace std;
const int maxn = 60010;
const int maxm = 2000010;
const int inf = 0x3f3f3f3f;

struct DINIC
{
    struct Edge
    {
        int u, v;
        int cap, flow;
        int next;
    } edge[maxm];
    int head[maxn], en; //需初始化
    int n, m, d[maxn], cur[maxn];
    int st, ed;
```

```
bool vis[maxn];
void init(int _n = 0)
{
    n = _n;
    memset(head, -1, sizeof(head));
    en = 0;
}
void addse(int u, int v, int cap, int flow)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].cap = cap;
    edge[en].flow = flow;
    edge[en].next = head[u];
    head[u] = en++;
    cur[u] = head[u];
}
void adde(int u, int v, int cap)
{
    addse(u, v, cap, 0);
    addse(v, u, 0, 0); //注意加反向0 边
}
bool BFS()
{
    queue<int> Q;
    memset(vis, 0, sizeof(vis));
    Q.push(st);
    d[st] = 0;
    vis[st] = 1;
    while (!Q.empty())
    {
        int u = Q.front();
        Q.pop();
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0 && !vis[v])
            {
                vis[v] = 1;
                Q.push(v);
                d[v] = d[u] + 1;
                if (v == ed) return 1;
            }
        }
```

```
        }
        return false;
    }
    int Aug(int u, int a)
    {
        if (u == ed) return a;
        int aug = 0, delta;
        for (int &i = cur[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            int w = edge[i].cap - edge[i].flow;
            if (w > 0 && d[v] == d[u] + 1)
            {
                delta = Aug(v, min(a, w));
                if (delta)
                {
                    edge[i].flow += delta;
                    edge[i ^ 1].flow -= delta;
                    aug += delta;
                    if (!(a -= delta)) break;
                }
            }
        }
        if (!aug) d[u] = -1;
        return aug;
    }
    int Dinic(int NdFlow)
    {
        int flow = 0;
        while (BFS())
        {
            memcpy(cur, head, sizeof(int) * (n + 1));
            flow += Aug(st, inf);
            /*如果超过指定流量就return 掉*/
            if (NdFlow == inf) continue;
            if (flow > NdFlow) break;
        }
        return flow;
    }
} dinic;

int kase;
int n, m;
int sum;
void init()
```

```
{
    kase++;
    sum = 0;
}
void input()
{
    scanf("%d%d", &n, &m);
}
void debug()
{
    //
}
void build()
{
    dinic.init(n + m + 2);
    dinic.st = 0;
    dinic.ed = 1;
    for (int i = 0; i < n; i++)
    {
        int x;
        scanf("%d", &x);
        dinic.adde(i + 2, dinic.ed, x);
    }
    for (int i = 0 + n + 2; i < m + n + 2; i++)
    {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        sum += w;
        dinic.adde(dinic.st, i, w);
        dinic.adde(i, u + 1, inf);
        dinic.adde(i, v + 1, inf);
    }
}
void solve()
{
    build();
    printf("%d\n", sum - dinic.Dinic(inf));
}
void output()
{
    //
}
int main()
{
#ifdef xysmlx
```

```
    freopen("in.cpp", "r", stdin);
#endif

    kase = 0;
    int T;
    scanf("%d", &T);
    while (T--)
    {
        init();
        input();
        solve();
        output();
    }
    return 0;
}
```

15.11.10 最大密度子图

密度 定义一个无向图$G = (V, E)$的密度(density)D为该图的边数$|E|$与该图的点数$|V|$的比值$D = |E|/|V|$。

最大密度子图 密度最大的子图。

方法

- 二分：$L = 0, R = U$(U为边的数量$|E|$)，二分g，($h(g) = |E'| - g * |V'|$)，用网络流判$h(g) > 0$。精度$eps = 1/n^2$

