

Bswoalwo`s 板子

算法基础

> 高精度

高精度比较

```
//判断a是否小于b
bool cmp(vector<int> &a, vector<int> &b)
{
    if (a.size() != b.size()) return a.size() < b.size();
    for (int i = a.size() - 1; i >= 0; i --)
        if (a[i] != b[i])
            return a[i] < b[i];
    return true;
}
```

高精度加法

```
// C = A + B, A >= 0, B >= 0
// 不传引用会慢一些
#include <bits/stdc++.h>
using namespace std;
vector<int> add(vector<int> &a, vector<int> &b) {
    vector<int> ans;
    if(b.size()>a.size()){
        return add(b,a);
    }
    int t=0;
    for(int i=0;i<a.size();i++){
        t+=a[i];
        if(i<b.size()){
            t+=b[i];
        }
        ans.push_back(t%10);
        t/=10;
    }
    if(t){
        ans.push_back(t);
    }
    return ans;
}
int main(){
    string a,b;
    cin>>a>>b;
```

```

vector<int> A, B;
for(int i=a.size()-1;i>=0;i--) {
    A.push_back(a[i]-'0');
}
for(int i=b.size()-1;i>=0;i--) {
    B.push_back(b[i]-'0');
}
//输入是顺序
vector<int> ans=add(A, B);
//输出是逆序
for(int i=ans.size()-1;i>=0;i--) {
    cout<<ans[i];
}
cout<<endl;
return 0;
}

```

高精度减法

```

#include <iostream>
#include <vector>

using namespace std;

bool cmp(vector<int> &A, vector<int> &B)
{
    if (A.size() != B.size()) return A.size() > B.size();

    for (int i = A.size() - 1; i >= 0; i -- )
        if (A[i] != B[i])
            return A[i] > B[i];

    return true;
}

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();
    return C;
}

```

```

int main()
{
    string a, b;
    vector<int> A, B;
    cin >> a >> b;
    for (int i = a.size() - 1; i >= 0; i --) A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; i --) B.push_back(b[i] - '0');

    vector<int> C;

    if (cmp(A, B)) C = sub(A, B);
    else C = sub(B, A), cout << '-';

    for (int i = C.size() - 1; i >= 0; i --) cout << C[i];
    cout << endl;

    return 0;
}

```

高精度乘法

高精度*低精度

```

#include <iostream>
#include <vector>

using namespace std;

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();
    return C;
}

int main()
{
    string a;
    int b;
    cin >> a >> b;
    vector<int> A;

```

```

        for (int i = a.size() - 1; i >= 0; i--) A.push_back(a[i] - '0');
        auto C = mul(A, b);
        for (int i = C.size() - 1; i >= 0; i--) printf("%d", C[i]);
        return 0;
    }
}

```

高精度*高精度

```

#include <iostream>
#include <vector>
using namespace std;
vector<int> mul(vector<int> &A, vector<int> &B) {
    vector<int> C(A.size() + B.size() + 7, 0); // 初始化为 0, C的size可以大一点

    for (int i = 0; i < A.size(); i++)
        for (int j = 0; j < B.size(); j++)
            C[i + j] += A[i] * B[j];

    int t = 0;
    for (int i = 0; i < C.size(); i++) { // i = C.size() - 1时 t 一定小于 10
        t += C[i];
        C[i] = t % 10;
        t /= 10;
    }

    while (C.size() > 1 && C.back() == 0) C.pop_back(); // 必须要去前导 0, 因为最高位很可能是 0
    return C;
}

int main() {
    string a, b;
    cin >> a >> b;
    vector<int> A, B;
    for (int i = a.size() - 1; i >= 0; i--)
        A.push_back(a[i] - '0');
    for (int i = b.size() - 1; i >= 0; i--)
        B.push_back(b[i] - '0');
    auto C = mul(A, B);
    for (int i = C.size() - 1; i >= 0; i--)
        cout << C[i];
    return 0;
}

