算法模板

时间复杂度

```
n≤30      dfs+剪枝,状态压缩dp
n≤100,    floyd,dp,高斯消元
n≤1000    dp,二分,朴素版Dijkstra、朴素版Prim、Bellman-Ford
n≤10^4    块状链表、分块、莫队
n≤10^5    各种sort，线段树、树状数组、set/map、heap、拓扑排序、dijkstra+heap、prim+heap、spfa、求凸包、求半平面交、二分、          CDQ分治、整体二分
n≤10^6    单调队列、 hash、双指针扫描、并查集，kmp、AC自动机，常数比较小的 O(nlogn)O(nlogn)的做法：sort、树状数组、heap、dijkstra、spfa
n≤10^7    双指针扫描、kmp、AC自动机、线性筛素数
n≤10^9    判断质数
n≤10^18   最大公约数，快速幂
```

1 排序

- 快速排序

```
void quick_sort(int q[], int l ,int r) {
    if (l >= r) return;

    int i = l - 1, j = r + 1, x = q[(l + r) >> 1];
    while (i < j) {
        do i++; while(q[i] < x);
        do j--; while(q[j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j), quick_sort(q, j + 1, r);
}
```

- 归并排序

```
void merge_sort(int q[], int l, int r)
{
    if (l >= r) return;

    int mid = l + r >> 1;
    merge_sort(q, l, mid);
    merge_sort(q, mid + 1, r);

    int k = 0, i = l, j = mid + 1;
    while (i <= mid && j <= r)
        if (q[i] <= q[j]) tmp[k ++ ] = q[i ++ ];
        else tmp[k ++ ] = q[j ++ ];

    while (i <= mid) tmp[k ++ ] = q[i ++ ];
    while (j <= r) tmp[k ++ ] = q[j ++ ];

    for (i = l, j = 0; i <= r; i ++, j ++ ) q[i] = tmp[j];
}
```

- 整数二分

```
bool check(int x) {/* ... */} // 检查x是否满足某种性质

// 区间[l, r]被划分成[l, mid]和[mid + 1, r]时使用:
int bsearch_1(int l, int r)
{
    while (l < r)
    {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;    // check()判断mid是否满足性质
        else l = mid + 1;
    }
    return l;
}
// 区间[l, r]被划分成[l, mid - 1]和[mid, r]时使用:
int bsearch_2(int l, int r)
{
    while (l < r)
    {
        int mid = l + r + 1 >> 1;
        if (check(mid)) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

例题-acwing-789

2 高精度 前缀和 差分

- 高精度加法

```
// C = A + B, A >= 0, B >= 0
// A, B 为倒序
// 计算A + B (1234 + 5678) 输入应该为 A [4, 3, 2, 1],  B[8,  7,  6,  5]
// 得到的 C 也为倒序
vector<int> add(vector<int> &A, vector<int> &B)
{
    if (A.size() < B.size()) return add(B, A);

    vector<int> C;
    int t = 0;
    for (int i = 0; i < A.size(); i ++ )
    {
        t += A[i];
        if (i < B.size()) t += B[i];
        C.push_back(t % 10);
        t /= 10;
    }

    if (t) C.push_back(t);   // 最高位进位
    return C;
}
```

- 高精度减法

```
// C = A - B, 满足A >= B, A >= 0, B >= 0
// 同加法, 输入为逆序
vector<int> sub(vector<int> &A, vector<int> &B)
{
    vector<int> C;
    for (int i = 0, t = 0; i < A.size(); i ++ )
    {
        t = A[i] - t;
        if (i < B.size()) t -= B[i];
        C.push_back((t + 10) % 10);
        if (t < 0) t = 1;
        else t = 0;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back();   // 去掉先导 0
    return C;
}
```

- 高精度乘低精度

```
// C = A * b, A >= 0, b > 0
vector<int> mul(vector<int> &A, int b)
{
    vector<int> C;
```

```
    int t = 0;
    for (int i = 0; i < A.size() || t; i ++ )
    {
        if (i < A.size()) t += A[i] * b;
        C.push_back(t % 10);
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back(); // 去掉先导 0

    return C;
}
```

- 高精度除以低精度

```
// A / b = C ... r, A >= 0, b > 0
vector<int> div(vector<int> &A, int b, int &r)
{
    vector<int> C;
    r = 0;
    for (int i = A.size() - 1; i >= 0; i -- )
    {
        r = r * 10 + A[i];
        C.push_back(r / b);
        r %= b;
    }
    reverse(C.begin(), C.end());
    while (C.size() > 1 && C.back() == 0) C.pop_back();
    return C;
}
```

- 前缀和

```
一维前缀和
S[i] = a[1] + a[2] + ... a[i]
a[l] + ... + a[r] = S[r] - S[l - 1]

二维前缀和
S[i, j] = 第i行j列格子左上部分所有元素的和
以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵的和为：
S[x2, y2] - S[x1 - 1, y2] - S[x2, y1 - 1] + S[x1 - 1, y1 - 1]
```

- 差分

```
一维差分
给区间[l, r]中的每个数加上c：B[l] += c, B[r + 1] -= c

二维差分
给以(x1, y1)为左上角，(x2, y2)为右下角的子矩阵中的所有元素加上c：
S[x1, y1] += c, S[x2 + 1, y1] -= c, S[x1, y2 + 1] -= c, S[x2 + 1, y2 + 1] += c
```

一维差分-练习题

二维差分-练习题

3 离散化 区间和并

- 离散化（值域很大，但数据稀疏）

```
vector<int> alls; // 存储所有待离散化的值
sort(alls.begin(), alls.end()); // 将所有值排序
alls.erase(unique(alls.begin(), alls.end()), alls.end());   // 去掉重复元素

// 二分求出x对应的离散化的值
int find(int x) // 找到第一个大于等于x的位置
{
    int l = 0, r = alls.size() - 1;
    while (l < r)
    {
        int mid = l + r >> 1;
        if (alls[mid] >= x) r = mid;
        else l = mid + 1;
    }
    return r + 1; // 映射到1, 2, ...n
}
```

区间和--(离散化 + 前缀和)

- 区间和并

```
// 将所有存在交集的区间合并
void merge(vector<PII> &segs)
{
    vector<PII> res;

    sort(segs.begin(), segs.end());

    int st = -2e9, ed = -2e9;
    for (auto seg : segs)
        if (ed < seg.first)
        {
            if (st != -2e9) res.push_back({st, ed});
            st = seg.first, ed = seg.second;
        }
        else ed = max(ed, seg.second);
```

```
    if (st != -2e9) res.push_back({st, ed});

    segs = res;
}
```

区间和并

4 数组链表 单调栈 KMP

- 数组实现静态链表，提高效率

```
// head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点
int head, e[N], ne[N], idx;

// 初始化
void init()
{
    head = -1;
    idx = 0;
}

// 在链表头插入一个数a
void insert(int a)
{
    e[idx] = a, ne[idx] = head, head = idx ++ ;
}

// 将头结点删除，需要保证头结点存在
void remove()
{
    head = ne[head];
}
```

- 单调栈

```
常见模型：找出每个数左边离它最近的比它大/小的数
int tt = 0;
for (int i = 1; i <= n; i ++ )
{
    while (tt && check(stk[tt], i)) tt -- ;
    stk[ ++ tt] = i;
}
```

单调栈习题

- 单调队列

```
常见模型：找出滑动窗口中的最大值/最小值
int hh = 0, tt = -1;
for (int i = 0; i < n; i ++ )
{
    while (hh <= tt && check_out(q[hh])) hh ++ ;  // 判断队头是否滑出窗口
    while (hh <= tt && check(q[tt], i)) tt -- ;
    q[ ++ tt] = i;
}
```

滑动窗口

- KMP算法

```
// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度
cin >> m >> p + 1 >> n >> s + 1;

// 求 next 数组
for (int i = 2, j = 0; i <= m; i++) {
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j++;
    ne[i] = j;
}


// 匹配
for (int i = 1, j = 0; i <= n; i++) {
    while( j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j++;
    if (j == m) { // 匹配成功
        j = ne[j];
        // todo
    }
}
```

KMP 字符串

5 Tire树 并查集

- Tire树

```
int son[N][26], cnt[N], idx;
// 这里 26 是表示所有小写字母(若有大写字母，则可修改为 52)
// 0号点既是根节点，又是空节点
// son[][]存储树中每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char *str) {
    int p = 0;
    for (int i = 0; str[i]; i ++ ) {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++ idx;
        p = son[p][u];
```

```
    }
    cnt[p] ++ ;
}

// 查询字符串出现的次数
int query(char *str) {
    int p = 0;
    for (int i = 0; str[i]; i ++ ) {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

Tire 字符串统计

- 并查集

(1)朴素并查集：

```
    int p[N]; //存储每个点的祖宗节点,根节点满足 p[x] = x

    // 返回x的祖宗节点
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ ) p[i] = i;

    // 合并a和b所在的两个集合：
    p[find(a)] = find(b);
```

(2)维护size的并查集：

```
    int p[N], size[N];
    //p[]存储每个点的祖宗节点, size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的
数量

    // 返回x的祖宗节点
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ ) {
        p[i] = i;
        size[i] = 1;
    }

    // 合并a和b所在的两个集合，必须先修改size，再修改父节点。
    size[find(b)] += size[find(a)];    // 这行必须在前
```

```
    p[find(a)] = find(b);
```

(3)维护到祖宗节点距离的并查集：

```
    int p[N], d[N];
    //p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离

    // 返回x的祖宗节点
    int find(int x) {
        if (p[x] != x) {
            int u = find(p[x]);
            d[x] += d[p[x]];
            p[x] = u;
        }
        return p[x];
    }

    // 初始化，假定节点编号是1~n
    for (int i = 1; i <= n; i ++ ) {
        p[i] = i;
        d[i] = 0;
    }

    // 合并a和b所在的两个集合：
    p[find(a)] = find(b);
    d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
```

合并集合

连通块中点的数量

- 手写堆

```
// h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1
// ph[k]存储第k个插入的点在堆中的位置
// hp[k]存储堆中下标是k的点是第几个插入的
int h[N], ph[N], hp[N], size;

// 交换两个点，及其映射关系
void heap_swap(int a, int b) {
    swap(ph[hp[a]],ph[hp[b]]);
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u) {
    int t = u;
    if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;  // 用 h[t]
比较，取三者最小值
    if (u != t) {
        heap_swap(u, t);
        down(t);
    }
}
```

```
void up(int u) {
    while (u / 2 && h[u] < h[u / 2]) {
        heap_swap(u, u / 2);
        u >>= 1;
    }
}

// O(n)建堆
for (int i = n / 2; i; i -- ) down(i);
```

堆排序

模拟堆

6 字符串哈希 STL常用函数

- 一般哈希

```
(1) 拉链法
    int h[N], e[N], ne[N], idx;

    // 向哈希表中插入一个数
    void insert(int x)
    {
        int k = (x % N + N) % N;
        e[idx] = x;
        ne[idx] = h[k];
        h[k] = idx ++ ;
    }

    // 在哈希表中查询某个数是否存在
    bool find(int x)
    {
        int k = (x % N + N) % N;
        for (int i = h[k]; i != -1; i = ne[i])
            if (e[i] == x)
                return true;

        return false;
    }

(2) 开放寻址法
    int h[N];

    // 如果x在哈希表中，返回x的下标；如果x不在哈希表中，返回x应该插入的位置
    int find(int x)
    {
        int t = (x % N + N) % N;
        while (h[t] != null && h[t] != x)
        {
            t ++ ;
            if (t == N) t = 0;
        }
```

```
        return t;
    }
```

模拟散列表

- 字符串哈希

```
核心思想：将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低
小技巧：取模的数用2^64，这样直接用unsigned long long存储，溢出的结果就是取模的结果

typedef unsigned long long ULL;
ULL h[N], p[N]; // h[k]存储字符串前k个字母的哈希值, p[k]存储 P^k mod 2^64

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i ++ )
{
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r)
{
    return h[r] - h[l - 1] * p[r - l + 1];
}
```

字符串哈希

- C++ STL常用函数

```
vector, 变长数组，倍增的思想
    size()  返回元素个数
    empty()  返回是否为空
    clear()  清空
    front()/back()
    push_back()/pop_back()
    begin()/end()
    []
    支持比较运算，按字典序

pair<int, int>
    first, 第一个元素
    second, 第二个元素
    支持比较运算，以first为第一关键字，以second为第二关键字（字典序）

string，字符串
    size()/length()  返回字符串长度
    empty()
    clear()
    substr(起始下标，(子串长度))  返回子串
    c_str()  返回字符串所在字符数组的起始地址

queue, 队列
```

```
    size()
    empty()
    push()  向队尾插入一个元素
    front()  返回队头元素
    back()  返回队尾元素
    pop()  弹出队头元素

priority_queue, 优先队列，默认是大根堆
    size()
    empty()
    push()  插入一个元素
    top()  返回堆顶元素
    pop()  弹出堆顶元素
    定义成小根堆的方式：priority_queue<int, vector<int>, greater<int>> q;

stack, 栈
    size()
    empty()
    push()  向栈顶插入一个元素
    top()  返回栈顶元素
    pop()  弹出栈顶元素

deque, 双端队列
    size()
    empty()
    clear()
    front()/back()
    push_back()/pop_back()
    push_front()/pop_front()
    begin()/end()
    []

set, map, multiset, multimap, 基于平衡二叉树（红黑树），动态维护有序序列
    size()
    empty()
    clear()
    begin()/end()
    ++, -- 返回前驱和后继，时间复杂度 O(logn)

    set/multiset
        insert()  插入一个数
        find()  查找一个数
        count()  返回某一个数的个数
        erase()
            (1) 输入是一个数x，删除所有x   O(k + logn)
            (2) 输入一个迭代器，删除这个迭代器
        lower_bound()/upper_bound()
            lower_bound(x)  返回大于等于x的最小的数的迭代器
            upper_bound(x)  返回大于x的最小的数的迭代器
    map/multimap
        insert()  插入的数是一个pair
        erase()  输入的参数是pair或者迭代器
        find()
        []  注意multimap不支持此操作。 时间复杂度是 O(logn)
        lower_bound()/upper_bound()

unordered_set, unordered_map, unordered_multiset, unordered_multimap, 哈希表
    和上面类似，增删改查的时间复杂度是 O(1)
```

```
    不支持 lower_bound()/upper_bound(),  迭代器的++, --

bitset, 压位
    bitset<10000> s;
    ~, &, |, ^
    >>, <<
    ==, !=
    []

    count()  返回有多少个1

    any()  判断是否至少有一个1
    none()  判断是否全为0

    set()  把所有位置成1
    set(k, v)  将第k位变成v
    reset()  把所有位变成0
    flip()  等价于~
    flip(k) 把第k位取反
```