  ```
  if(h(g)>0) L=M;
  else R=M;
  ```

- 网络流：
  - 添加源汇S, T
  - 对原图边(u, v)，插入边$(u, v, 1)$和$(v, u, 1)$
  - 对任意顶点v，插入边(S, v, U)和$(v, T, U + 2 * g - d_v)$；(d_v为原图节点v的度数)。
  - 跑最大流，返回$(U * n - flow)/2$(即：$h(g)$)。

边带权拓广 设边权为w_e(非负)

- 二分精度eps减小，U为边权之和
- 网络流：
  - 添加源汇S, T
  - 对原图边(u, v)，插入边(u, v, w_e)和(v, u, w_e)

  - 对任意顶点v，插入边(S, v, U)和$(v, T, U + 2 * g - d_v)$；(d_v为连接原图节点v的边权之和)。
  - 跑最大流，返回$(U * n - flow)/2$(即：$h(g)$)。

点边带权拓广 设边权为w_e(非负)，点权为p_v(实数)

- 二分精度eps减小，U为边权之和与点权绝对值之和的加和($U = \sum w_e + \sum |p_v|$)。
- 网络流：
  - 添加源汇S, T
  - 对原图边(u, v)，插入边(u, v, w_e)和(v, u, w_e)
  - 对任意顶点v，插入边(S, v, U)和$(v, T, U + 2 * g - d_v)$；(d_v为连接原图节点v的边权之和$+2 * p_v$)。
  - 跑最大流，返回$(U * n - flow)/2$(即：$h(g)$)。

例：POJ 3155 Hard Life(不带权，输出点集)

```
double h(double g)
{
    dinic.init(n + 2);
    dinic.st = n;
    dinic.ed = n + 1;
    for (int i = 0; i < n; i++)
    {
        dinic.adde(dinic.st, i, m);
        dinic.adde(i, dinic.ed, m + 2 * g - deg[i]);
    }
    for (int i = 0; i < m; i++)
    {
        dinic.adde(e[i].u, e[i].v, 1);
        dinic.adde(e[i].v, e[i].u, 1);
    }
    return ((double)n * (double)m - dinic.Dinic(INF)) * 0.5;
}
void solve()
{
    double L = 0, R = m;
    double ee = 1.0 / n / n;
    while (R - L >= ee)
    {
        double M = (R + L) * 0.5;
        if (h(M) > eps) L = M;
        else R = M;
    }
    h(L);
```

```
    dinic.bfsx();
    vector<int> ans;
    for (int i = 0; i < n; i++)
    {
        if (dinic.vis[i]) ans.pb(i + 1);
    }
    if (ans.size() == 0) ans.pb(1);
    sort(ans.begin(), ans.end());
    printf("%d\n", ans.size());
    for (int i = 0; i < ans.size(); i++)
        printf("%d\n", ans[i]);
}
```

15.11.11 混合图(有向+无向)的欧拉路径

见Euler路径部分

15.11.12 二分图最小点权覆盖

描述 从x或者y集合中选取一些点，使这些点覆盖所有的边，并且选出来的点的权值尽可能小。

建模 原二分图中的边(u, v)替换为容量为∞的有向边(u, v, ∞)，设立源点s和汇点t，将s和x集合中的点相连，容量为该点的权值；将y中的点同t相连，容量为该点的权值。在新图上求最大流，最大流量即为最小点权覆盖的权值和。

15.11.13 二分图最大点权独立集

描述 在二分图中找到权值和最大的点集，使得它们之间两两没有边。

建模 其实它是最小点权覆盖的对偶问题。答案=总权值−最小点覆盖集。

例：HDU 1569

```
/*
一个m*n的棋盘，每个格子都有一个权值，从中取出某些数，使得任意两个数所在的格子没有公
共边，
并且所取去出的数和最大。求这个最大的值。

因为这个数据比较大，所以用动态规划会超时。
将格子染色成二分图，显然是求二分图的最大点权独立集。
将图转换成黑白棋盘问题，i + j 为奇数的与s节点相连，边的权值为棋盘上对应位置的值，
其他的与t节点相连，边的权值为棋盘上对应位置的值，
然后让棋盘上相邻之间的节点用边相连，边的权值为INF。这样问题就转换为了最大点权独立集
问题。
将问题转化为二分图最小点权覆盖来求解，最终结果=总权和-最大流。