```

高精度除法

```

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

```

```

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;
}

int main()
{
    string a;
    vector<int> A;

    int B;
    cin >> a >> B;
    for (int i = a.size() - 1; i >= 0; i -- ) A.push_back(a[i] - '0');

    int r;
    auto C = div(A, B, r);

    for (int i = C.size() - 1; i >= 0; i -- ) cout << C[i];

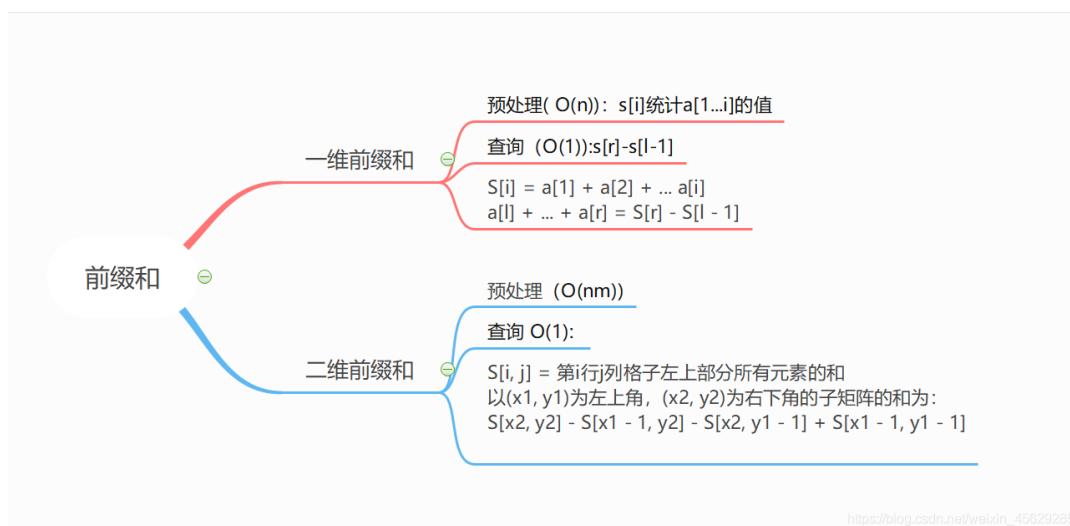
    cout << endl << r << endl;

    return 0;
}

```

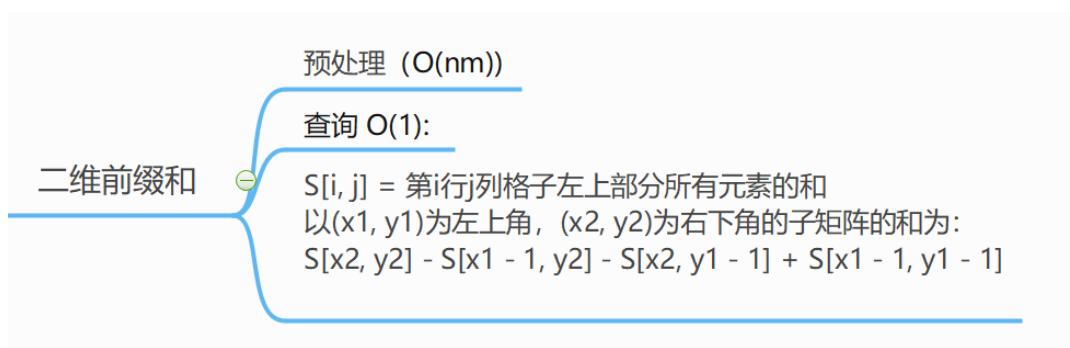
> 前缀和

一维前缀和



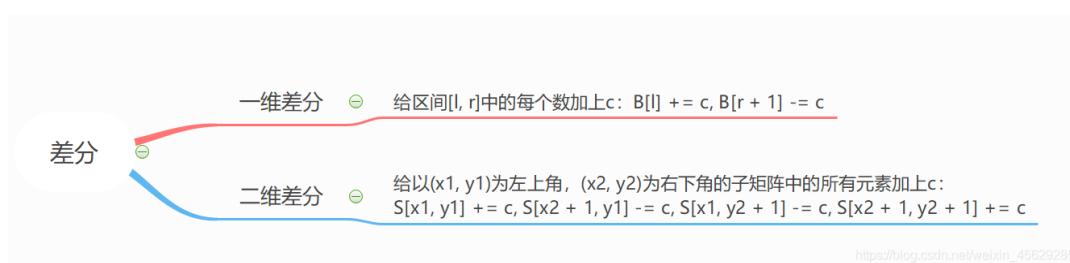
```
for (int i = 1; i <= n; i++) s[i] = s[i - 1] + a[i]; // 一维前缀和的初始化
```