7 DFS BFS 拓扑排序

- 树与图的存储(邻接表)

```
// 对于每个点k，开一个单链表，存储k所有可以走到的点。h[k]存储这个单链表的头结点
int h[N], e[N], ne[N], idx;

// 添加一条边a->b
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

// 初始化
idx = 0;
memset(h, -1, sizeof h);
```

- 树与图的遍历

```
(1) 深度优先
int dfs(int u)
{
    st[u] = true; // st[u] 表示点u已经被遍历过

    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j]) dfs(j);
    }
}
```

```
(2) 宽度优先
queue<int> q;
st[1] = true; // 表示1号点已经被遍历过
q.push(1);

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

    for (int i = h[t]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true; // 表示点j已经被遍历过
            q.push(j);
        }
    }
}
```

[树的重心]

[图中点的层次]

- 拓扑排序 O(n + m)

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <queue>
#include <bitset>

using namespace std;

int n, m;
const int N = 30010;
int h[N], e[N], ne[N], idx;
bool st[N];
int q[N], d[N];
bitset<N> f[N];

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

bool topsort() {
    int hh = 0, tt = -1;

    // d[i] 存储点i的入度
    for (int i = 1; i <= n; i ++ )
        if (!d[i])
          q[ ++ tt] = i;

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

```
        for (int i = h[t]; i != -1; i = ne[i]) {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
    }
}

int main() {
    cin >> n >> m;
    memset(h, -1, sizeof h);

    for (int i = 1; i <= m; i ++) {
        int x, y;
        cin >> x >> y;
        add(x, y);
        d[y] ++;
    }

    topsort();

    for (int i = n - 1; i >= 0; i -- ) {
        int j = q[i];
        f[j][j] = 1;
        for (int k = h[j]; ~k; k = ne[k])
            f[j] |= f[e[k]];
    }

    for (int i = 1; i <= n; i ++ ) cout << f[i].count() << endl;

    return 0;
}
```

[有向图的拓扑排序]

最大最小搜索

Alice 和 Bob 正在玩井字棋游戏。
井字棋游戏的规则很简单：两人轮流往 3×3 的棋盘中放棋子，Alice 放的是 X，Bob 放的是 O，Alice执先。
当同一种棋子占据一行、一列或一条对角线的三个格子时，游戏结束，该种棋子的持有者获胜。
当棋盘被填满的时候，游戏结束，双方平手。
Alice 设计了一种对棋局评分的方法：

对于 Alice 已经获胜的局面，评估得分为(棋盘上的空格子数+1)；
对于 Bob 已经获胜的局面，评估得分为 -(棋盘上的空格子数+1)；
对于平局的局面，评估得分为 0；

输入格式
输入的第一行包含一个正整数 T，表示数据的组数。
每组数据输入有 3 行，每行有 3 个整数，用空格分隔，分别表示棋盘每个格子的状态。0 表示格子为空，1 表示格子中为X，2 表示格子中为 O。保证不会出现其他状态。

保证输入的局面合法。(即保证输入的局面可以通过行棋到达，且保证没有双方同时获胜的情况).保证输入的局面轮到 Alice 行棋。

输出格式

对于每组数据，输出一行一个整数，表示当前局面的得分。

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

using namespace std;

const int N = 3, INF = 1e8;

int g[N][N];

bool check(int x)
{
    for (int i = 0; i < 3; i ++ )
    {
        int s = 0;
        for (int j = 0; j < 3; j ++ )
            if (g[i][j] == x)
                s ++ ;
        if (s == 3) return true;
        s = 0;
        for (int j = 0; j < 3; j ++ )
            if (g[j][i] == x)
                s ++ ;
        if (s == 3) return true;
    }
    if (g[0][0] == x && g[1][1] == x && g[2][2] == x) return true;
    if (g[2][0] == x && g[1][1] == x && g[0][2] == x) return true;
    return false;
}

int evaluate()
{
    int s = 0;
    for (int i = 0; i < 3; i ++ )
        for (int j = 0; j < 3; j ++ )
            if (!g[i][j])
                s ++ ;
    if (check(1)) return s + 1;
    if (check(2)) return -(s + 1);
    if (!s) return 0;
    return INF;
}

int dfs(int u)
{
    int t = evaluate();
    if (t != INF) return t;

    if (!u)  // Alice
    {
        int res = -INF;
```

```
        for (int i = 0; i < 3; i ++ )
            for (int j = 0; j < 3; j ++ )
                if (!g[i][j])
                {
                    g[i][j] = 1;
                    res = max(res, dfs(1));
                    g[i][j] = 0;
                }
        return res;
    }
    else  // Bob
    {
        int res = INF;
        for (int i = 0; i < 3; i ++ )
            for (int j = 0; j < 3; j ++ )
                if (!g[i][j])
                {
                    g[i][j] = 2;
                    res = min(res, dfs(0));
                    g[i][j] = 0;
                }
        return res;
    }
}

int main()
{
    int T;
    cin >> T;
    while (T -- )
    {
        for (int i = 0; i < 3; i ++ )
            for (int j = 0; j < 3; j ++ )
                cin >> g[i][j];
        cout << dfs(0) << endl;
    }
    return 0;
}
```

8 最短路算法

朴素dijkstra算法

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

using namespace std;

const int N = 1010;
int n, m;
```

```
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra() {
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ ) {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

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

int main() {
  cin >> n >> m;
  // 求最短路，初始化距离为极大
  memset(g, 0x3f, sizeof(g));
  for (int i = 1; i <= m; i ++) {
    int x, y, z;
    cin >> x >> y >> z;
    g[x][y] = min(g[x][y], z);
  }

  cout << dijkstra() << endl;

  return 0;
}
```

dijkstra 求最短路

堆优化版dijkstra

```
// 时间复杂度 O(nm), n表示点数，m表示边数
// 注意在模板题中需要对下面的模板稍作修改，加上备份数组，详情见模板题。
typedef pair<int, int> PII;

int n;      // 点的数量
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储所有点到1号点的距离
bool st[N];     // 存储每个点的最短距离是否已确定
```

```cpp
// 求1号点到n号点的最短距离，如果不存在，则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, 1});      // first存储距离，second存储节点编号

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

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

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

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

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

[dijkstra 求最短路II]

拆点

小明和小芳出去乡村玩，小明负责开车，小芳来导航。
小芳将可能的道路分为大道和小道。

大道比较好走，每走 1 公里小明会增加 1 的疲劳度。
小道不好走，如果连续走小道，小明的疲劳值会快速增加，连续走 s 公里小明会增加 s2 的疲劳度。

现在小芳拿到了地图，请帮助她规划一个开车的路线，使得按这个路线开车小明的疲劳度最小。

输入格式
输入的第一行包含两个整数 n,m，分别表示路口的数量和道路的数量。路口由 1 至 n 编号，小明需要开车从 1 号路口到 n 号路口。
接下来 m 行描述道路，每行包含四个整数 t,a,b,c，表示一条类型为 t，连接 a 与 b 两个路口，长度为 c 公里的双向道路。其中 t 为 0 表示大道，t 为 1 表示小道。

保证 1 号路口和 n 号路口是连通的。

输出格式
输出一个整数，表示最优路线下小明的疲劳度。

数据范围
对于 30% 的评测用例，1≤n≤8，1≤m≤10；
对于另外 20% 的评测用例，不存在小道；
对于另外 20% 的评测用例，所有的小道不相交；
对于所有评测用例，1≤n≤500，1≤m≤10^5，1≤a,b≤n，t 是 0 或 1，c≤10^5。
保证答案不超过 10^6。

```cpp
// 答案不超过 10^6，因此小路最大连续距离不超过 1000，考虑拆点
#include <iostream>
#include <cstring>
#include <algorithm>
#include <queue>

using namespace std;

const int N = 510, M = 200010, INF = 1e6;

int n, m;
int h[N], e[M], f[M], w[M], ne[M], idx;
int dist[N][1010];
bool st[N][1010];
struct Node {
    int x, y, v;
    bool operator< (const Node& t) const {
        return v > t.v;
    }
};

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

void dijkstra() {
    priority_queue<Node> heap;
    heap.push({1, 0, 0});
    memset(dist, 0x3f, sizeof dist);
    dist[1][0] = 0;
    while (heap.size()) {
        auto t = heap.top();
        heap.pop();

        if (st[t.x][t.y]) continue;
        st[t.x][t.y] = true;
        for (int i = h[t.x]; ~i; i = ne[i]) {
            int x = e[i], y = t.y;
            if (f[i]) {
                y += w[i];
                if (y <= 1000) {
                    if (dist[x][y] > t.v - t.y * t.y + y * y) {
                        dist[x][y] = t.v - t.y * t.y + y * y;
                        if (dist[x][y] <= INF)
                            heap.push({x, y, dist[x][y]});
                    }
                }
            }
            else {
```

```
                if (dist[x][0] > t.v + w[i]) {
                    dist[x][0] = t.v + w[i];
                    if (dist[x][0] <= INF)
                        heap.push({x, 0, dist[x][0]});
                }
            }
        }
    }
}

int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m -- )
    {
        int t, a, b, c;
        scanf("%d%d%d%d", &t, &a, &b, &c);
        add(t, a, b, c), add(t, b, a, c);
    }

    dijkstra();
    int res = INF;
    for (int i = 0; i <= 1000; i ++ ) res = min(res, dist[n][i]);
    printf("%d\n", res);

    return 0;
}
```

Bellman-Ford

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

using namespace std;

const int N = 510, M = 10010;

struct Edge {
    int a, b, c;
}edges[M];

int n, m, k;
int dist[N];
int last[N];

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

    dist[1] = 0;
    for (int i = 0; i < k; i ++ ) {   // 循环 K 次，从1号点到n号点的最多经过k条边的最短距离
```

```
        memcpy(last, dist, sizeof dist);
        for (int j = 0; j < m; j ++ ) {
            auto e = edges[j];
            dist[e.b] = min(dist[e.b], last[e.a] + e.c);
        }
    }
}

int main() {
    scanf("%d%d%d", &n, &m, &k);

    for (int i = 0; i < m; i ++ ) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        edges[i] = {a, b, c};
    }

    bellman_ford();

    if (dist[n] > 0x3f3f3f3f / 2) puts("impossible");
    else printf("%d\n", dist[n]);

    return 0;
}
```

spfa

G 国国王来中国参观后，被中国的高速铁路深深的震撼，决定为自己的国家也建设一个高速铁路系统。建设高速铁路投入非常大，为了节约建设成本，G 国国王决定不新建铁路，而是将已有的铁路改造成高速铁路。
现在，请你为 G 国国王提供一个方案，将现有的一部分铁路改造成高速铁路，使得任何两个城市间都可以通过高速铁路到达，而且从所有城市乘坐高速铁路到首都的最短路程和原来一样长。
请你告诉 G 国国王在这些条件下最少要改造多长的铁路。

输入格式
    输入的第一行包含两个整数 n,m，分别表示 G 国城市的数量和城市间铁路的数量。所有的城市由 1 到 n 编号，首都为 1 号。
    接下来 m 行，每行三个整数 a,b,c，表示城市 a 和城市 b 之间有一条长度为 c 的双向铁路。这条铁路不会经过 a 和 b 以外的城市。

输出格式
    输出一行，表示在满足条件的情况下最少要改造的铁路长度。

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

using namespace std;

const int N = 10010, M = 200010, INF = 0x3f3f3f3f;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
int dist[N], q[N];
bool st[N];
```

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

// 做完spfa之后, dist数组表示各个点到 源点 的最短路径距离
void spfa() {
    int hh = 0, tt = 1;
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    q[0] = 1;

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

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

int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m -- ) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c), add(b, a, c);
    }
    spfa();

    int res = 0;
    // 反推哪些边被选中, 很关键
    for (int a = 2; a <= n; a ++ ) {
        int minw = INF;
        for (int j = h[a]; ~j; j = ne[j]) {
            int b = e[j];
            if (dist[a] == dist[b] + w[j])
                minw = min(minw, w[j]);
        }

        res += minw;
    }

    printf("%d\n", res);
    return 0;
}
```

Floyd

```
const int INF = 0x3f3f3f3f;
int n, m, Q;
int d[N][N];

// 初始化：
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

// 处理结果时(查询 a b)
if (d[a][b] > INF / 2) puts("impossible");  // 可能存在 INF ->(-2) INF 的情况
else printf("%d", d[a][b]);
```

Floyd求最短路

Floyd 传递闭包

给定 n 个变量和 m 个不等式。其中 n 小于等于 26，变量分别用前 n 的大写英文字母表示。
不等式之间具有传递性，即若 A>B 且 B>C，则 A>C。请从前往后遍历每对关系，每次遍历时判断：
1. 如果能够确定全部关系且无矛盾，则结束循环，输出确定的次序；
2. 如果发生矛盾，则结束循环，输出有矛盾；
3. 如果循环结束时没有发生上述两种情况，则输出无定解。

输入格式
输入包含多组测试数据。每组测试数据，第一行包含两个整数 n 和 m。接下来 m 行，每行包含一个不等式，不等式全部为小于关系。当输入一行 0 0 时，表示输入终止。

输出格式
每组数据输出一个占一行的结果。结果可能为下列三种之一：
1. 如果可以确定两两之间的关系，则输出 "Sorted sequence determined after t relations: yyy...y.",其中't'指迭代次数，'yyy...y'是指升序排列的所有变量。
2. 如果有矛盾，则输出： "Inconsistency found after t relations."，其中't'指迭代次数。
3. 如果没有矛盾，且不能确定两两之间的关系，则输出 "Sorted sequence cannot be determined."。