最大点权独立集：
转化为最小点权覆盖问题，最大点权独立集=总权值-最小点权覆盖集
```

```
最小点权覆盖:
设立源点s和t，s连边到点i，容量为i点的权值；点j连边到t，容量为j点权值；
原二分图中的边容量为INF，求最大流即为最小点权覆盖。
*/
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
const int INF = 0x7fffffff;
const int maxv = 2600;
const int maxe = 1000000;
int n, m;
int g[55][55];
struct Edge
{
    int v;
    int next;
    int flow;
};
Edge e[maxe];
int head[maxv], edgeNum;
int now[maxv], d[maxv], vh[maxv], pre[maxv], preh[maxv];
int start, end;

void addEdge(int a, int b, int c)
{
    e[edgeNum].v = b;
    e[edgeNum].flow = c;
    e[edgeNum].next = head[a];
    head[a] = edgeNum++;
    e[edgeNum].v = a;
    e[edgeNum].flow = 0;
    e[edgeNum].next = head[b];
    head[b] = edgeNum++;
}

void Init()
{
    edgeNum = 0;
    memset(head, -1, sizeof(head));
    memset(d, 0, sizeof(d));
}

int sap(int s, int t, int n)     //源点，汇点，结点总数
{
```

```
int i, x, y;
int f, ans = 0;
for (i = 0; i < n; i++)
    now[i] = head[i];
vh[0] = n;
x = s;
while (d[s] < n)
{
    for (i = now[x]; i != -1; i = e[i].next)
        if (e[i].flow > 0 && d[y = e[i].v] + 1 == d[x])
            break;
    if (i != -1)
    {
        now[x] = preh[y] = i;
        pre[y] = x;
        if ((x = y) == t)
        {
            for (f = INF, i = t; i != s; i = pre[i])
                if (e[preh[i]].flow < f)
                    f = e[preh[i]].flow;
            ans += f;
            do
            {
                e[preh[x]].flow -= f;
                e[preh[x] ^ 1].flow += f;
                x = pre[x];
            }
            while (x != s);
        }
    }
    else
    {
        if (!--vh[d[x]])
            break;
        d[x] = n;
        for (i = now[x] = head[x]; i != -1; i = e[i].next)
        {
            if (e[i].flow > 0 && d[x] > d[e[i].v] + 1)
            {
                now[x] = i;
                d[x] = d[e[i].v] + 1;
            }
        }
        ++vh[d[x]];
        if (x != s)
```

```
                x = pre[x];
        }
    }
    return ans;
}


void build()
{
    int i, j;
    for (i = 1; i <= m; i++)
    {
        for (j = 1; j <= n; j++)
        {
            int t = (i - 1) * n + j;
            if ((i + j) % 2)
            {
                addEdge(start, t, g[i][j]);
                if (i > 1)
                    addEdge(t, t - n, INF);
                if (i < m)
                    addEdge(t, t + n, INF);
                if (j > 1)
                    addEdge(t, t - 1, INF);
                if (j < n)
                    addEdge(t, t + 1, INF);
            }
            else
                addEdge(t, end, g[i][j]);
        }
    }
}

int main()
{
    int i, j;
    int result;
    while (scanf("%d %d", &m, &n) != EOF)
    {
        result = 0;
        Init();
        for (i = 1; i <= m; i++)
        {
            for (j = 1; j <= n; j++)
            {
```

```
                scanf("%d", &g[i][j]);
                result += g[i][j];
            }
        }
        start = 0;
        end = n * m + 1;
        build();
        printf("%d\n", result - sap(start, end, end + 1));
    }
    return 0;
}
```

15.11.14 最小K路径覆盖

```
/**
HDU 4862 Jump 最小K路径覆盖
题意:
给你一个n*m的矩阵，填充着0-9的数字，每次能从一个点出发，到它的右边或者下边的点，
花费为|x1-x2|+|y1-y2|-1,
如果跳跃的起点和终点的数字相同，则获得这个数字的收益，不能走已经走过的点
有K次重新选择起点的机会
如果可以走遍所有点，则输出最大的价值(收益-花费)
否则，输出-1