二维前缀和



```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++)
        s[i][j] += s[i - 1][j] + s[i][j - 1] - s[i - 1][j - 1]; // 二维前缀和的初始化
```

> 差分



https://blog.csdn.net/weixin_45629285

一维差分

a 数组是 **b** 数组的前缀和数组, 反过来我们把 **b** 数组叫做 **a** 数组的差分数组。换句话说, 每一个 **a[i]** 都是 **b** 数组中从头开始的一段区间和。

差分可以看成前缀和的逆运算

$b[1] + c$, 效果使得 a 数组中 $a[1]$ 及以后的数都加上了 c , 但我们只要求 1 到 r 区间加上 c , 因此还需要执行 $b[r + 1] - c$, 让 a 数组中 $a[r + 1]$ 及往后的区间再减去 c , 这样对于 $a[r]$ 以后区间的数相当于没有发生改变。

H5 例题

输入一个长度为 n 的整数序列。

接下来输入 m 个操作, 每个操作包含三个整数 l, r, c , 表示将序列中 $[l, r]$ 之间的每个数加上 c 。请你输出进行完所有操作后的序列。

输入格式

第一行包含两个整数 n 和 m 。

第二行包含 n 个整数, 表示整数序列。

接下来 m 行, 每行包含三个整数 l, r, c , 表示一个操作。

输出格式

共一行, 包含 n 个整数, 表示最终序列。

AC代码

```
#include<iostream>
using namespace std;
const int N = 1e5 + 10;
int a[N], b[N];
int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
    for(int i = 1; i <= n; i++)
    {
        scanf("%d", &a[i]);
        b[i] = a[i] - a[i - 1];           //构建差分数组, 差分初始化
    }
    int l, r, c;
    while(m--)
    {
        scanf("%d%d%d", &l, &r, &c);
        b[l] += c;                      //表示将序列中[l, r]之间的每个数加上c
        b[r + 1] -= c;
    }
    for(int i = 1; i <= n; i++)
    {
        b[i] += b[i - 1];             //求前缀和运算
        printf("%d ", b[i]);
    }
    return 0;
}
```

二维差分

始终要记得, a 数组是 b 数组的前缀和数组, 比如对 b 数组的 $b[i][j]$ 的修改, 会影响到 a 数组中从 $a[i][j]$ 及往后的每一个数。

已知原数组 a 中被选中的子矩阵为以 (x_1, y_1) 为左上角，以 (x_2, y_2) 为右下角所围成的矩形区域，要使被选中的子矩阵中的每个元素的值加上 c ，则构造好的差分数组 b 将执行一下操作

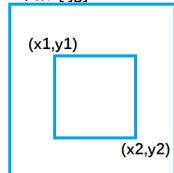
$b[x_1][y_1] += c$

$b[x_1,][y_2+1] -= c$

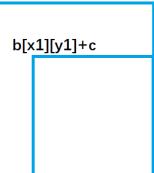
$b[x_2+1][y_1] -= c$

$b[x_2+1][y_2+1] += c$

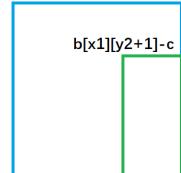
目的：让小蓝色矩形中的每一个数 $a[i][j] += c$



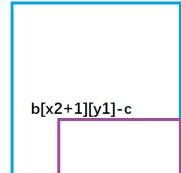
1



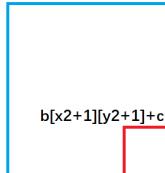
2



3



4



<http://www.cnblogs.com/wangzhiqiang/>

将上述操作写成一个函数

```
void insert(int x1, int y1, int x2, int y2, int c)
{
    //对b数组执行插入操作，等价于对a数组中的(x1, y1)到(x2, y2)之间的元素都加上了c
    b[x1][y1] += c;
    b[x2 + 1][y1] -= c;
    b[x1][y2 + 1] -= c;
    b[x2 + 1][y2 + 1] += c;
}
```