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

```cpp
using namespace std;

const int N = 26;

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

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

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

int check() {
    for (int i = 0; i < n; i ++ )
        if (d[i][i])
            return 2;

    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < i; j ++ )
            if (!d[i][j] && !d[j][i])
                return 0;

    return 1;
}

char get_min() {
    for (int i = 0; i < n; i ++ )
        if (!st[i]) {
            bool flag = true;
            for (int j = 0; j < n; j ++ )
                if (!st[j] && d[j][i]) {
                    flag = false;
                    break;
                }
            if (flag) {
                st[i] = true;
                return 'A' + i;
            }
        }
}

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

            if (!type) {
                g[a][b] = 1;
                floyd();
```

```
                type = check();
                if (type) t = i;
            }
        }

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

    return 0;
}
```

9 最小生成树

Prim

```
const int INF = 0x3f3f3f3f;

// 时间复杂度是 O(n^2 + m), n表示点数，m表示边数
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f)，否则返回最小生成树的树边权重之和
int prim() {
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ ) {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

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

    return res;
```

```
}

// 需要初始化 g[][] :
//          g[i][i] = 0,  g[i][j] = INF
```

Prim求最小生成树

Kruskal

```
const int INF = 0x3f3f3f3f;

// 时间复杂度是 O(mlogm), n表示点数，m表示边数
int n, m;       // n是点数，m是边数
int p[N];       // 并查集的父节点数组

struct Edge {    // 存储边
    int a, b, w;

    bool operator< (const Edge &W)const {
        return w < W.w;
    }
}edges[M];

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

int kruskal() {
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ ) {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b) {    // 如果两个连通块不连通，则将这两个连通块合并
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}
```

Kruskal最小生成树

10 二分图

二分图的一个等价定义是：不含有 含奇数条边的环 的图

染色法判别二分图

```
int n;      // n表示点数
int h[N], e[M], ne[M], idx;     // 邻接表存储图
int color[N];       // 表示每个点的颜色，-1表示未染色，0表示白色，1表示黑色

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

// 参数：u表示当前节点，c表示当前点的颜色
bool dfs(int u, int c)
{
    color[u] = c;
    for (int i = h[u]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (color[j] == -1)
        {
            if (!dfs(j, !c)) return false;
        }
        else if (color[j] == c) return false;
    }

    return true;
}

bool check()
{
    memset(color, -1, sizeof color);
    bool flag = true;
    for (int i = 1; i <= n; i ++ )
        if (color[i] == -1)
            if (!dfs(i, 0))
            {
                flag = false;
                break;
            }
    return flag;
}
```

染色法判定二分图

```
class Solution {
public:
    int vis[101], color[101];
    queue<int> q;
    bool bfs(int u, vector<vector<int>> &graph){
        q.push(u);
        color[u] = 1;
        while (!q.empty()){
            int u = q.front();
```

```
            q.pop();
            for (auto v : graph[u]){
                if (color[v] != 0){
                    if (color[v] == color[u]) return false;
                }
                else{
                    q.push(v);
                    color[v] = (color[u] == 1)? 2 : 1;
                }
            }
        }
        return true;
    }
    bool isBipartite(vector<vector<int>>& graph) {
        for (int i = 0; i < graph.size(); ++i){
            if (!color[i] && !bfs(i, graph)) return false;
        }
        return true;
    }
};
```

匈牙利算法(二分图最大匹配算法)

最小点覆盖：无向图中选取最少的点，使得覆盖掉所有的边

最大独立集：无向图中选取最多的点，使得选出的点没有边

最大匹配数 = 最小点覆盖 = 总点数 - 最大独立集 = 总点数 - 最小路径点覆盖

```
int n1, n2;     // n1表示第一个集合中的点数，n2表示第二个集合中的点数
int h[N], e[M], ne[M], idx;     // 邻接表存储所有边，匈牙利算法中只会用到从第一个集合指向第二个集合的边，所以这里只用存一个方向的边
int match[N];       // 存储第二个集合中的每个点当前匹配的第一个集合中的点是哪个
bool st[N];     // 表示第二个集合中的每个点是否已经被遍历过

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

bool find(int x)
{
    for (int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if (!st[j])
        {
            st[j] = true;
            if (match[j] == 0 || find(match[j]))
            {
                match[j] = x;
                return true;
            }
        }
    }
```

```
    return false;
}

// 求最大匹配数，依次枚举第一个集合中的每个点能否匹配第二个集合中的点
int res = 0;
for (int i = 1; i <= n1; i ++ )
{
    memset(st, false, sizeof st);
    if (find(i)) res ++ ;
}
```

二分图的最大匹配

11 背包问题

01背包

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

using namespace std;

const int N = 1010;
int dp[N];
int v[N], w[N];

int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) cin >> v[i] >> w[i];

    // 状态转移方程 dp[i][j] = max(dp[i-1][j-w] + v, dp[i-1][j-w])
    // 优化版本，注意第二层循环，思考优化的根据
    for (int i = 1; i <= n; ++i)
        for (int j = m; j >= v[i]; --j)
                dp[j] = max(dp[j], dp[j - v[i]] + w[i]);

    cout << dp[m] << endl;
    return 0;
}
```

01背包模板题

完全背包

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

using namespace std;

const int N = 1010;
int dp[N];
int v[N], w[N];
```

```
int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; ++i) cin >> v[i] >> w[i];
/*
f[i,j] = max(f[i-1,j],f[i-1,j-v]+w,f[i-1,j-2*v]+2*w,f[i-1,j-3*v]+3*w, .....)
f[i,j-v]= max(f[i-1,j-v],f[i-1,j-2*v]+w,f[i-1,j-2*v]+2*w,.....)
由上两式，可得出如下递推关系：
f[i][j]=max(f[i-1][j], f[i,j-v]+w)
*/
    for (int i = 1; i <= n; ++i) {
        for (int j = v[i]; j <= m; ++j) {
            dp[j] = max(dp[j], dp[j - v[i]] + w[i]);
        }
    }

    cout << dp[m] << endl;

    return 0;
}
```

完全背包模板题

多重背包

有 N 种物品和一个容量是 V 的背包。
第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。
求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

输入格式
    第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。
    接下来有 N 行，每行三个整数 vi,wi,si，用空格隔开，分别表示第 i 种物品的体积、价值和数量。
输出格式
    输出一个整数，表示最大价值。

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

using namespace std;

const int N = 110;

int n, m;
int v[N], w[N], s[N];
int f[N][N];

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

    for (int i = 1; i <= n; i ++ ) cin >> v[i] >> w[i] >> s[i];

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

```
        for (int j = 0; j <= m; j ++ )
            for (int k = 0; k <= s[i] && k * v[i] <= j; k ++ )
                f[i][j] = max(f[i][j], f[i - 1][j - v[i] * k] + w[i] * k);

    cout << f[n][m] << endl;
    return 0;
}
```

分组背包

有 N 组物品和一个容量是 V 的背包。
每组物品有若干个，同一组内的物品最多只能选一个。
每件物品的体积是 v_ij，价值是 w_ij，其中 i 是组号，j 是组内编号。
求解将哪些物品装入背包，可使物品总体积不超过背包容量，且总价值最大。
输出最大价值。

输入格式
第一行有两个整数 N，V，用空格隔开，分别表示物品组数和背包容量。
接下来有 N 组数据：
每组数据第一行有一个整数 S_i，表示第 i 个物品组的物品数量；
每组数据接下来有 S_i 行，每行有两个整数 v_ij,w_ij，用空格隔开，分别表示第 i 个物品组的第 j 个物品的体积和价值；
输出格式
输出一个整数，表示最大价值

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

using namespace std;

const int N = 110;

int n, m;
int v[N][N], w[N][N], s[N];
int f[N];

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

    for (int i = 1; i <= n; i ++ ) {
        cin >> s[i];
        for (int j = 0; j < s[i]; j ++ )
            cin >> v[i][j] >> w[i][j];
    }

    for (int i = 1; i <= n; i ++ )
        for (int j = m; j >= 0; j -- )
            for (int k = 0; k < s[i]; k ++ )
                if (v[i][k] <= j)
                    f[j] = max(f[j], f[j - v[i][k]] + w[i][k]);

    cout << f[m] << endl;

    return 0;
```

```
}
```

12 数学相关

- 判定质数

```
bool is_prime(int x) {
    if (x < 2) return false;
    for (int i = 2; i <= x / i; i ++ )
        if (x % i == 0)
            return false;
    return true;
}
```

- 求区间内 素数个数 (线性筛法)

```
// 时间复杂度   O(n)
int primes[N], cnt;     // primes[]存储所有素数
bool st[N];         // st[x]存储x是否被筛掉

void get_primes(int n) {
    for (int i = 2; i <= n; i ++) {
        if (!st[i]) primes[cnt ++ ] = i;
        for (int j = 0; primes[j] <= n / i; j ++) {
            st[primes[j] * i] = true;
            if (i % primes[j] == 0) break;
        }
    }
}
```

- 最大公约数

```
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
```

- 快速幂

```
// 求 m^k mod p，时间复杂度 O(logk)。
int qmi(int m, int k, int p) {
    int res = 1 % p, t = m;
    while (k) {
        if (k & 1) res = res * t % p;
        t = t * t % p;
        k >>= 1;
    }
    return res;
}
```

- 组合数

```
// c[a][b] 表示从a个苹果中选b个的方案数
// 注意 溢出问题，通常会在下面代码基础上取模
for (int i = 0; i < N; i ++ )
    for (int j = 0; j <= i; j ++ )
        if (!j) c[i][j] = 1;
        else c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]);
```

- 扩展欧几里得算法

```
// 扩展欧几里得算法，求x，y，使得ax + by = gcd(a，b)
// 返回值为 gcd(a，b)
LL exgcd(LL a, LL b, LL &x, LL &y) {
    if (!b) {
        x = 1; y = 0;
        return a;
    }
    LL d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d;
}
```

13 线段树

朴素版

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

using namespace std;

typedef long long LL;

const int N = 100010;

int n, m;
int a[N];
struct Node {  // 根据问题具体选择存储内容
    int l, r;
    LL sum, add;
}tr[N * 4];

// 往上传一层
void pushup(int u) {
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}

// 往下传一层
```

```
void pushdown(int u) {
    auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1];
    if (root.add) {
        left.add += root.add, left.sum += (LL)(left.r - left.l + 1) * root.add;
        right.add += root.add, right.sum += (LL)(right.r - right.l + 1) *
root.add;
        root.add = 0;
    }
}

// 构建线段树
void build(int u, int l, int r) {   // 固定写法
    if (l == r) tr[u] = {l, r, a[r], 0};
    else {
        tr[u] = {l, r};  // 左右端点初始化
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}

// 区间修改
void modify(int u, int l, int r, int d) {
    if (tr[u].l >= l && tr[u].r <= r) {
        tr[u].sum += (LL)(tr[u].r - tr[u].l + 1) * d;
        tr[u].add += d;
    }
    else {  // 分裂
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, d);
        if (r > mid) modify(u << 1 | 1, l, r, d);
        pushup(u);
    }
}

// 区间查询
LL query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].sum;

    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    LL sum = 0;
    if (l <= mid) sum = query(u << 1, l, r);
    if (r > mid) sum += query(u << 1 | 1, l, r);
    return sum;
}


int main() {
    scanf("%d%d", &n, &m);

    for (int i = 1; i <= n; i ++ ) scanf("%d", &a[i]);

    build(1, 1, n);
    /*----------------------------------具体看----------------------------------
*/
    char op[2];
```

```
    int l, r, d;

    while (m -- ) {
        scanf("%s%d%d", op, &l, &r);
        if (*op == 'C') {
            scanf("%d", &d);
            modify(1, l, r, d);
        }
        else printf("%lld\n", query(1, l, r));
    }
    /*------------------------------------------------------------------------
*/

    return 0;
}
```

可在此测试代码

支持加法和乘法

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

using namespace std;

typedef long long LL;

const int N = 100010;

int n, p, m;
int a[N];
struct Node {
    int l, r;
    int sum, add, mul;
}tr[N * 4];

void pushup(int u) {
    tr[u].sum = (tr[u << 1].sum + tr[u << 1 | 1].sum) % p;
}

void eval(Node &t, int add, int mul) {
    // 先乘后加
    // 修改前        x = x * a + b
    // 修改后希望是 x = ((x * a) + b) * c + d = x * (a * c) + b * c + d
    t.sum = ((LL)t.sum * mul + (LL)(t.r - t.l + 1) * add) % p;
    t.mul = (LL)t.mul * mul % p;
    t.add = ((LL)t.add * mul + add) % p;
}

void pushdown(int u) {
    eval(tr[u << 1], tr[u].add, tr[u].mul);
    eval(tr[u << 1 | 1], tr[u].add, tr[u].mul);
    tr[u].add = 0, tr[u].mul = 1;
}
```