方法:
最小K路径覆盖，最小费用最大流
建图:
每个点拆为2点：X部和Y部，(a,b)表示流量a，费用b
源点与X部每个点连(1,0)的边
Y部每个点与汇点连(1,0)的边
X部的点如果可以到Y部的点，则连(1,花费-收益)的边
源点与一个新点连(k,0)的边，新点与Y部每个点连(1,0)的边
结果:
如果满流，则输出0-费用
否则，输出-1
*/
// #pragma comment(linker, "/STACK:102400000,102400000")
#include <cstdio>
#include <iostream>
#include <cstring>
#include <string>
#include <cmath>
#include <set>
#include <list>
#include <map>
```

```
#include <iterator>
#include <cstdlib>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <functional>
using namespace std;
typedef long long LL;
#define ROUND(x) round(x)
#define FLOOR(x) floor(x)
#define CEIL(x) ceil(x)
// const int maxn = 210;
// const int maxm = 200010;
// const int inf = 0x3f3f3f3f;
const LL inf64 = 0x3f3f3f3f3f3f3f3fLL;
const double INF = 1e30;
const double eps = 1e-6;
const int P[4] = {0, 0, -1, 1};
const int Q[4] = {1, -1, 0, 0};
const int PP[8] = { -1, -1, -1, 0, 0, 1, 1, 1};
const int QQ[8] = { -1, 0, 1, -1, 1, -1, 0, 1};

/**
*最小(大)费用最大流: SPFA增广路($O(w*O(SPFA))$)
*最大费用: 费用取反addEdge(,,,-cost);
*输入: 图(链式前向星),n(顶点个数,包含源汇),s(源),t(汇)
*输出: minCostMaxflow(int s, int t, int &cost)返回流量, cost为费用
*打印路径方法: 按反向边(i&1)的flow 找, 或者按边的flow找
*/
const int maxn = 210;
const int maxm = 200010;
const int inf = 0x3f3f3f3f;
struct Edge
{
    int u, v;
    int cap, flow;
    int cost;
    int next;
} edge[maxm];
int head[maxn], en; //需初始化
int n, m;
int st, ed;
bool vis[maxn];
int pre[maxn], dis[maxn];
```

```
void addse(int u, int v, int cap, int flow, int cost)
{
    edge[en].u = u;
    edge[en].v = v;
    edge[en].cap = cap;
    edge[en].flow = flow;
    edge[en].cost = cost;
    edge[en].next = head[u];
    head[u] = en++;
}
void adde(int u, int v, int cap, int cost)
{
    addse(u, v, cap, 0, cost);
    addse(v, u, 0, 0, -cost); //注意加反向0 边
}
bool spfa(int s, int t)
{
    queue<int>q;
    for (int i = 0; i < n; i++)
    {
        dis[i] = inf;
        vis[i] = false;
        pre[i] = -1;
    }
    dis[s] = 0;
    vis[s] = true;
    q.push(s);
    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        vis[u] = false;
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;
            if (edge[i].cap > edge[i].flow &&
                    dis[v] > dis[u] + edge[i].cost )
            {
                dis[v] = dis[u] + edge[i].cost;
                pre[v] = i;
                if (!vis[v])
                {
                    vis[v] = true;
                    q.push(v);
                }
```

```
            }
        }
    }
    if (pre[t] == -1)return false;
    else return true;
}
int minCostMaxflow(int s, int t, int &cost)//返回流量, cost为费用
{
    int flow = 0;
    cost = 0;
    while (spfa(s, t))
    {
        int Min = inf;
        for (int i = pre[t]; i != -1; i = pre[edge[i ^ 1].v])
        {
            if (Min > edge[i].cap - edge[i].flow)
                Min = edge[i].cap - edge[i].flow;
        }
        for (int i = pre[t]; i != -1; i = pre[edge[i ^ 1].v])
        {
            edge[i].flow += Min;
            edge[i ^ 1].flow -= Min;
            cost += edge[i].cost * Min;
        }
        flow += Min;
    }
    return flow;
}
int N, M, K;
int kase;
int mtx[maxn][maxn];
int disxy(int x1, int y1, int x2, int y2)
{
    return abs(x1 - x2) + abs(y1 - y2) - 1;
}
void build()
{
    n = 3 + N * M * 2;
    st = 0, ed = 1;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            adde(st, i * M + j + 3, 1, 0);
        }
```