那么初始化就可以用这个函数

```
for(int i = 1; i <= n; i++)
{
    for(int j = 1; j <= m; j++)
    {
        insert(i, j, i, j, a[i][j]);      //构建差分数组
    }
}
// 对于初始化也有另外一种方法：b[i][j] = a[i][j] - a[i - 1][j] - a[i][j - 1] + a[i - 1][j - 1]
```

H5 例题

输入一个 n 行 m 列的整数矩阵，再输入 q 个操作，每个操作包含五个整数 x_1, y_1, x_2, y_2, c ，其中 (x_1, y_1) 和 (x_2, y_2) 表示一个子矩阵的左上角坐标和右下角坐标。

每个操作都要将选中的子矩阵中的每个元素的值加上 c 。

请你将进行完所有操作后的矩阵输出。

输入格式

第一行包含整数 n, m, q 。

接下来 n 行，每行包含 m 个整数，表示整数矩阵。

接下来 q 行，每行包含 5 个整数 x_1, y_1, x_2, y_2, c ，表示一个操作。

输出格式

共 n 行，每行 m 个整数，表示所有操作进行完毕后的最终矩阵。

AC代码

```
#include<iostream>
#include<cstdio>
using namespace std;
const int N = 1e3 + 10;
int a[N][N], b[N][N];
void insert(int x1, int y1, int x2, int y2, int c)
{
    b[x1][y1] += c;
    b[x2 + 1][y1] -= c;
    b[x1][y2 + 1] -= c;
    b[x2 + 1][y2 + 1] += c;
}
int main()
{
    int n, m, q;
    cin >> n >> m >> q;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            cin >> a[i][j];
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            insert(i, j, i, j, a[i][j]);           //构建差分数组
        }
    }
    while (q--)
    {
        int x1, y1, x2, y2, c;
        cin >> x1 >> y1 >> x2 >> y2 >> c;
        insert(x1, y1, x2, y2, c);
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            b[i][j] += b[i - 1][j] + b[i][j - 1] - b[i - 1][j - 1]; //二维前缀和
        }
    }
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= m; j++)
        {
            printf("%d ", b[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

lowbit

lowbit(x)是x的二进制表达式中最低位的1所对应的值

```
int lowbit(int x){  
    return x&(-x);  
}
```

> 搜索

BFS

H5 Flood fill

可以在线性的复杂度内，求出某个点所在的联通块，也可以求联通块数量

```
typedef pair<int, int> PII;  
const int N=1010;  
const int dir[] [2]={ {1, 1}, {-1, 1}, {1, -1}, {0, 1}, {0, -1}, {1, 0}, {-1, 0}, {-1, -1} } ;  
char g[N] [N];  
int n, m;  
bool st[N] [N];  
void bfs(int sx, int sy) {  
    st[sx] [sy]=1;  
    queue<PII> q;  
    q.push({sx, sy});  
    while(!q. empty()) {  
        auto t=q. front();  
        q. pop();  
        int x=t. first, y=t. second;  
        for(int k=0;k<8;k++) {  
            int xx=x+dir[k] [0];  
            int yy=y+dir[k] [1];  
            if(xx>=1 && xx<=n && yy>=1 && yy<=m && !st[xx] [yy] && g[xx] [yy]=='W') {  
                q. push({xx, yy});  
                st[xx] [yy]=1;  
            }  
        }  
    }  
}  
void solve() {  
    cin>>n>>m;  
    for(int i=1;i<=n;i++) {  
        for(int j=1;j<=m;j++) {  
            cin>>g[i] [j];  
        }  
    }  
    int ans=0;  
    for(int i=1;i<=n;i++) {
```

```

        for(int j=1;j<=m;j++) {
            if(g[i][j]=='W' && !st[i][j]) {
                bfs(i,j);
                ans++;
            }
        }
    }
    cout<<ans<<'\n';
}

```

深搜和宽搜都可以实现，但是由于深搜可能会爆栈，能用宽搜尽量用宽搜

H5 最短路模型+求具体方案

迷宫问题