```
void build(int u, int l, int r) {
    if (l == r) tr[u] = {l, r, a[r], 0, 1};
    else {
        tr[u] = {l, r, 0, 0, 1};
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        pushup(u);
    }
}

void modify(int u, int l, int r, int add, int mul) {
    if (tr[u].l >= l && tr[u].r <= r) eval(tr[u], add, mul);
    else {
        pushdown(u);
        int mid = tr[u].l + tr[u].r >> 1;
        if (l <= mid) modify(u << 1, l, r, add, mul);
        if (r > mid) modify(u << 1 | 1, l, r, add, mul);
        pushup(u);
    }
}

int query(int u, int l, int r) {
    if (tr[u].l >= l && tr[u].r <= r) return tr[u].sum;

    pushdown(u);
    int mid = tr[u].l + tr[u].r >> 1;
    int sum = 0;
    if (l <= mid) sum = query(u << 1, l, r);
    if (r > mid) sum = (sum + query(u << 1 | 1, l, r)) % p;
    return sum;
}

int main() {
    scanf("%d%d", &n, &p);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &a[i]);
    build(1, 1, n);

    scanf("%d", &m);
    while (m -- ) {
        int t, l, r, d;
        scanf("%d%d%d", &t, &l, &r);
        if (t == 1) {
            scanf("%d", &d);
            modify(1, l, r, 0, d);
        }
        else if (t == 2) {
            scanf("%d", &d);
            modify(1, l, r, d, 1);
        }
        else printf("%d\n", query(1, l, r));
    }

    return 0;
}
```

可在此测试代码

14 网络流

基本概念

对于任意一张有向图（也就是网络），其中有N个点、M条边以及源点S和汇点T

c(x,y)称为边的容量

f(x,y)称为边的流量

流函数f(x,y)三大性质：

容量限制：每条边的流量总不可能大于该边的容量的（不然水管就爆了）

斜对称：正向边的流量=反向边的流量（反向边后面会具体讲）

流量守恒：正向的所有流量和=反向的所有流量和（就是总量始终不变）

最大流

使得整个网络流量之和最大的流函数称为网络的最大流，此时的流量和被称为网络的最大流量

增广路

若一条从S到T的路径上所有边的剩余容量都大于0，则称这样的路径为一条增广路

反向边

因为可能一条边可以被包含于多条增广路径，所以为了寻找所有的增广路经我们就要让这一条边有多次被选择的机会

构建反向边则是这样一个机会，相当于给程序一个反悔的机会。

残量网络

在任意时刻，网络中所有节点以及剩余容量大于0的边构成的子图被称为残量网络

邻接表“成对存储”

将正向边和反向边存在“2和3”、“4和5”、“6和7”。因为在更新边权的时候，我们就可以直接使用xor的方

式，找到对应的正向边和反向边

知识点梳理

1. 基本概念
    1.1 流网络，不考虑反向边
    1.2 可行流，不考虑反向边
        1.2.1 两个条件：容量限制、流量守恒
        1.2.2 可行流的流量指从源点流出的流量 - 流入源点的流量
        1.2.3 最大流是指最大可行流
    1.3 残留网络，考虑反向边，残留网络的可行流f' + 原图的可行流f = 原题的另一个可行流
        (1) |f' + f| = |f'| + |f|
        (2) |f'| 可能是负数
    1.4 增广路径
    1.5 割
        1.5.1 割的定义
        1.5.2 割的容量，不考虑反向边，“最小割”是指容量最小的割。
        1.5.3 割的流量，考虑反向边，f(S, T) <= c(S, T)
        1.5.4 对于任意可行流f，任意割[S, T]，|f| = f(S, T)
        1.5.5 对于任意可行流f，任意割[S, T]，|f| <= c(S, T)
        1.5.6 最大流最小割定理
            (1) 可以流f是最大流
            (2) 可行流f的残留网络中不存在增广路
            (3) 存在某个割[S, T]，|f| = c(S, T)
    1.6. 算法

Dinic算法

1. 在残量网络上 BFS 求出节点的层次，构造分层图
2. 在分层图上 DFS 寻找增广路，在回溯时同时更新边权

代码实现(固定风格)

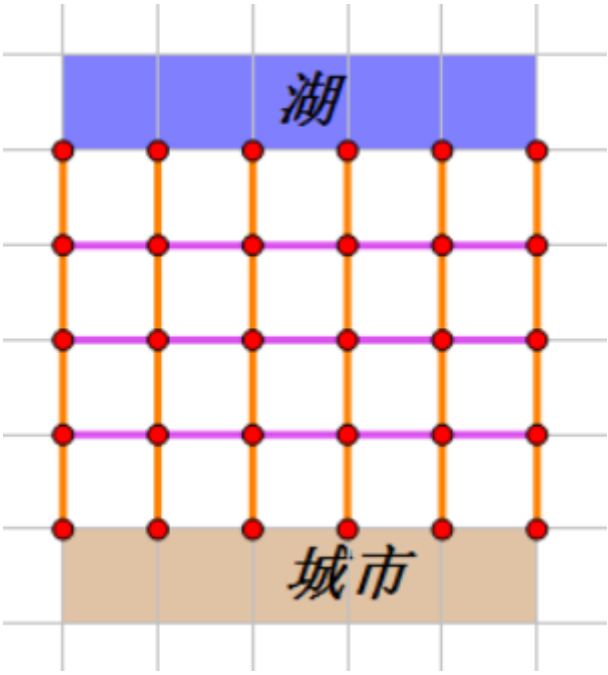
```
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 10010, M = 200010, INF = 1e8;

int n, m, S, T;
int h[N], e[M], f[M], ne[M], idx; // f[N] 为 容量
int q[N], d[N], cur[N]; // 队列    深度    当前弧

void add(int a, int b, int c) {
    e[idx] = b, f[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
    e[idx] = a, f[idx] = 0, ne[idx] = h[b], h[b] = idx ++ ;
}

bool bfs() {
    int hh = 0, tt = 0;
    memset(d, -1, sizeof d);
    q[0] = S, d[S] = 0, cur[S] = h[S];

    while (hh <= tt) {
        int t = q[hh ++ ];
        for (int i = h[t]; ~i; i = ne[i]) {
            int ver = e[i];
            if (d[ver] == -1 && f[i]) {
                d[ver] = d[t] + 1;
                cur[ver] = h[ver];
                if (ver == T)  return true;
                q[ ++ tt] = ver;
            }
        }
    }
```

```
    return false;
}

int find(int u, int limit) {
    if (u == T) return limit;
    int flow = 0;

    for (int i = cur[u]; ~i && flow < limit; i = ne[i]) {
        cur[u] = i;  // 当前弧优化
        int ver = e[i];
        if (d[ver] == d[u] + 1 && f[i]) {
            int t = find(ver, min(f[i], limit - flow));
            if (!t) d[ver] = -1;
            f[i] -= t, f[i ^ 1] += t, flow += t;
        }
    }

    return flow;
}

int dinic() {
    int r = 0, flow;
    while (bfs()) {  // 存在增广路时循环继续
        while (flow = find(S, INF)) r += flow;
    }
    return r;
}

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

    printf("%d\n", dinic());

    return 0;
}
```

最大流模板题

参考题解

最小割



这片管网由 n 行 m 列节点（红色，图中 n=5，m=6），横向管道（紫色）和纵向管道（橙色）构成。
行和列分别用 1 到 n 的整数和 1 到 m 的整数表示。
第 1 行的任何一个节点均可以抽取湖水，湖水到达第 n 行的任何一个节点即算作引入了城市。
除第一行和最后一行外，横向相邻或纵向相邻的两个节点之间一定有一段管道，每一段管道都有各自的最大的抽水速率，并需要根据情况选择抽水还是放水。

对于纵向的管道（橙色），允许从上方向下方抽水或从下方向上方放水；如果从图中的上方向下方抽水，那么单位时间内能通过的水量不能超过管道的最大速率；如果从下方向上方放水，因为下方海拔较高，因此可以允许有任意大的水量。
对于横向的管道（紫色），允许从左向右或从右向左抽水，不允许放水，两种情况下单位时间流过的水量都不能超过管道的最大速率。

现在 MF 城市的水务负责人想知道，在已知每个管道单位时间容量的情况下，MF 城每单位时间最多可以引入多少的湖水。

```
// 最大流 -> 平面图最小割 -> 最短路 -> 分层图最短路
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

typedef long long LL;
const int N = 5050;

int n, m, A, B, Q, X;
int r[N][N], c[N][N];
LL d[N];

int main()
{
    scanf("%d%d%d%d%d%d", &n, &m, &A, &B, &Q, &X);
    // 边权
    for (int i = 1; i <= n - 1; i ++ )
        for (int j = 1; j <= m; j ++ )
```

```
            c[i][j] = X = ((LL)A * X + B) % Q;
    for (int i = 2; i <= n - 1; i ++ )
        for (int j = 1; j <= m - 1; j ++ )
            r[i][j] = X = ((LL)A * X + B) % Q;

    for (int j = 1; j <= m; j ++ )
    {
        for (int i = 1; i < n; i ++ )
            d[i] += c[i][j];
        // 只可能单向更新
        for (int i = 2; i < n; i ++ )
            d[i] = min(d[i], d[i - 1] + r[i][j]);
        for (int i = n - 2; i; i -- )
            d[i] = min(d[i], d[i + 1] + r[i + 1][j]);
    }
    LL res = 1e18;
    for (int i = 1; i < n; i ++ ) res = min(res, d[i]);
    printf("%lld\n", res);
    return 0;
}
```

无源汇上下界可行流

问题描述

给定一个包含 n 个点 m 条边的有向图，每条边都有一个流量下界和流量上界。

求一种可行方案使得在所有点满足流量平衡条件的前提下，所有边满足流量限制。

输入格式

第一行包含两个整数 n 和 m。接下来 m 行，每行包含四个整数 a,b,c,d 表示点 a 和 b 之间存在一条有向边，该边的流量下界为 c，流量上界为 d。点编号从 1 到 n。

输出格式

如果存在可行方案，则第一行输出 YES，接下来 m 行，每行输出一个整数，其中第 i 行的整数表示输入的第 i 条边的流量。

如果不存在可行方案，直接输出一行 NO。

如果可行方案不唯一，则输出任意一种方案即可

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

using namespace std;

const int N = 211, M = (10200 + N) * 2, INF = 1e8;

int n, m, S, T;
int h[N], e[M], f[M], l[M], ne[M], idx;   // l[M] 存储容量下界
int q[N], d[N], cur[N], A[N];  // A[N] 存储第 i 个点的出入度容量差

void add(int a, int b, int c, int d) {   // c : 容量下界   d : 容量上界
    e[idx] = b, f[idx] = d - c, l[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
```

```
    e[idx] = a, f[idx] = 0, ne[idx] = h[b], h[b] = idx ++ ;
}

bool bfs() {
    int hh = 0, tt = 0;
    memset(d, -1, sizeof d);
    q[0] = S, d[S] = 0, cur[S] = h[S];
    while (hh <= tt) {
        int t = q[hh ++ ];
        for (int i = h[t]; ~i; i = ne[i]) {
            int ver = e[i];
            if (d[ver] == -1 && f[i]) {
                d[ver] = d[t] + 1;
                cur[ver] = h[ver];
                if (ver == T) return true;
                q[ ++ tt] = ver;
            }
        }
    }
    return false;
}

int find(int u, int limit) {
    if (u == T) return limit;
    int flow = 0;
    for (int i = cur[u]; ~i && flow < limit; i = ne[i]) {
        cur[u] = i;
        int ver = e[i];
        if (d[ver] == d[u] + 1 && f[i]) {
            int t = find(ver, min(f[i], limit - flow));
            if (!t) d[ver] = -1;
            f[i] -= t, f[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}

int dinic() {
    int r = 0, flow;
    while (bfs()) while (flow = find(S, INF)) r += flow;
    return r;
}

int main() {
    scanf("%d%d", &n, &m);
    S = 0, T = n + 1;
    memset(h, -1, sizeof h);
    for (int i = 0; i < m; i ++ ) {
        int a, b, c, d;
        scanf("%d%d%d%d", &a, &b, &c, &d);
        add(a, b, c, d);
        A[a] -= c, A[b] += c;
    }

    int tot = 0;
    for (int i = 1; i <= n; i ++ )   // 注意流网络的处理    减去下界需要补边来保持流量守
恒
        if (A[i] > 0) add(S, i, 0, A[i]), tot += A[i];
```

```
        else if (A[i] < 0) add(i, T, 0, -A[i]);

    if (dinic() != tot) puts("NO");
    else {
        puts("YES");
        for (int i = 0; i < m * 2; i += 2)
            printf("%d\n", f[i ^ 1] + l[i]);
    }
    return 0;
}
```

有源汇上下界最大流

问题描述

给定一个包含 n 个点 m 条边的有向图，每条边都有一个流量下界和流量上界。

给定源点 S 和汇点 T，求源点到汇点的最大流。

输入格式

第一行包含四个整数 n,m,S,T。接下来 m 行，每行包含四个整数 a,b,c,d 表示点 a 和 b 之间存在一条有向边，该边的流量下界为 c，流量上界为 d。点编号从 1 到 n。

输出格式

输出一个整数表示最大流。如果无解，则输出 No Solution。

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

using namespace std;

const int N = 210, M = (N + 10000) * 2, INF = 1e8;

int n, m, S, T;
int h[N], e[M], f[M], ne[M], idx;
int q[N], d[N], cur[N], A[N];

void add(int a, int b, int c) {
    e[idx] = b, f[idx] = c, ne[idx] = h[a], h[a] = idx ++ ;
    e[idx] = a, f[idx] = 0, ne[idx] = h[b], h[b] = idx ++ ;
}

bool bfs() {
    int hh = 0, tt = 0;
    memset(d, -1, sizeof d);
    q[0] = S, d[S] = 0, cur[S] = h[S];
    while (hh <= tt) {
        int t = q[hh ++ ];
        for (int i = h[t]; ~i; i = ne[i]) {
            int ver = e[i];
            if (d[ver] == -1 && f[i]) {
                d[ver] = d[t] + 1;
                cur[ver] = h[ver];
                if (ver == T) return true;
                q[ ++ tt] = ver;
```

```
            }
        }
    }
    return false;
}

int find(int u, int limit) {
    if (u == T) return limit;
    int flow = 0;
    for (int i = cur[u]; ~i && flow < limit; i = ne[i]) {
        cur[u] = i;
        int ver = e[i];
        if (d[ver] == d[u] + 1 && f[i]) {
            int t = find(ver, min(f[i], limit - flow));
            if (!t) d[ver] = -1;
            f[i] -= t, f[i ^ 1] += t, flow += t;
        }
    }
    return flow;
}

int dinic() {
    int r = 0, flow;
    while (bfs()) while (flow = find(S, INF)) r += flow;
    return r;
}

int main() {
    int s, t;
    scanf("%d%d%d%d", &n, &m, &s, &t);
    S = 0, T = n + 1;
    memset(h, -1, sizeof h);
    while (m -- ) {
        int a, b, c, d;
        scanf("%d%d%d%d", &a, &b, &c, &d);
        add(a, b, d - c);
        A[a] -= c, A[b] += c;
    }

    int tot = 0;
    for (int i = 1; i <= n; i ++ )
        if (A[i] > 0) add(S, i, A[i]), tot += A[i];
        else if (A[i] < 0) add(i, T, -A[i]);

    add(t, s, INF); // 先以 s, t 为源点(注意到这是最后增加的两条边, 输出结果时会用到这个地方)

    if (dinic() < tot) puts("No Solution");
    else {
        int res = f[idx - 1];
        S = s, T = t;
        f[idx - 1] = f[idx - 2] = 0;
        printf("%d\n", res + dinic());

        /* 有源汇上下界最小流 只需要修改这里即可
        int res = f[idx - 1];
        S = t, T = s;
        f[idx - 1] = f[idx - 2] = 0;
```