```
    }
    adde(st, 2, K, 0);
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            adde(2, i * M + j + N * M + 3, 1, 0);
            adde(i * M + j + N * M + 3, ed, 1, 0);
        }
    }
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < M; j++)
        {
            for (int h = i + 1; h < N; h++)
            {
                if (mtx[i][j] == mtx[h][j])
                {
                    adde(i * M + j + 3, h * M + j + N * M + 3, 1, h - i - 1 - mtx[i][j]);
                    // cout << i << " " << j << " " << h << " " << j << endl;
                }
                else
                {
                    adde(i * M + j + 3, h * M + j + N * M + 3, 1, h - i - 1);
                }
            }
            for (int h = j + 1; h < M; h++)
            {
                if (mtx[i][j] == mtx[i][h])
                {
                    adde(i * M + j + 3, i * M + h + N * M + 3, 1, h - j - 1 - mtx[i][j]);
                    // cout << i << " " << j << " " << i << " " << h << endl;
                }
                else
                {
                    adde(i * M + j + 3, i * M + h + N * M + 3, 1, h - j - 1);
                }
            }
        }
    }
}
void init()
{
    memset(head, -1, sizeof(head));
    en = 0;
```

```
    kase++;
}
void input()
{
    scanf("%d%d%d", &N, &M, &K);
    for (int i = 0; i < N; i++)
    {
        char str[maxn];
        scanf("%s", str);
        for (int j = 0; j < M; j++)
        {
            mtx[i][j] = str[j] - '0';
        }
    }
}
void debug()
{
    //
}
void solve()
{
    build();
    int cost;
    int flow = minCostMaxflow(st, ed, cost);
    // cout << "flow,cost: " << flow << " " << cost << endl;
    if (flow == N * M)
    {
        printf("Case %d : %d\n", kase, -cost);
    }
    else
    {
        printf("Case %d : %d\n", kase, -1);
    }
}
void output()
{
    //
}
int main()
{
    // int size = 256 << 20; // 256MB
    // char *p = (char *)malloc(size) + size;
    // __asm__("movl %0, %%esp\n" :: "r"(p));