```

const int dir[] [2]={ {1, 0}, {-1, 0}, {0, 1}, {0, -1} } ;
int g[N] [N];
PII pre[N] [N];//存储路径
int n;
void bfs(int sx, int sy) {
    queue<PII> q;
    q.push({sx, sy});
    memset(pre, -1, sizeof pre);
    pre[sx][sy]= {0, 0};
    while(!q.empty()) {
        auto t=q.front();
        q.pop();
        int x=t.first, y=t.second;
        for(int k=0;k<8;k++) {
            int xx=x+dir[k][0];
            int yy=y+dir[k][1];
            if(xx>=0 && xx<n && yy>=0 && yy<n && !g[xx][yy] && pre[xx][yy].first== -1) {
                q.push({xx, yy});
                pre[xx][yy]=t;
            }
        }
    }
}
void solve() {
    cin>>n;
    for(int i=0;i<n;i++) {
        for(int j=0;j<n;j++) {
            cin>>g[i][j];
        }
    }
    //逆向搜索，顺序输出具体方案
    bfs(n-1, n-1);
    PII t={0, 0};
    while(1) {
        int x=t.first, y=t.second;
        cout<<x<< ' '<<y<<'\n';

```

```

        if(x==n-1 && y==n-1) break;
        t=pre[x][y];
    }
}

```

H5 最小步数

一个地图对应一种状态，一种状态到另一种状态的最小步数，也可以把一种状态对应一个点

H5 多源BFS

建立一个虚拟源点，虚拟源点到所有起点的距离为0

```

typedef pair<int, int> PII;
const int N=1010;
const int dir[][][2]={{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
int dist[N][N];
char g[N][N];
int n, m;
void bfs() {
    memset(dist, -1, sizeof dist);
    queue<PII> q;
    // 遍历一遍，将所有起点放入队列中就是建立虚拟源点的过程
    for(int i=1; i<=n; i++) {
        for(int j=1; j<=m; j++) {
            if(g[i][j]== '1') q.push({i, j}), dist[i][j]=0;
        }
    }
    while(!q.empty()) {
        auto t=q.front();
        q.pop();
        int x=t.first;
        int y=t.second;
        for(int k=0; k<4; k++) {
            int xx=x+dir[k][0];
            int yy=y+dir[k][1];
            if(xx>=1 && xx<=n && yy>=1 && yy<=m && dist[xx][yy]==-1) {
                dist[xx][yy]=dist[x][y]+1;
                q.push({xx, yy});
            }
        }
    }
}

```

H5 双端队列广搜

双端队列主要解决图中边的权值只有0或者1的最短路问题

如果边权为1放在队尾，如果边权为0则放在队头。

广搜优化

H5 双向广搜

从起点开始搜，从终点开始搜。

常用于最小步数模型里，最短路和flood fill里数据范围一般不大，可以线性求出，而最小步数模型里状态数量是指指数级的

小技巧：两边同时进行搜索时，选择队列中元素数量小的进行扩展

字符串变换