```
        printf("%d\n", res - dinic());
        */
    }

    return 0;
}
```

最大权闭合子图

场景

若选择用户i，则ai,bi都必须选择，最大化点权和。这便是最大权闭合图的模型。

最大权闭合图的解法

新建源点S,向正权点连容量为点权的边；新建汇点T,负权点向T连容量为点权的相反数的边。图中原有的边容量改为正无穷。正权点点权和减去最小割即为答案。

正确性证明

选择用户i ⇒ ai,bi 必须选择，即必须割掉ai,bi连向T的边；放弃用户i ⇒ 割掉S连向i的边。

问题描述

在前期市场调查和站址勘测之后，公司得到了一共 N 个可以作为通讯信号中转站的地址，而由于这些地址的地理位置差异，在不同的地方建造通讯中转站需 要投入的成本也是不一样的，所幸在前期调查之后这些都是已知数据：

建立第 i 个通讯中转站需要的成本为 Pi(1≤i≤N)。

另外公司调查得出了所有期望中的用户群，一共 M 个。

关于第 i 个用户群的信息概括为 Ai,Bi 和 Ci：这些用户会使用中转站 Ai 和中转站 Bi 进行通讯，公司可以获益 Ci。（1≤i≤M,1≤Ai,Bi≤N）

THU 集团的 CS&T 公司可以有选择的建立一些中转站（投入成本），为一些用户提供服务并获得收益（获益之和）。

那么如何选择最终建立的中转站才能让公司的净获利最大呢？（净获利 = 获益之和 - 投入成本之和）

输入格式

第一行有两个正整数 N 和 M。第二行中有 N 个整数描述每一个通讯中转站的建立成本，依次为 P1,P2,…,PN 。以下 M 行，第 (i+2) 行的三个数 Ai,Bi 和 Ci 描述第 i 个用户群的信息。所有变量的含义可以参见题目描述。

输出格式

输出一个整数，表示公司可以得到的最大净获利。

```
// 模板省略，同上
int main() {
    scanf("%d%d", &n, &m);
    S = 0, T = n + m + 1;
    memset(h, -1, sizeof h);
    for (int i = 1; i <= n; i ++ ) {
        int p;
        scanf("%d", &p);
        add(m + i, T, p);
    }

    int tot = 0;
    for (int i = 1; i <= m; i ++ ) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(S, i, c);
        add(i, m + a, INF);
```

```
        add(i, m + b, INF);
        tot += c;
    }

    printf("%d\n", tot - dinic());

    return 0;
}
```

费用流

问题描述

给定一个包含 n 个点 m 条边的有向图，并给定每条边的容量和费用，边的容量非负。

图中可能存在重边和自环，保证费用不会存在负环。

求从 S 到 T 的最大流，以及在流量最大时的最小费用。

输入格式

第一行包含四个整数 n,m,S,T。接下来 m 行，每行三个整数 u,v,c,w，表示从点 u 到点 v 存在一条有向边，容量为 c，费用为 w(这里费用指的是单位流量的费用)。点的编号从 1 到 n。

输出格式

输出点 S 到点 T 的最大流和流量最大时的最小费用。

如果从点 S 无法到达点 T 则输出 0 0。

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

using namespace std;

const int N = 5010, M = 100010, INF = 1e8;

int n, m, S, T;
int h[N], e[M], f[M], w[M], ne[M], idx;
int q[N], d[N], pre[N], incf[N];
bool st[N];

void add(int a, int b, int c, int d) {
    e[idx] = b, f[idx] = c, w[idx] = d, ne[idx] = h[a], h[a] = idx ++ ;
    e[idx] = a, f[idx] = 0, w[idx] = -d, ne[idx] = h[b], h[b] = idx ++ ;  //
w[idx] = -d : 退流
}

bool spfa() { // 找最短增广路
    int hh = 0, tt = 1;
    memset(d, 0x3f, sizeof d);
    memset(incf, 0, sizeof incf);
    q[0] = S, d[S] = 0, incf[S] = INF;
    while (hh != tt) {
        int t = q[hh ++ ];
        if (hh == N) hh = 0;
        st[t] = false;

        for (int i = h[t]; ~i; i = ne[i]) {
```

```
            int ver = e[i];
            if (f[i] && d[ver] > d[t] + w[i]) {
                d[ver] = d[t] + w[i];
                pre[ver] = i;
                incf[ver] = min(f[i], incf[t]);
                if (!st[ver]) {
                    q[tt ++ ] = ver;
                    if (tt == N) tt = 0;
                    st[ver] = true;
                }
            }
        }
    }

    return incf[T] > 0;
}

void EK(int& flow, int& cost) {
    flow = cost = 0;
    while (spfa()) {
        int t = incf[T];
        flow += t, cost += t * d[T];
        for (int i = T; i != S; i = e[pre[i] ^ 1]) {
            f[pre[i]] -= t;
            f[pre[i] ^ 1] += t;
        }
    }
}

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

    int flow, cost;
    EK(flow, cost); // flow 为 最大流   cost 为 最小费用流
    printf("%d %d\n", flow, cost);

    /* 若还要求最大费用流, 则先恢复图, 然后将费用取反再跑 EK, -cost 则为最大费用流
    for (int i = 0; i < idx; i += 2) {
        f[i] += f[i ^ 1], f[i ^ 1] = 0;
        w[i] = -w[i], w[i ^ 1] = -w[i ^ 1];
    }

    EK(flow, cost);
    printf("%d\n", -cost);
    */

    return 0;
}
```

无源汇最小费用流

题目描述

管道网络可以用 n 个节点 m 条边的有向图表示。机器人从 1 号节点出发，清理完所有需要被清理的管道，最终回到 1 号节点。

管道一共有以下四种类型:

A 类:管道需要被清理，而且可以重复经过;

B 类:管道需要被清理，但是不能重复经过;

C 类:管道不需要被清理，而且可以重复经过;

D 类:管道不需要被清理，但是不能重复经过。

一条需要清理的管道在 机器人经过它时，就会被它清理。因为管道是有向的，所以机器人只能从一端走到另一端不能反过来走。对于不能重复经过的管道，机器人最多只能经过它一次。因为管道网络的特殊性，如果把管道看成无向图，A 类和 B 类管道及其端点构成的子图是连通的，并且 1 号节点一定在这个连通子图中。

机器人能不能在约束条件下清理完所有应该清理的管道（A 类和 B 类)。机器人 每经过一条管道一次，就会消耗 E 个包子。如果网络能够被成功清理，少要消耗多少个包子。如果网络无法被成功清理，则直接输出 -1。

输入格式

第一行包含三个非负整数 T、S、E，表示数据的组数、测试点的编号和经过一条管道一次消耗的包子数量。

接下来有 T 个部分，每个部分描述一组数据。对于评测数据，保证 T=10。保证 E∈{0,1}，这 T 组数据的包子消耗速度相同。每个部分第一行包含两个正整数 n、m，表示管道网络的节点数和边数。节点用 1~n 的整数编号。接下来 m 行，每行包含两个正整数 u,v 和一个大写字母 t，相邻两个元素之间用一个空格隔开，表示一条从节点 u 连向节点 v 的管道，其类型为 t。保证 1≤u,v≤n，t 一定是 A、B、C、D 之一。

输出格式

输出 T 行，每行一个整数，表示每组数据的答案。

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

using namespace std;

const int N = 210, M = 2010, INF = 1e8;

int n, m, S, T;
int h[N], e[M], f[M], w[M], ne[M], idx;
int q[N], d[N], pre[N], incf[N];
bool st[N];
int din[N], dout[N];

void add(int a, int b, int c, int d) {
    e[idx] = b, f[idx] = c, w[idx] = d, ne[idx] = h[a], h[a] = idx ++ ;
    e[idx] = a, f[idx] = 0, w[idx] = -d, ne[idx] = h[b], h[b] = idx ++ ;  //
w[idx] = -d : 退流
}

bool spfa() { // 找最短增广路
    int hh = 0, tt = 1;
    memset(d, 0x3f, sizeof d);
    memset(incf, 0, sizeof incf);
    q[0] = S, d[S] = 0, incf[S] = INF;
    while (hh != tt) {
```

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

        for (int i = h[t]; ~i; i = ne[i]) {
            int ver = e[i];
            if (f[i] && d[ver] > d[t] + w[i]) {
                d[ver] = d[t] + w[i];
                pre[ver] = i;
                incf[ver] = min(f[i], incf[t]);
                if (!st[ver]) {
                    q[tt ++ ] = ver;
                    if (tt == N) tt = 0;
                    st[ver] = true;
                }
            }
        }
    }

    return incf[T] > 0;
}

void EK(int& flow, int& cost) {
    flow = cost = 0;
    while (spfa()) {
        int t = incf[T];
        flow += t, cost += t * d[T];
        for (int i = T; i != S; i = e[pre[i] ^ 1]) {
            f[pre[i]] -= t;
            f[pre[i] ^ 1] += t;
        }
    }
}

int main()
{
    int Times, SS, E;
    scanf("%d%d%d", &Times, &SS, &E);
    while (Times -- ) {
        memset(h, -1, sizeof h);
        idx = 0;
        memset(din, 0, sizeof din);
        memset(dout, 0, sizeof dout);
        scanf("%d%d", &n, &m);
        S = 0, T = n + 1;

        int down_cost = 0;
        while (m -- ) {
            int a, b;
            char c;
            scanf("%d %d %c", &a, &b, &c);
            int down, up;
            if (c == 'A') down = 1, up = INF, down_cost += E;
            else if (c == 'B') down = up = 1, down_cost += E;
            else if (c == 'C') down = 0, up = INF;
            else down = 0, up = 1;
            add(a, b, up - down, E);
            din[b] += down, dout[a] += down;
```

```
        }

        // tot 用来判断是否满流
        int tot = 0;
        // 根据入流量和出流量, 决定向源点还是汇点连边
        for (int i = 1; i <= n; i ++ )
            if (din[i] > dout[i]) {
                add(S, i, din[i] - dout[i], 0);
                tot += din[i] - dout[i];
            }
            else add(i, T, dout[i] - din[i], 0);

        int flow, cost;
        EK(flow, cost);
        if (flow == tot) cost += down_cost;
        else {
            puts("-1");
            continue;
        }

        printf("%d\n", cost);
    }

    return 0;
}
```

15 差分约束

简介

- 应用背景 1： 求不等式的可行解
- 源点需要满足条件 : 从源点出发，一定可以走到所有边
- 步骤(最短路)
  - 先将每个不等式 $x_i <= x_j + c_k$ 转化为一条从 x_i 到 x_j的边， 边权为 c_k
  - 找一个满足条件的源点
  - 从源点出发求一遍最短路
    - 若存在负环， 则原不等式组无解
  - 否则 `dist[i]` 就是原不等式组的一个可行解

- 应用背景 2 : 如何求可行解的最值（变量x_i的最值）
  - 若求最小值 , 则应该求最长路 $x_i >= x_j + c_k$
  - 若求最大值 , 则应该求最短路 $x_i <= x_j + c_k$

细节

- 转化 $x_i <= c_k$ 这类不等式： 建立超级源点 x_0 , 然后建立 x_0 到 x_i 长度为 c_k的边

一道例题

幼儿园里有 N 个小朋友，老师现在想要给这些小朋友们分配糖果，要求每个小朋友都要分到糖果。
但是小朋友们也有嫉妒心，总是会提出一些要求，比如小明不希望小红分到的糖果比他的多，于是在分配糖果的时候， 老师需要满足小朋友们的 K 个要求。
幼儿园的糖果总是有限的，老师想知道他至少需要准备多少个糖果，才能使得每个小朋友都能够分到糖果，并且满足小朋友们所有的要求。

输入格式

输入的第一行是两个整数 N,K。接下来 K 行，表示分配糖果时需要满足的关系，每行 3 个数字 X,A,B。

如果 X=1. 表示第 A 个小朋友分到的糖果必须和第 B 个小朋友分到的糖果一样多。

如果 X=2，表示第 A 个小朋友分到的糖果必须少于第 B 个小朋友分到的糖果。

如果 X=3，表示第 A 个小朋友分到的糖果必须不少于第 B 个小朋友分到的糖果。

如果 X=4，表示第 A 个小朋友分到的糖果必须多于第 B 个小朋友分到的糖果。

如果 X=5，表示第 A 个小朋友分到的糖果必须不多于第 B 个小朋友分到的糖果。

小朋友编号从 1 到 N。

输出格式

输出一行，表示老师至少需要准备的糖果数，如果不能满足小朋友们的所有要求，就输出 -1。

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <stack>

using namespace std;

typedef long long LL;

const int N = 100010, M = 300010;

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

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

bool spfa() {
    stack<int> q;
    memset(dist, -0x3f, sizeof dist);
    dist[0] = 0;
    q.push(0);
    st[0] = true;

    while (q.size()) {
        int t = q.top();
```

```
        q.pop();
        st[t] = false;

        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            // 最长路
            if (dist[j] < dist[t] + w[i]) {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n + 1) return false;
                if (!st[j]) {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return true;
}

int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);

    while (m -- ) {
        int x, a, b;
        scanf("%d%d%d", &x, &a, &b);
        if (x == 1) add(b, a, 0), add(a, b, 0); // a >= b && b >= a
        else if (x == 2) add(a, b, 1);  // b >= a + 1
        else if (x == 3) add(b, a, 0);  // a >= b
        else if (x == 4) add(b, a, 1);  // a >= b + 1
        else add(a, b, 0);              // b >= a
    }

    // 建立 x0 超级源点
    for (int i = 1; i <= n; i ++ ) add(0, i, 1);

    if (!spfa()) puts("-1");
    else {
        LL res = 0;
        for (int i = 1; i <= n; i ++ ) res += dist[i];
        printf("%lld\n", res);
    }

    return 0;
}
```

16 字符串使用手册

- 读入加速

```
std::ios::sync_with_stdio(false);
std::cin.tie(0);
```

- 字符串大小写转换

```
transform(s.begin(), s.end(), s.begin(), ::tolower);
transform(s.begin(), s.end(), s.begin(), ::toupper);
```

- 字符串替换

```
// 从下标 pos 开始, 长度为 len 的子字符串被  str 替换
basic_string& replace (size_type pos, size_type len, const basic_string&
str); //str.replace(9,5,str2);