    // std::ios_base::sync_with_stdio(false);
```

```
#ifndef ONLINE_JUDGE
    freopen("in.cpp", "r", stdin);
#endif

    kase = 0;
    int T;
    scanf("%d", &T);
    while (T--)
    {
        init();
        input();
        solve();
        output();
    }
    return 0;
}
```

15.11.15 点连通度与边连通度

连通度问题 在图中删去部分元素（点或边），使得图中指定的两个点s和t不连通（不存在从s到t的路径），求至少要删去几个元素。

点连通度 只许删点，求至少要删掉几个点（当然，s和t不能删去，这里保证原图中至少有三个点）；

边连通度 只许删边，求至少要删掉几条边。

有向图点连通度 需要拆点。建立一个网络，原图中的每个点i在网络中拆成i'与i''，有一条边$< i', i'' >$，容量为1（$< s', s'' >$和$< t', t'' >$例外，容量为正无穷）。原图中的每条边$< i, j >$在网络中为边$< i'', j' >$，容量为正无穷。以s'为源点、t''为汇点求最大流，最大流的值即为原图的点连通度。容量为正无穷的边不可能通过最小割，也就是原图中的边和s、t两个点不能删去；若边$< i', i'' >$通过最小割，则表示将原图中的点i删去。

有向图边连通度 这个其实就是最小割问题。以s为源点，t为汇点建立网络，原图中的每条边在网络中仍存在，容量为1，求该网络的最小割（也就是最大流）的值即为原图的边连通度。

无向图 将图中的每条边(i, j)拆成$< i, j >$和$< j, i >$两条边，再按照有向图的办法处理。

混合图 对于混合图，只需将其中所有的无向边按照无向图的办法处理、有向边按照有向图的办法处理即可。

点或边带权 边权为权重即可。

Chapter 16

Dynamic Programming

§ 16.1 线性模型

§ 16.2 串模型

§ 16.3 状态压缩模型

§ 16.4 四边形优化

16.4.1 朴素四边形优化

当函数$w(i,j)$满足$w(a,c)+w(b,d) <= w(b,c)+w(a,d)$ 且$a <= b < c <= d$ 时，我们称$w(i,j)$满足四边形不等式。。

当函数$w(i,j)$满足$w(i',j) <= w(i,j'); i <= i' < j <= j'$ 时，称w关于关于区间包含关系单调。

$s(i,j)=k$是指$m(i,j)$这个状态的最优决策

以上定理的证明自己去查些资料

今天看得lrj的书中介绍的四边形优化做个笔记，加强理解

最有代价用$d[i,j]$表示

$$d[i,j] = mind[i,k-1] + d[k+1,j] + w[i,j]$$

其中$w[i,j] = sum[i,j]$

- 四边形不等式$w[a,c]+w[b,d] <= w[b,c]+w[a,d] (a < b < c < d)$ 就称其满足凸四边形不等式
- 决策单调性$w[i,j] <= w[i',j']$ ($[i,j]$属于$[i',j']$) 既$i' <= i < j <= j'$

于是有以下三个定理

- 定理一：如果w同时满足四边形不等式和决策单调性,则d也满足四边形不等式
- 定理二：当定理一的条件满足时，让d[i,j]取最小值的k为K[i,j]，则$K[i,j-1] <= K[i,j] <= K[i+1,j]$
- 定理三：w为凸当且仅当$w[i,j]+w[i+1,j+1] <= w[i+1,j]+w[i,j+1]$

由定理三知判断w是否为凸即判断$w[i, j+1] - w[i, j]$的值随着i的增加是否递减

于是求K值的时候$K[i, j]$只和$K[i+1, j]$ 和$K[i, j-1]$有关，所以可以以$i-j$递增为顺序递推各个状态值最终求得结果将$O(n^3)$转为$O(n^2)$

POJ 1738 An old Stone Game

```
#include <cstdio>
#include <cstring>
#define N 1005
int s[N][N], f[N][N], sum[N], n;
int main()
{
    while (scanf("%d", &n) != EOF)
    {
        memset(f, 127, sizeof(f));
        sum[0] = 0;
        for (int i = 1; i <= n; i++)
        {
            scanf("%d", &sum[i]);
            sum[i] += sum[i - 1];
            f[i][i] = 0;
            s[i][i + 1] = i;
        }
        for (int i = 1; i <= n; i++)
            f[i][i + 1] = sum[i + 1] - sum[i - 1];

        for (int i = n - 2; i >= 1; i--)
            for (int j = i + 2; j <= n; j++)
                for (int k = s[i][j - 1]; k <= s[i + 1][j]; k++)
                    if (f[i][j] > f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1])
                    {
                        f[i][j] = f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1];
                        s[i][j] = k;
                    }

        printf("%d\n", f[1][n]);
    }
    return 0;
}
```

16.4.2 GarsiaWachs算法(POJ 1738 An old Stone Game)

1. 这类题目一开始想到是DP, 设$dp[i][j]$表示第i堆石子到第j堆石子合并最小得分. 状态方程: $dp[i][j] = min(dp[i][k] + dp[k+1][j] + sum[j] - sum[i-1])$; sum[i]表示第1到第i堆石子总和. 递归记忆化搜索即可.

2. 不过此题有些不一样, $1 <= n <= 50000$范围特大, dp[50000][50000]开不到这么大数组. 问题分析:

   (a) 假设我们只对3堆石子a,b,c进行比较, 先合并哪2堆, 使得得分最小.

   $$score1 = (a + b) + ((a + b) + c)$$

   $$score2 = (b + c) + ((b + c) + a)$$

   再次加上$score1 <= score2$, 化简得: $a <= c$, 可以得出只要a和c的关系确定, 合并的顺序也确定.

   (b) GarsiaWachs算法, 就是基于(1)的结论实现.找出序列中满足$stone[i-1] <= stone[i+1]$最小的i, 合并$temp = stone[i] + stone[i-1]$, 接着往前面找是否有满足$stone[j] > temp$, 把temp值插入$stone[j]$的后面(数组的右边). 循环这个过程一直到只剩下一堆石子结束.

   (c) 为什么要将$temp$插入$stone[j]$的后面, 可以理解为(1)的情况从$stone[j+1]$到$stone[i-2]$看成一个整体$stone[mid]$,现在$stone[j], stone[mid], temp(stone[i-1] + stone[i-1])$, 情况因为$temp < stone[j]$,因此不管怎样都是$stone[mid]$和temp先合并, 所以讲temp值插入$stone[j]$ 的后面是不影响结果.