```
int f(queue<string>& qa, unordered_map<string, int>& da, unordered_map<string, int>& db, string a[], string b[]) {
    int d=da[qa.front()];
    while(qa.size() && d==da[qa.front()]) {
        string t=qa.front();
        qa.pop();
        for(int j=0;j<n;j++) {
            for(int i=0;i<t.size();i++) {
                if(t.substr(i,a[j].size())==a[j]) {
                    string s=t.substr(0,i)+b[j]+t.substr(i+a[j].size());
                    if(da.count(s)) continue;
                    if(db.count(s)) {
                        return da[t]+db[s]+1;
                    }
                    da[s]=da[t]+1;
                    qa.push(s);
                }
            }
        }
    }
    return 11;
}
int bfs()
{
    if (A == B) return 0;
    queue<string> qa, qb;
    unordered_map<string, int> da, db;
    qa.push(A), qb.push(B);
    da[A] = db[B] = 0;
    int step = 0;
    while (qa.size() && qb.size())
    {
        int t;
        if (qa.size() < qb.size()) t = f(qa, da, db, a, b);
        else t = f(qb, db, da, b, a);
        if (t <= 10) return t;
        if (++step == 10) return -1;
    }
    return -1;
}
```

H5 A*

A*算法就是加了估价函数求最短路，每次从堆中取出 $dis[i]+g[i]$ 最小的那个节点，其中 $g[i]$ 是 i 到目标节点的估计距离，这个估计距离不能比实际距离大。

这样可以避免一种情况：当前节点 dis 最小，但是实际上到达目标时花费较大，这样就增加了无用操作。有许多例子，比如网格图中估价函数可以是曼哈顿距离。

启发函数

启发函数，会以任意“状态”作为输入，计算从该状态到目标状态所需代价的估计值。

在搜索中，仍然维护一个堆，不断从中取出“当前代价 + 未来估计（启发函数的值）”最小状态进行扩展

基本准则：保证第一次从堆中取出目标状态时达到最优解。

设当前状态 state 到目标状态所需代价的估计值为 $f(state)$ 。

设在未来的搜索算法中，实际求出的从当前状态 state 到目标状态的最小代价为 $g(state)$

对于任意的 state，应该有 $f(state) \leq g(state)$

即：启发函数的值（估价）不能大于未来实际代价，估价比实际代价更优。

A* 算法只能保证 在终点出队之后，到终点的距离是最优的，并不能保证在中间某个状态下是最优的，并且每个点不是只被 扩展一次。

八数码

八数码有解的条件是将数字从上到下，从左到右排列，逆序对的数量是偶数则有解，是奇数则无解

$N \times N$ 的棋盘， N 为奇数时，与八数码问题相同。

N 为偶数时，空格每上下移动一次，奇偶性改变。称空格位置所在的行到目标空格所在的行步数为空格的距离（不计左右距离），若两个状态的可相互到达，则有，**两个状态的逆序奇偶性相同且空格距离为偶数，或者，逆序奇偶性不同且空格距离为奇数数**。否则不能。

也就是说，当此表达式成立时，两个状态可相互到达：(状态1的逆序数 + 空格距离)的奇偶性 == 状态2奇偶性)

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
#include <map>
#include <set>
#define ll long long
#define mkp make_pair
using namespace std;
typedef pair<int, string> PIS;
const char op[4]={'u', 'd', 'l', 'r'};
const int dir[] [2]={{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
string start, ed="12345678x";
int f(string s){ //启发函数
    int cnt=0;
    for(int i=0; i<s.size(); i++){
        if(s[i]!='x'){
            int t=s[i]-1';
```

```

        cnt+=abs(i/3-t/3)+abs(i%3-t%3);
    }
}

return cnt;
}

void bfs() {
    map<string, int> dist;
    map<string, pair<char, string>> pre;//存储路径
    priority_queue<PIs, vector<PIs>, greater<PIs>> q;
    q.push({f(start), start});
    dist[start]=0;
    while(!q.empty()) {
        auto t=q.top();
        q.pop();
        string s=t.second;
        if(s==ed) {
            break;
        }
        int x,y;
        for(int i=0;i<9;i++) {
            if(s[i]=='x') {
                x=i/3;
                y=i%3;
                break;
            }
        }
        for(int k=0;k<4;k++) {
            int xx=x+dir[k][0];
            int yy=y+dir[k][1];
            if(!(xx>=0 && xx<3 && yy>=0 && yy<3)) continue;
            string temp=s;
            swap(temp[xx*3+yy], temp[x*3+y]);
            if(dist[temp]==0 || dist[temp]>dist[s]+1) {
                dist[temp]=dist[s]+1;
                q.push({dist[temp]+f(temp), temp});
                pre[temp]={op[k], s};
            }
        }
    }
    string ans;
    while(ed!=start) {
        ans+=pre[ed].first;
        ed=pre[ed].second;
    }
    reverse(ans.begin(), ans.end());
    cout<<ans<<'\n';//输出路径
}

void solve() {
    string t;
    char c;
    for(int i=0;i<9;i++) {
        cin>>c;

```

```

    start+=c;
    if(c!=x) t+=c;
}
int cnt=0;
for(int i=0;i<8;i++) {
    for(int j=i;j<8;j++) {
        if(t[i]>t[j]) cnt++;
    }
}
if(cnt&1) {
    cout<<"unsolvable"<<'\n';
} else{
    bfs();
}
}

```

DFS

H5 基于联通性的模型 (Flood fill)