// iterator
basic_string& replace (const_iterator i1, const_iterator i2, const
basic_string& str);
```

- 新字符串 替换 所有指定子字符串 (自己实现的)

```
string subreplace(string resource_str, string sub_str, string new_str) {
        string::size_type pos = 0;
        while((pos = resource_str.find(sub_str)) != string::npos) {
        resource_str.replace(pos, sub_str.length(), new_str);
    }
    return resource_str;
}
```

- 删除元素

```
// 删除 pos 开始 长度为 len 的子字符串
basic_string& erase (size_type pos = 0, size_type len = npos); // str.erase
(10,8);

// iterator
iterator erase (const_iterator first, const_iterator last);  //str.erase
(str.begin()+5, str.end()-9);
```

- 查找给定字符串

```
// 找到 str 第一次出现的位置 (pos 可以指定搜索起点)
size_type find (const basic_string& str, size_type pos = 0);

std::string str ("There are two needles in this haystack with needles.");
std::string str2 ("needle");

// different member versions of find in the same order as above:
std::string::size_type found = str.find(str2);
if (found != std::string::npos)
    std::cout << "first 'needle' found at: " << found << '\n';
```

- 取子字符串

```
// 从 pos 开始, 长度为 len 的子字符串
basic_string substr (size_type pos = 0, size_type len = npos) const;
```

- 常用成员函数

```
find(ch, start = 0) 查找并返回从 start 开始的字符 ch 的位置；rfind(ch) 从末尾开始，查找并返回第一个找到的字符 ch 的位置（皆从 0 开始）（如果查找不到，返回 -1）。
substr(start, len) 可以从字符串的 start（从 0 开始）截取一个长度为 len 的字符串（缺省 len 时代码截取到字符串末尾）。
append(s) 将 s 添加到字符串末尾。
append(s, pos, n) 将字符串 s 中，从 pos 开始的 n 个字符连接到当前字符串结尾。
replace(pos, n, s) 删除从 pos 开始的 n 个字符，然后在 pos 处插入串 s。
erase(pos, n) 删除从 pos 开始的 n 个字符。
insert(pos, s) 在 pos 位置插入字符串 s。
```

17 Regex 正则

```
#include <iostream>
#include <string>
#include <regex>

using namespace std;

int main () {
    string s ("subject");
    regex e ("(sub)(.*)");

    if (regex_match (s,e))
    cout << "string object matched\n";

    if ( regex_match ( s.begin(), s.end(), e ) )
    cout << "range matched\n";

    smatch sm;    // same as match_results<string::const_iterator> sm;
    regex_match (s,sm,e);
    cout << "string object with " << sm.size() << " matches\n";
```

```
    cout << "the matches were: ";
    for (unsigned i=0; i<sm.size(); ++i) {
        cout << "[" << sm[i] << "] ";
    }

    cout << endl;

  return 0;
}

/*  输出
string object matched
range matched
string object with 3 matches
the matches were: [subject] [sub] [ject]
*/
```

18 日期处理模板

```
int month[] = {
    0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

// 判断 是否为 闰年
int is_leap(int year) {
    if (year % 4 == 0 && year % 100 || year % 400 == 0)
        return 1;
    return 0;
}

// 获取 m 月 天数
int get_month(int y, int m) {
    if (m == 2) return month[m] + is_leap(y);
    return month[m];
}
```

19 最近公共祖先

给定一棵包含 n 个节点的有根无向树，节点编号互不相同，但不一定是 1~n。
有 m 个询问，每个询问给出了一对节点的编号 x 和 y，询问 x 与 y 的祖孙关系。

输入格式
    输入第一行包括一个整数 表示节点个数；
    接下来 n 行每行一对整数 a 和 b，表示 a 和 b 之间有一条无向边。如果 b 是 -1，那么 a 就是树的根；
    第 n + 2 行是一个整数 m 表示询问个数；
    接下来 m 行，每行两个不同的正整数 x 和 y，表示一个询问。

输出格式
    对于每一个询问，若 x 是 y 的祖先则输出 1，若 y 是 x 的祖先则输出 2，否则输出 0。

倍增查询

```cpp
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 40010, M = N * 2;

int n, m;
int h[N], e[M], ne[M], idx;
int depth[N], fa[N][16];
int q[N];

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

void bfs(int root) {
    memset(depth, 0x3f, sizeof depth);
    depth[0] = 0, depth[root] = 1;
    int hh = 0, tt = 0;
    q[0] = root;
    while (hh <= tt) {
        int t = q[hh ++ ];
        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            if (depth[j] > depth[t] + 1) {
                depth[j] = depth[t] + 1;
                q[ ++ tt] = j;
                fa[j][0] = t;
                for (int k = 1; k <= 15; k ++ )
                    fa[j][k] = fa[fa[j][k - 1]][k - 1];
            }
        }
    }
}

int lca(int a, int b) {
    if (depth[a] < depth[b]) swap(a, b);
    for (int k = 15; k >= 0; k -- )
        if (depth[fa[a][k]] >= depth[b])
            a = fa[a][k];
    if (a == b) return a;
    for (int k = 15; k >= 0; k -- )
        if (fa[a][k] != fa[b][k]) {
            a = fa[a][k];
            b = fa[b][k];
        }
    return fa[a][0];
}

int main() {
    scanf("%d", &n);
    int root = 0;
```

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

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

    bfs(root);

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

    return 0;
}
```

20. 有向图强连通分量(Tarjan)

在有向图G中，如果两个顶点间至少存在一条路径，称两个顶点**强连通**。如果有向图G的每两个顶点都强连通，称G是一个**强连通图**。非强连通图有向图的极大强连通子图，称为**强连通分量**

此题Targan算法主要是将 图 进行缩点。得到各个强连通分量，构成拓扑图。

每一头牛的愿望就是变成一头最受欢迎的牛。
现在有 N 头牛，编号从 1 到 N，给你 M 对整数 (A,B)，表示牛 A 认为牛 B 受欢迎。
这种关系是具有传递性的，如果 A 认为 B 受欢迎，B 认为 C 受欢迎，那么牛 A 也认为牛 C 受欢迎。
你的任务是求出有多少头牛被除自己之外的所有牛认为是受欢迎的。

输入格式
    第一行两个数 N,M;
    接下来 M 行，每行两个数 A,B，意思是 A 认为 B 是受欢迎的（给出的信息有可能重复，即有可能出现多个 A,B）。

输出格式
    输出被除自己之外的所有牛认为是受欢迎的牛的数量。

输入样例：
```
3 3
1 2
2 1
2 3
```
输出样例：
```
1
```
样例解释
    只有第三头牛被除自己之外的所有牛认为是受欢迎的。

```cpp
#include <cstdio>
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 10010, M = 50010;

int n, m;
int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
// dfn[u] : 遍历到 u 的时间戳
// low[u] : 从 u 出发能够遍历到的最小时间戳
// timestamp : 当前时间戳
int stk[N], top;
bool in_stk[N];                  // 点是否在栈中
int id[N], scc_cnt, Size[N];
// id[u] : 点u所在强连通分量编号
// scc_cnt : 当前强连通分量个数
// Size[u] : 强连通分量的大小
int dout[N]; // 联通分量出度

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

void tarjan(int u) {
    dfn[u] = low[u] = ++ timestamp;
    stk[ ++ top] = u, in_stk[u] = true;

    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
        }
        else if (in_stk[j]) low[u] = min(low[u], dfn[j]);
    }

    if (dfn[u] == low[u]) {
        ++ scc_cnt;
        int y;
        do {
            y = stk[top -- ];
            in_stk[y] = false;
            id[y] = scc_cnt;
            Size[scc_cnt] ++ ;
        } while (y != u);
    }
}

int main() {
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    while (m -- ) {
        int a, b;
        scanf("%d%d", &a, &b);
```

```
        add(a, b);
    }

    for (int i = 1; i <= n; i ++ )
        if (!dfn[i])
            tarjan(i);

    for (int i = 1; i <= n; i ++ )
        for (int j = h[i]; ~j; j = ne[j]) {
            int k = e[j];
            int a = id[i], b = id[k];
            if (a != b) dout[a] ++ ;
        }

    int zeros = 0, sum = 0;
    for (int i = 1; i <= scc_cnt; i ++ )
        if (!dout[i]) {
            zeros ++ ;
            sum += Size[i];
            if (zeros > 1) {
                sum = 0;
                break;
            }
        }

    printf("%d\n", sum);

    return 0;
}
```

21. 最长公共上升子序列

问题描述

熊大妈的奶牛在小沐沐的熏陶下开始研究信息题目。
小沐沐先让奶牛研究了最长上升子序列，再让他们研究了最长公共子序列，现在又让他们研究最长公共上升子序列了。
小沐沐说，对于两个数列A和B，如果它们都包含一段位置不一定连续的数，且数值是严格递增的，那么称这一段数是两个数列的公共上升子序列，而所有的公共上升子序列中最长的就是最长公共上升子序列了。
奶牛半懂不懂，小沐沐要你来告诉奶牛什么是最长公共上升子序列。
不过，只要告诉奶牛它的长度就可以了。
数列A和B的长度均不超过3000。

输入格式
第一行包含一个整数N，表示数列A，B的长度。
第二行包含N个整数，表示数列A。
第三行包含N个整数，表示数列B。
输出格式
输出一个整数，表示最长公共上升子序列的长度。

数据范围
1≤N≤3000,序列中的数字均不超过231−1

输入样例：
4
2 2 1 3
2 1 2 3
输出样例：
2

问题分析

1. 状态表示
   f[i][j]代表所有a[1 ~ i]和b[1 ~ j]中以b[j]结尾的公共上升子序列的集合；
   f[i][j]的值等于该集合的子序列中长度的最大值；
2. 状态计算
   首先依据公共子序列中是否包含a[i]，将f[i][j]所代表的集合划分成两个不重不漏的子集：
      不包含a[i]的子集，最大值是f[i - 1][j]；
      包含a[i]的子集，将这个子集继续划分，依据是子序列的倒数第二个元素在b[]中是哪个数：
         子序列只包含b[j]一个数，长度是1；
         子序列的倒数第二个数是b[1]的集合，最大长度是f[i - 1][1] + 1；
         …
         子序列的倒数第二个数是b[j - 1]的集合，最大长度是f[i - 1][j - 1] + 1；

代码实现思路

```
for (int i = 1; i <= n; i ++ )
{
    for (int j = 1; j <= n; j ++ )
    {
        f[i][j] = f[i - 1][j];
        if (a[i] == b[j])
        {
            int maxv = 1;
            for (int k = 1; k < j; k ++ )
                if (a[i] > b[k])
                    maxv = max(maxv, f[i - 1][k] + 1);
            f[i][j] = max(f[i][j], maxv);
        }
    }
}
```

优化思路

我们发现每次循环求得的maxv是满足a[i] > b[k] 的 f[i - 1][k] + 1的前缀最大值。
因此可以直接将maxv提到第一层循环外面，减少重复计算，此时只剩下两重循环。
最终答案枚举子序列结尾取最大值即可。

最终代码实现

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

using namespace std;

const int N = 3010;

int n;
int a[N], b[N];
int f[N][N];

int main()
{
    scanf("%d", &n);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &a[i]);
    for (int i = 1; i <= n; i ++ ) scanf("%d", &b[i]);

    for (int i = 1; i <= n; i ++ )
    {
        int maxv = 1;
        for (int j = 1; j <= n; j ++ )
        {
            f[i][j] = f[i - 1][j];
            if (a[i] == b[j]) f[i][j] = max(f[i][j], maxv);
            if (a[i] > b[j]) maxv = max(maxv, f[i - 1][j] + 1);
        }
    }

    int res = 0;
    for (int i = 1; i <= n; i ++ ) res = max(res, f[n][i]);
    printf("%d\n", res);

    return 0;
}
```

22. 点分治

题目描述

给定一个有 N 个点（编号 0,1,…,N−1）的树，每条边都有一个权值（不超过 1000）。
树上两个节点 x 与 y 之间的路径长度就是路径上各条边的权值之和。
求长度不超过 K 的路径有多少条。

输入格式

输入包含多组测试用例。
每组测试用例的第一行包含两个整数 N 和 K。
接下来 N−1 行，每行包含三个整数 u,v,l，表示节点 u 与 v 之间存在一条边，且边的权值为 l。
当输入用例 N=0，K=0 时，表示输入终止，且该用例无需处理。

输出格式

每个测试用例输出一个结果。
每个结果占一行。

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 10010, M = N * 2;

int n, m;
int h[N], e[M], w[M], ne[M], idx;
bool st[N];
int p[N], q[N];

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

int get_size(int u, int fa) { // 求子树大小
    if (st[u]) return 0;  // 若节点 u 已经删掉则返回 0
    int res = 1;
    for (int i = h[u]; ~i; i = ne[i])
        if (e[i] != fa) // 若不往回搜
            res += get_size(e[i], u);
    return res;
}

int get_wc(int u, int fa, int tot, int& wc) { // 求重心(满足任意连通块大小小于 n/2 即可)
    if (st[u]) return 0;
    int sum = 1, ms = 0;  // sum : 子树大小   ms : 连通块大小的最大值
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (j == fa) continue;  // 判断是否往回搜
        int t = get_wc(j, u, tot, wc);
        ms = max(ms, t);
        sum += t;
    }

    ms = max(ms, tot - sum);
    if (ms <= tot / 2) wc = u;
    return sum;
}

void get_dist(int u, int fa, int dist, int& qt) {
    if (st[u] || dist > m) return;  // 若 u 已经删掉则直接 return
    q[qt ++ ] = dist;
    for (int i = h[u]; ~i; i = ne[i])
        if (e[i] != fa)
            get_dist(e[i], u, dist + w[i], qt);
}

int get(int a[], int k) {
    sort(a, a + k);
    int res = 0;
    for (int i = k - 1, j = -1; i >= 0; i -- ) {
        while (j + 1 < i && a[j + 1] + a[i] <= m) j ++ ;
        j = min(j, i - 1);
```

```
        res += j + 1;
    }

    return res;
}

int calc(int u) {
    if (st[u]) return 0;
    int res = 0;
    get_wc(u, -1, get_size(u, -1), u);
    st[u] = true;  // 删除重心

    // 重点 ------------------------------------------
    int pt = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i], qt = 0;
        get_dist(j, -1, w[i], qt);
        res -= get(q, qt);
        for (int k = 0; k < qt; k ++ ) {
            if (q[k] <= m) res ++ ;
            p[pt ++ ] = q[k];
        }
    }
    res += get(p, pt);
    // 归并部分----------------------------------------

    for (int i = h[u]; ~i; i = ne[i]) res += calc(e[i]);
    return res;
}

int main()
{
    while (scanf("%d%d", &n, &m), n || m)
    {
        memset(st, 0, sizeof st);
        memset(h, -1, sizeof h);
        idx = 0;
        for (int i = 0; i < n - 1; i ++ )
        {
            int a, b, c;
            scanf("%d%d%d", &a, &b, &c);
            add(a, b, c), add(b, a, c);
        }

        printf("%d\n", calc(0));
    }

    return 0;
}
```