```
/*
有n堆石头排成一条直线 ，每堆石头的个数已知，现在要将这n堆石头合并成一堆，每次合并只
能合并相邻的两堆石头，代价就是新合成石头堆的石头数，现在问将这n堆石头合并成一堆，最
小代价是多少？
*/
#include <cstdio>
#include <iostream>
#include <cstring>
using namespace std;
#define MAX 50005
int n;
int a[MAX];
int num, result;
void combine(int k)
{
    int i, j;
    int temp = a[k] + a[k - 1];
    result += temp;
    for (i = k; i < num - 1; ++i)
        a[i] = a[i + 1];
    num--;
    for (j = k - 1; j > 0 && a[j - 1] < temp; --j)
        a[j] = a[j - 1];
    a[j] = temp;
    while (j >= 2 && a[j] >= a[j - 2])
    {
        int d = num - j;
```

```
        combine(j - 1);
        j = num - d;
    }
}
int main()
{
    int i;
    while (scanf("%d", &n) != EOF)
    {
        if (n == 0) break;
        for (i = 0; i < n; ++i)
            scanf("%d", &a[i]);
        num = 1;
        result = 0;
        for (i = 1; i < n; ++i)
        {
            a[num++] = a[i];
            while (num >= 3 && a[num - 3] <= a[num - 1])
                combine(num - 2);
        }
        while (num > 1) combine(num - 1);
        printf("%d\n", result);
    }
    return 0;
}
```

§ 16.5 经典问题

16.5.1 最长上升子序列LIS

16.5.2 最长公共子序列LCS

Subsequence(不连续)

$$dp(i,j) = \begin{cases} dp(i-1,j-1)+1, & a[i] = b[j] \\ max\{dp(i-1,j), dp(i,j-1)\}, & a[i] \neq b[j] \end{cases}$$

Substring(连续)

$$dp(i,j) = \begin{cases} dp(i-1,j-1)+1, & a[i] = b[j] \\ 0, & a[i] \neq b[j] \end{cases}$$

相同位置相同

$$dp(i,j)=\begin{cases}dp(i-1,j-1)+1, & a[i]=b[j]\\ dp(i-1,j-1), & a[i]\neq b[j]\end{cases}$$

16.5.3 最大子矩阵和(Ural 1146)

预处理$dp[i][j]$表示第i行前j个元素之和（即：前缀和），将其压缩为一维最大子段和问题，$O(N^3)$。

```
#include <cstdio>
#include <algorithm>
using namespace std;
typedef long long LL;
const int maxn = 1010;
const int inf = 0x3f3f3f3f;
int kase, n;
int mtx[maxn][maxn];
int dp[maxn][maxn];
void init()
{
    kase++;
}
void input()
{
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            scanf("%d", &mtx[i][j]);
}
void solve()
{
    for (int i = 1; i <= n; i++)
    {
        dp[i][0] = 0;
        for (int j = 1; j <= n; j++)
            dp[i][j] = dp[i][j - 1] + mtx[i][j];
    }
    int ans = -inf;
    for (int i = 1; i <= n; i++)
    {
        for (int j = i; j <= n; j++)
        {
            int tmp = dp[1][j] - dp[1][i - 1];
            int sum = tmp;
            for (int k = 2; k <= n; k++)
```