23. 状态压缩DP

棋盘式

题目描述

在 n × n 的棋盘上放 k 个国王，国王可攻击相邻的 8 个格子，求使它们无法互相攻击的方案总数。

输入格式

共一行，包含两个整数 n 和 k。

输出格式

共一行，表示方案总数，若不能够放置则输出0。

```
#include <iostream>
#include <cstring>
#include <algorithm>
#include <vector>

using namespace std;

const int N = 12, K = 110, M = 1 << 10;
int n, m;
long long f[N][K][M];
int cnt[M];
vector<int> states;
vector<int> head[M];

bool check(int state) {
    for (int i = 0; i < n; i ++ )
        if ((state >> i & 1) && (state >> i + 1 & 1))
            return false;
    return true;
}

int get(int state) {
    int cnt = 0;
    for (int i = 0; i < n; i ++) cnt += state >> i & 1;
    return cnt;
}

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

    for (int i = 0; i < 1 << n; i ++) {
        if (check(i)) {
            states.push_back(i);
            cnt[i] = get(i);
        }
    }

    for (int i = 0; i < states.size(); i ++ )
        for (int j = 0; j < states.size(); j ++ ) {
            int a = states[i], b = states[j];
            if ((a & b) == 0 && check(a | b))
                head[i].push_back(j);
        }
```

```
    f[0][0][0] = 1;
    for (int i = 1; i <= n + 1; i ++) {
        for (int j = 0; j <= m; j ++) {
            for (int a = 0; a < states.size(); a ++ )
                for (int b : head[a]) {
                    int c = cnt[states[a]];
                    if (j >= c) {
                        f[i][j][a] += f[i - 1][j - c][b];
                    }
                }
        }
    }

    cout << f[n + 1][m][0] << endl;

    return 0;
}
```

棋盘式 + 矩阵乘法 + 快速幂

题目描述

给出一个 n×m 的方格图，现在要用如下 L 型的积木拼到这个图中，使得方格图正好被拼满，请问总共有多少种拼法。

其中，方格图的每一个方格正好能放积木中的一块。

积木可以任意旋转。

输入格式

输入的第一行包含两个整数 n,m，表示方格图的大小。

输出格式

输出一行，表示可以放的方案数，由于方案数可能很多，所以请输出方案数除以 109+7 的余数。

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

using namespace std;

typedef long long LL;
const int N = 130, MOD = 1e9 + 7;

LL n;
int m;
int w[N][N];

// 这里将原 n * m 方格 看为 m * n 方格，便于理解
// x 为 当前列状态，y 为 下一列状态，u 为当前列枚举到的方格 index
void dfs(int x, int y, int u) {
    if (u == m) w[x][y] ++ ;
    else if (x >> u & 1) dfs(x, y, u + 1);
    else {
        // 由于 列 从左往右，u(行) 从下往上 枚举，因此共有四种转移方式
        if (u && !(y >> u & 1) && !(y >> u - 1 & 1))
```

```
            dfs(x, y + (1 << u) + (1 << u - 1), u + 1);
        if (u + 1 < m && !(y >> u & 1) && !(y >> u + 1 & 1))
            dfs(x, y + (1 << u) + (1 << u + 1), u + 1);
        if (u + 1 < m && !(x >> u + 1 & 1)) {
            if (!(y >> u & 1)) dfs(x, y + (1 << u), u + 2);
            if (!(y >> u + 1 & 1)) dfs(x, y + (1 << u + 1), u + 2);
        }
    }
}

// 矩阵乘法模板
void mul(int c[][N], int a[][N], int b[][N]) {
    static int tmp[N][N];
    memset(tmp, 0, sizeof tmp);
    for (int i = 0; i < 1 << m; i ++ )
        for (int j = 0; j < 1 << m; j ++ )
            for (int k = 0; k < 1 << m; k ++ )
                tmp[i][j] = (tmp[i][j] + (LL)a[i][k] * b[k][j]) % MOD;
    memcpy(c, tmp, sizeof tmp);
}

int main() {
    cin >> n >> m;
    // 构造转移矩阵
    for (int i = 0; i < 1 << m; i ++ )
        dfs(i, 0, 0);

    int res[N][N] = {0};
    res[0][(1 << m) - 1] = 1;

    // 快速幂
    while (n) {
        if (n & 1) mul(res, res, w);
        mul(w, w, w);
        n >>= 1;
    }
    cout << res[0][(1 << m) - 1] << endl;
    return 0;
}
```

0 1 覆盖问题

题目描述

T 市有 N 个酒店，这些酒店由 N-1 条双向道路连接，所有酒店和道路构成一颗树。

不同的道路可能有不同的长度，运输车通过该道路所需要的时间受道路的长度影响。在 T 市，一共有 K 种主流食材。莱莱公司有 K 辆车，每辆车负责一种食材的配送，不存在多辆车配送相同的食材。

由于不同酒店的特点不同，因此不同酒店对食材的需求情况也不同，比如可能 1 号酒店只需要第 1,5 种食材， 2 号酒店需要全部的 K 种食材.莱莱公司每天给这些公司运输食材。对于运输第 i 种食材的车辆，这辆车可以从任意酒店出发，然后将食材运输到所有需要第 i 种食材的酒店.为了提高配送效率，这 K 辆车可以从不同的酒店出发。但是由于 T 市对于食品安全特别重视，因此每辆车在配送之前需要进行食品安全检查。鉴于进行食品安全检查的人手不足，最多可以设置 M 个检查点。

现在莱莱公司需要你制定一个运输方案：选定不超过 M 个酒店设立食品安全检查点，确定每辆运输车从哪个检查点出发，规划每辆运输车的路线.假设所有的食材运输车在进行了食品安全检查之后同时出发，请制定一个运输方案，使得所有酒店的等待时间的最大值最小。酒店的等待时间从运输车辆出发时开始计算，到该酒店所有需要的食材都运输完毕截至。

如果一个酒店不需要任何食材，那么它的等待时间为 0。

输入格式

第一行包含 3 个正整数 N,M,K，含义见题目描述。

接下来 N 行，每行包含 K 个整数。每行输入描述对应酒店对每种食材的需求情况，1 表示需要对应的食材， 0 表示不需要。

接下来 N−1 行，每行包含 3 个整数 u,v,w，表示存在一条通行时间为 w 的双向道路连接 u 号酒店和 v 号酒店。

保证输入数据是一颗树，酒店从 1 编号到 N。

输出格式

输出一个整数，表示在你的方案中，所有酒店的等待时间的最大值。

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

#define x first
#define y second

using namespace std;

typedef pair<int, int> PII;
const int N = 110, M = 10, S = 1 << M;

int n, m, k;
int need[N][M];
int h[N], e[N * 2], w[N * 2], ne[N * 2], idx;
int d[N][M];
int f[S], state[N];

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

PII dfs(int u, int fa, int v) {
    PII res(0, -1);
    if (need[u][v]) res.y = 0;
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (j == fa) continue;
        auto t = dfs(j, u, v);
        if (t.y != -1) {
            res.x += t.x + w[i] * 2;
            res.y = max(res.y, t.y + w[i]);
        }
    }
    return res;
}

bool check(int mid) {
    memset(state, 0, sizeof state);
    for (int i = 1; i <= n; i ++ )
        for (int j = 0; j < k; j ++ )
            if (d[i][j] <= mid) // 构建 0 1 矩阵(共 n 行, k 列)
                state[i] |= 1 << j;
```

```
    memset(f, 0x3f, sizeof f);
    f[0] = 0;
    for (int i = 0; i < 1 << k; i ++ )
        for (int j = 1; j <= n; j ++ )
            f[i | state[j]] = min(f[i | state[j]], f[i] + 1);

    return f[(1 << k) - 1] <= m;
}

int main() {
    cin >> n >> m >> k;
    for (int i = 1; i <= n; i ++ )
        for (int j = 0; j < k; j ++ )
            cin >> need[i][j];

    memset(h, -1, sizeof h);
    for (int i = 0; i < n - 1; i ++ ) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c), add(b, a, c);
    }

    // 以第 i 个店为起点，到满足所有第 j 种食材的供应所需最小距离
    for (int i = 1; i <= n; i ++ )
        for (int j = 0; j < k; j ++ ) {
            auto t = dfs(i, -1, j);
            if (t.y != -1) d[i][j] = t.x - t.y;
        }

    int l = 0, r = 2e8;
    while (l < r) {
        int mid = l + r >> 1;
        if (check(mid)) r = mid;
        else l = mid + 1;
    }

    printf("%d\n", r);
    return 0;
}
```

24. 状态机DP

KMP

你现在需要设计一个密码 S，S 需要满足：
    S 的长度是 N；
    S 只包含小写英文字母；
    S 不包含子串 T；
例如：abc 和 abcde 是 abcde 的子串，abd 不是 abcde 的子串。
请问共有多少种不同的密码满足要求？
由于答案会非常大，请输出答案模 109+7 的余数。

输入格式
    第一行输入整数N，表示密码的长度。
    第二行输入字符串T，T中只包含小写字母。

输出格式

输出一个正整数，表示总方案数模 109+7 后的结果。

数据范围

1≤N≤50,

1≤|T|≤N，|T|是T的长度。

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

const int N = 55, mod = 1e9 + 7;

int n, m;
char str[N];
int ne[N];
int f[N][N];

int main() {
    cin >> n >> str + 1;

    m = strlen(str + 1);

    for (int i = 2, j = 0; i <= m; i ++ ) {
        while (j && str[i] != str[j + 1]) j = ne[j];
        if (str[i] == str[j + 1]) j ++ ;
        ne[i] = j;
    }

    f[0][0] = 1;
    for (int i = 0; i < n; i ++ )
        for (int j = 0; j < m; j ++ )
            for (char k = 'a'; k <= 'z'; k ++ ) {
                int u = j;
                while (u && k != str[u + 1]) u = ne[u];
                if (k == str[u + 1]) u ++ ;

                if (u < m) f[i + 1][u] = (f[i + 1][u] + f[i][j]) % mod;
            }

    int res = 0;
    for (int i = 0; i < m; i ++ ) res = (res + f[n][i]) % mod;

    cout << res << endl;

    return 0;
}
```

AC自动机

题目描述

DNA看作是一个由‘A’，‘G’，‘C’，‘T’构成的字符串。修复技术就是通过改变字符串中的一些字符，从而消除字符串中包含的致病片段。

例如，我们可以通过改变两个字符，将DNA片段”AAGCAG”变为”AGGCAC”，从而使得DNA片段中不再包含致病片段”AAG”，”AGC”，”CAG”，以达到修复该DNA片段的目的。需注意，被修复的DNA片段中，仍然只能包含字符‘A’，‘G’，‘C’，‘T’。

请你帮助生物学家修复给定的DNA片段，并且修复过程中改变的字符数量要尽可能的少。

输入格式

输入包含多组测试数据。每组数据第一行包含整数N，表示致病DNA片段的数量。

接下来N行，每行包含一个长度不超过20的非空字符串，字符串中仅包含字符‘A’，‘G’，‘C’，‘T’，用以表示致病DNA片段,再一行，包含一个长度不超过1000的非空字符串，字符串中仅包含字符‘A’，‘G’，‘C’，‘T’，用以表示待修复DNA片段。

最后一组测试数据后面跟一行，包含一个0，表示输入结束。

输出格式

每组数据输出一个结果，每个结果占一行。输入形如”Case x: y”，其中x为测试数据编号（从1开始），y为修复过程中所需改变的字符数量的最小值，如果无法修复给定DNA片段，则y为”-1”。

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

using namespace std;

const int N = 1010;

int n, m;
int tr[N][4], dar[N], idx;
int q[N], ne[N];
char str[N];

int f[N][N];

int get(char c) {
    if (c == 'A') return 0;
    if (c == 'T') return 1;
    if (c == 'G') return 2;
    return 3;
}

void insert() {
    int p = 0;
    for (int i = 0; str[i]; i ++ ) {
        int t = get(str[i]);
        if (tr[p][t] == 0) tr[p][t] = ++ idx;
        p = tr[p][t];
    }

    dar[p] = 1;  // 终结符标识
}

void build() {
    int hh = 0, tt = -1;
```

```
    for (int i = 0; i < 4; i ++ )
        if (tr[0][i])
            q[ ++ tt] = tr[0][i];

    while (hh <= tt) {
        int t = q[hh ++ ];
        for (int i = 0; i < 4; i ++ ) {
            int p = tr[t][i];
            if (!p) tr[t][i] = tr[ne[t]][i];
            else {
                ne[p] = tr[ne[t]][i];
                q[ ++ tt] = p;
                dar[p] |= dar[ne[p]];
            }
        }
    }
}

int main() {
    int T = 1;
    while (scanf("%d", &n), n) {
        memset(tr, 0, sizeof tr);
        memset(dar, 0, sizeof dar);
        memset(ne, 0, sizeof ne);
        idx = 0;

        for (int i = 0; i < n; i ++ ) {
            scanf("%s", str);
            insert();
        }

        build();

        scanf("%s", str + 1);
        m = strlen(str + 1);

        memset(f, 0x3f, sizeof f);
        f[0][0] = 0;
        for (int i = 0; i < m; i ++ )
            for (int j = 0; j <= idx; j ++ )
                for (int k = 0; k < 4; k ++ ) {
                    int t = get(str[i + 1]) != k;
                    int p = tr[j][k];
                    if (!dar[p]) f[i + 1][p] = min(f[i + 1][p], f[i][j] + t);
                }

        int res = 0x3f3f3f3f;
        for (int i = 0; i <= idx; i ++ ) res = min(res, f[m][i]);

        if (res == 0x3f3f3f3f) res = -1;
        printf("Case %d: %d\n", T ++, res);
    }

    return 0;
}
```