```
            {
                if (sum > 0) sum += dp[k][j] - dp[k][i - 1];
                else sum = dp[k][j] - dp[k][i - 1];
                tmp = max(tmp, sum);
            }
            ans = max(tmp, ans);
        }
    }
    printf("%d\n", ans);
}
int main()
{
    kase = 0;
    while (~scanf("%d", &n))
    {
        init();
        input();
        solve();
    }
    return 0;
}
```

16.5.4 类TSP问题(状压DP)(POJ 3311 Hie with the Pie)

```
/*
POJ 3311 Hie with the Pie
有N个城市(1~N)和一个PIZZA店(0),要求一条回路，从0出发，又回到0，而且距离最短

用FLOYD先求出任意2点的距离dis[i][j]
枚举所有状态，用11位二进制表示10个城市和pizza店，1表示经过，0表示没有经过
定义状态DP(i,s)表示在s状态下，到达城市i的最优值
状态转移方程:DP(i,s) = min{DP(k,s^(1<<(i-1))) + dis[k][j],DP(i,s)}
其中s^(1<<(i-1))表示未到达城市i的所有状态，1<=k<=n
对于全1的状态，即s = (1<<n)-1则表示经过所有城市的状态，最终还需要回到PIZZA店0
那么最终答案就是min{DP(i,s) + dis[i][0]}
*/
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
const int maxn = 12;
const int inf = 0x3f3f3f3f;
int dis[maxn][maxn];
int n;
int dp[maxn][(1 << 10) + 5];
```

```
void init()
{
    memset(dp, 0, sizeof(dp));
}
void input()
{
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= n; j++)
            scanf("%d", &dis[i][j]);
}
void floyd()
{
    for (int k = 0; k <= n; k++)
        for (int i = 0; i <= n; i++)
            for (int j = 0; j <= n; j++)
                dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
}
void caldp()
{
    int tot = (1 << n);
    for (int s = 0; s < tot; s++)
    {
        for (int i = 1; i <= n; i++)
        {
            if (s == (1 << (i - 1)))
                dp[i][s] = dis[0][i];
            else if (s & (1 << (i - 1)))
            {
                dp[i][s] = inf;
                for (int j = 1; j <= n; j++)
                    if ((s & (1 << (j - 1))) && j != i)
                        dp[i][s] = min(dp[i][s], dp[j][s ^ (1 << (i - 1))]
                                       + dis[j][i]);
            }
        }
    }
}
void solve()
{
    floyd();
    caldp();
    int ans = inf;
    int tot = (1 << n) - 1;
    for (int i = 1; i <= n; i++)
        ans = min(ans, dp[i][tot] + dis[i][0]);
```

```
    printf("%d\n", ans);
}
int main()
{
    while (~scanf("%d", &n))
    {
        if (!n) break;
        init();
        input();
        solve();
    }
    return 0;
}
```

Chapter 17

Other Algorithms

§ 17.1 Get Min(A[i]-A[j]) (0,1,2,. . . ,n-1)

```
int ans=A[0]-A[1];
int MaxAi=A[0];
for(int i=1;i<n;i++)
{
    ans=max(ans,MaxAi-A[i]);
    MaxAi=max(A[i],MaxAi);
}
```

§ 17.2 Get a Circle (Floyd)

即：Floyd 判圈法

```
do
{
    k1=next(n,k1);//People 1

    k2=next(n,k2);//People 2, first step
    if(k2>ans) ans=k2;

    k2=next(n,k2);//People 2, second step
    if(k2>ans) ans=k2;
}while(k1!=k2);//stop when overtake
```

§ 17.3 Meet in the Middle

即：中途相遇法。

先找出前$n/2$ 的结果并保存在一个集合中，然后找后$n/2$ 的结果并在前面的集合中查找得出最终结果(可使用STL 的map)。

例题：UVa 1326 - Jurassic Remains。使用后将复杂度从$O(2^n)$ 降到$O(2^{n/2}lgn)$。

Part V

Classic Problems