小明最近在研究一门新的语言，叫做 QQ 语言。
QQ 语言单词和文章都可以用且仅用只含有小写英文字母的字符串表示，任何由这些字母组成的字符串也都是一篇合法的 QQ 语言文章。
在 QQ 语言的所有单词中，小明选出了他认为最重要的 n 个。使用这些单词，小明可以评价一篇 QQ 语言文章的“重要度”。
文章“重要度”的定义为：在该文章中，所有重要的 QQ 语言单词出现次数的总和。其中多次出现的单词，不论是否发生包含、重叠等情况，每次出现均计算在内。

例如，假设 n=2n=2，小明选出的单词是 `gvagv` 和 `agva`。
在文章 `gvagvagvagv` 中，`gvagv` 出现了 3 次，`agva` 出现了 2 次，因此这篇文章的重要度为 3+2=5
现在，小明想知道，一篇由 m 个字母组成的 QQ 语言文章，重要度最高能达到多少。

输入格式
输入的第一行包含两个整数 n,m表示小明选出的单词个数和最终文章包含的字母个数。
接下来 n 行，每行包含一个仅由英文小写字母构成的字符串，表示小明选出的这 n 个单词。

输出格式
输出一行一个整数，表示由 m 个字母组成的 QQ 语言文章中，重要度最高的文章的重要度。

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

using namespace std;

typedef long long LL;
const int N = 110;
const LL INF = 1e18;

int n;
LL m;
int tr[N][26], cnt[N], ne[N], idx;
int q[N];
LL ans[N][N], w[N][N];

void insert(char* str) {
    int p = 0;
    for (int i = 0; str[i]; i ++ ) {
        int u = str[i] - 'a';
        if (!tr[p][u]) tr[p][u] = ++ idx;
        p = tr[p][u];
    }
    cnt[p] ++ ;
}

void build() {
    int hh = 0, tt = -1;
    for (int i = 0; i < 26; i ++ )
        if (tr[0][i])
            q[ ++ tt] = tr[0][i];
    while (hh <= tt) {
        int t = q[hh ++ ];
        for (int i = 0; i < 26; i ++ ) {
```

```
            int p = tr[t][i];
            if (!p) tr[t][i] = tr[ne[t]][i];
            else {
                ne[p] = tr[ne[t]][i];
                // 本题需要存一下当前节点是多少模式串的终点
                cnt[p] += cnt[ne[p]];
                q[ ++ tt] = p;
            }
        }
    }
}

void mul(LL c[][N], LL a[][N], LL b[][N]) {
    static LL tmp[N][N];
    memset(tmp, -0x3f, sizeof tmp);
    for (int i = 0; i <= idx; i ++ )
        for (int j = 0; j <= idx; j ++ )
            for (int k = 0; k <= idx; k ++ )
                tmp[i][j] = max(tmp[i][j], a[i][k] + b[k][j]);
    memcpy(c, tmp, sizeof tmp);
}

int main() {
    cin >> n >> m;
    char str[N];
    while (n -- ) {
        cin >> str;
        insert(str);
    }
    build();

    memset(w, -0x3f, sizeof w);
    for (int i = 0; i <= idx; i ++ )
        for (int j = 0; j < 26; j ++ ) {
            int k = tr[i][j];
            w[i][k] = max(w[i][k], (LL)cnt[k]);
        }

    for (int i = 1; i <= idx; i ++ ) ans[0][i] = -INF;
    while (m) {
        if (m & 1) mul(ans, ans, w);
        mul(w, w, w);
        m >>= 1;
    }
    LL res = 0;
    for (int i = 0; i <= idx; i ++ ) res = max(res, ans[0][i]);
    cout << res << endl;

    return 0;
}
```

25. 欧拉路径

对于无向图：

欧拉路径存在的充分必要条件：度数为奇数的点个数为2个或0个

欧拉回路存在的充分必要条件：度数为奇数的点个数为0个

度数为奇数的点有2个时，起点的度数不能是偶数：

题目描述

城市中有 n 个交叉路口，m 条街道连接在这些交叉路口之间，每条街道的首尾都正好连接着一个交叉路口。除开街道的首尾端点，街道不会在其他位置与其他街道相交。每个交叉路口都至少连接着一条街道，有的交叉路口可能只连接着一条或两条街道。

小明希望设计一个方案，从编号为 1 的交叉路口出发，每次必须沿街道去往街道另一端的路口，再从新的路口出发去往下一个路口，直到所有的街道都经过了正好一次。

输入格式

输入的第一行包含两个整数 n,m，表示交叉路口的数量和街道的数量，交叉路口从 1 到 n 标号。

接下来 m 行，每行两个整数 a,b(a≠b)，表示和标号为 a 的交叉路口和标号为 b 的交叉路口之间有一条街道，街道是双向的，小明可以从任意一端走向另一端。两个路口之间最多有一条街道。

输出格式

如果小明可以经过每条街道正好一次，则输出一行包含 m+1 个整数 p1,p2,p3,…,pm+1，表示小明经过的路口的顺序，相邻两个整数之间用一个空格分隔。如果有多种方案满足条件，则输出字典序最小的一种方案，即首先保证 p1 最小，p1 最小的前提下再保证 p2 最小，依此类推。如果不存在方案使得小明经过每条街道正好一次，则输出一个整数 −1。

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
#include <set>

using namespace std;

const int N = 10010, M = 100010;

int n, m;
set<int> g[N];
int p[N];
int ans[M], top;

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

// 记录一种走法
void dfs(int u) {
    while (g[u].size()) {
        int t = *g[u].begin();
        g[u].erase(t), g[t].erase(u);
        dfs(t);
    }
    ans[ ++ top] = u;
}

int main() {
```

```
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ ) p[i] = i;
    while (m -- ) {
        int a, b;
        scanf("%d%d", &a, &b);
        g[a].insert(b), g[b].insert(a);
        p[find(a)] = find(b);
    }

    int s = 0;
    for (int i = 1; i <= n; i ++ ) {
        if (find(i) != find(1)) {
            puts("-1");
            return 0;
        }
        else if (g[i].size() % 2) s ++ ;
    }

    // 存在欧拉路径 则 奇顶点个数为 0 或 2 且 如果为2，则起点度数为奇数
    if (s != 0 && s != 2 || s == 2 && g[1].size() % 2 == 0) {
        puts("-1");
        return 0;
    }

    dfs(1);

    for (int i = top; i; i --) printf("%d ", ans[i]);

    return 0;
}
```

26. AC自动机

题目描述
给定 n 个模式串 s_i 和一个文本串 t，求有多少个不同的模式串在文本串里出现过。
两个模式串不同当且仅当他们编号不同。

输入格式
第一行是一个整数，表示模式串的个数 n。
第 2 到第 (n + 1) 行，每行一个字符串，第 (i + 1) 行的字符串表示编号为 i 的模式串 s_i
最后一行是一个字符串，表示文本串 t。

输出格式
输出一行一个整数表示答案。

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

using namespace std;

struct Tree//字典树
```

```
{
    int fail;//失配指针
    int son[26];//子节点的位置
    int end;//标记有几个单词以这个节点结尾
}AC[1000010];//Trie树
int cnt = 0;//Trie的指针

inline void Build(string s)
{
        int len = s.length();
        int now = 0;//字典树的当前指针
        for(int i = 0; i < len; ++i)//构造Trie树
        {
                if(AC[now].son[s[i]-'a'] == 0)//Trie树没有这个子节点
                   AC[now].son[s[i]-'a']= ++cnt;//构造出来
                now = AC[now].son[s[i]-'a'];//向下构造
        }
        AC[now].end += 1;//标记单词结尾
}

void Get_fail()//构造fail指针
{
        queue<int> Q;//队列
        for(int i = 0; i < 26; ++i) {//第二层的fail指针提前处理一下
            if(AC[0].son[i] != 0) {
                AC[AC[0].son[i]].fail = 0;//指向根节点
                Q.push(AC[0].son[i]);//压入队列
            }
        }
        while(!Q.empty()) {//BFS求fail指针
            int u = Q.front();
            Q.pop();
            for(int i = 0;i < 26; ++i) {//枚举所有子节点(仅有小写字母 26个)
                if(AC[u].son[i] != 0) { //存在这个子节点
                    AC[AC[u].son[i]].fail=AC[AC[u].fail].son[i];
                    //子节点的fail指针指向当前节点的
                    //fail指针所指向的节点的相同子节点
                    Q.push(AC[u].son[i]);//压入队列
                }
                else AC[u].son[i] = AC[AC[u].fail].son[i];
                      //当前节点的这个子节点指向当
                      //前节点fail指针的这个子节点
            }
        }
}

int query(string s)//AC自动机匹配
{
        int l = s.length();
        int now = 0,ans = 0;
        for(int i = 0; i < l; ++i)
        {
                now = AC[now].son[s[i]-'a'];//向下一层
                for(int t = now; t && AC[t].end != -1; t = AC[t].fail)//循环求解
                {
                         ans += AC[t].end;
                         AC[t].end = -1;
                }
```

```
        }
        return ans;
}
int main()
{
     int n;
     string s;
     cin >> n;
     for(int i = 1;i <= n; ++i)
     {
             cin >> s;
             Build(s);
     }
     AC[0].fail=0;//结束标志
     Get_fail();//求出失配指针
     cin >> s;//文本串
     cout<< query(s) << endl;
     return 0;
}
```

27. 区间DP

题目描述

将 n 堆石子绕圆形操场排放，现要将石子有序地合并成一堆。
规定每次只能选相邻的两堆合并成新的一堆，并将新的一堆的石子数记做该次合并的得分。
请编写一个程序，读入堆数 n 及每堆的石子数，并进行如下计算：
选择一种合并石子的方案，使得做 n-1 次合并得分总和最大。
选择一种合并石子的方案，使得做 n-1 次合并得分总和最小。

输入格式

第一行包含整数 n，表示共有 n 堆石子。
第二行包含 n 个整数，分别表示每堆石子的数量。

输出格式

输出共两行：
第一行为合并得分总和最小值，
第二行为合并得分总和最大值。

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

using namespace std;

const int N = 410;
int n;
int a[N], s[N];
int f[N][N], g[N][N];

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

```
    for (int i = 1; i <= n; i++) cin >> a[i];
    for (int i = n + 1; i <= 2 * n; i++) a[i] = a[i - n];
    for (int i = 1; i <= 2 * n; i++) s[i] = s[i - 1] + a[i];

    memset(f, 0x3f, sizeof f);
    memset(g, -0x3f, sizeof g);
    for (int len = 1; len <= n; len ++) {
        for (int i = 1; i + len - 1 <= 2 * n; i++) {
            int l = i, r = i + len - 1;
            if (l == r) {
                f[l][r] = g[l][r] = 0;
            }
            else {
                for (int k = l; k < r; k ++) {
                    f[l][r] = min(f[l][k] + f[k + 1][r] + s[r] - s[l - 1], f[l]
[r]);
                    g[l][r] = max(g[l][k] + g[k + 1][r] + s[r] - s[l - 1], g[l]
[r]);
                }
            }
        }
    }

    int mi = 0x3f3f3f3f, ma = -0x3f3f3f3f;
    for (int l = 1; l <= n; l++) {
        mi = min(f[l][l + n - 1], mi);
        ma = max(g[l][l + n - 1], ma);
    }

    cout << mi << endl;
    cout << ma << endl;

    return 0;
}
```

28.树型DP

有一座城市，城市中有 N 个公交站，公交站之间通过 N−1 条道路连接，每条道路有相应的长度。
保证所有公交站两两之间能够通过一唯一的通路互相达到。两个公交站之间路径长度定义为两个公交站之间路径上所有边的边权和。
现在要对城市进行规划，将其中 M 个公交站定为“重要的”。
现在想从中选出 K 个节点，使得这 K 个公交站两两之间路径长度总和最小。输出路径长度总和即可（节点编号从 1 开始）。

输入格式

第 1 行包含三个正整数 N，M 和 K 分别表示树的节点数，重要的节点数，需要选出的节点数。
第 2 行包含 M 个正整数，表示 M 个重要的节点的节点编号。
接下来 N−1 行，每行包含三个正整数 a，b，c，表示编号为 a 的节点与编号为 b 的节点之间有一条权值为 c 的无向边。
每行中相邻两个数之间用一个空格分隔。

输出格式

输出只有一行，包含一个整数表示路径长度总和的最小值。

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

typedef long long LL;
const int N = 50010, M = N * 2;

int n, m, K;
int h[N], e[M], w[M], ne[M], idx;
// f[i][j] 表示以 i 为 根的树中选 j 个点，  该子树所有边对答案的贡献值
LL f[N][110];
bool st[N];
// 记录子树大小
int sz[N];
LL ans = 1e18;

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

void dfs(int u, int fa) {
    f[u][0] = 0;
    if (st[u]) f[u][1] = 0;
    sz[u] = 1;
    // 分组背包
    for (int i = h[u]; ~i; i = ne[i])  // 枚举物品组
    {
        int ver = e[i];
        // 父节点不搜
        if (ver == fa) continue;
        dfs(ver, u);
        sz[u] += sz[ver];
        for (int j = min(sz[u], K); j >= 0; j -- )  // 枚举体积
            for (int k = 0; k <= min(j, sz[ver]); k ++ )  // 枚举决策
                f[u][j] = min(f[u][j], f[u][j - k] + f[ver][k] + (LL)w[i] * k *
(K - k));
    }

    ans = min(ans, f[u][K]);
}

int main() {
    scanf("%d%d%d", &n, &m, &K);
    memset(h, -1, sizeof h);
    while (m -- ) {
        int x;
        scanf("%d", &x);
        st[x] = true;
    }
    for (int i = 0; i < n - 1; i ++ ) {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        add(a, b, c), add(b, a, c);
```

```
    }

    memset(f, 0x3f, sizeof f);
    // 第二个参数记录父节点，无向图避免往回搜
    dfs(1, -1);
    printf("%lld\n", ans);
    return 0;
}
```