H5 剪枝

1.优化搜索顺序:大部分情况下，优先搜索分支较少的节点

2.排除等效冗余:尽量不搜索重复的状态

3.可行性剪枝:到某个状态时，发现当前结果已不是最优解，提前退出

4.最优化剪枝:与可行性剪枝类似，当发现已不是最优解时，提前退出

5.记忆化搜索 (DP)

木棍

优化(剪枝)方式:

1.枚举每根棍子时,都必须要是所有小木棍总长度的约数,这个很好理解,因为原来每根木棒长度是相同的,所以(总和) $sum = length * num$ (木棒数量);

2.要按组合的方式枚举方案

这个问题中木棒的长度跟搭配顺序没有关系;

例如:由编号为1,2,3结合的棍子;



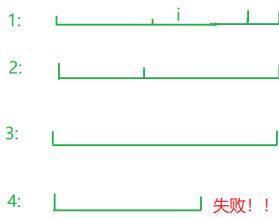
木棒还是原来一样的,所以这个是组合问题,而非排列问题,为了避免出现排列的情况,我们需要定义一个顺序来枚举由小木棍来凑合成这跟木棒;

3.定义的顺序

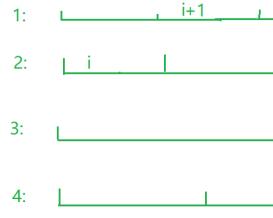
定义什么顺序可以使得分支最少呢?

由大到小的顺序,因为每个木棒的长度是不变的,枚举小木棍时,先枚举大的,后面的空间就少了,唉,这样装下的小木棍的范围不也少了吗?所以这样做可以减少后面分支!!!

1.如果第*i*根棍子失败了,如果第*i+1*根棍子跟第*i*根棍子一样长,那会不会也是失败的?



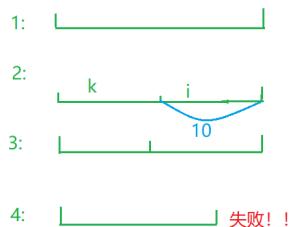
因为*i*失败了,由前面一个数在一
组只能枚举一次的规则,*i*到第2组



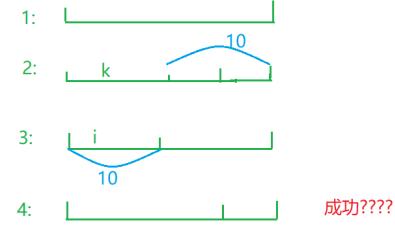
注:这里的失败指的是把*i*放在第1组的第2个位置的方案失败了

显然时错误的,因为既然成功了,那么*i*跟*i+1*换个位置,不也能成功吗???
而前面*i*放在*i+1*的位置上面,却失败了,互相矛盾!!!所以本题得证!!!

如果*i*放在最后面,该方案失败了,那么这种摆放是不是从*k*开始无论怎
么放都是错误的???

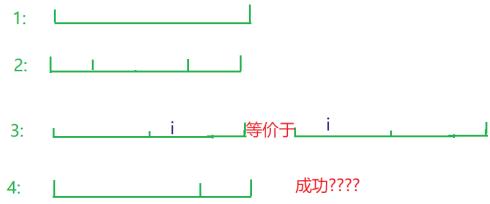
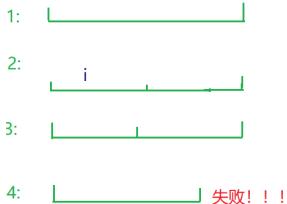


同理,把*i*放到下一组



跟上题同理,既然后一种的方案是成功的话,那么把*i*放在跟第2组后面两端交换也是可以的啊,而前
面一种方案把*i*放在第二组第二段是错误的,这样做不就互相矛盾了吗??所以得证!!!

如果*i*放在最前面的方案是失败的,那么随后的方案无论怎么放都是错的



既然后面一组是成功方案,那么把第3组换到
第2组不也是等价关系吗??而第二组把*i*段放
在前面的方式确实错误的,矛盾了!证毕

```
const int N=70;  
int a[N];  
bool st[N];  
int sum, len, n;  
//从start开始, 枚举过的不再枚举, 避免重复计算  
bool dfs(int u, int s, int start){  
    if(u*len==sum) return true;  
    if(s==len) return dfs(u+1, 0, 1);  
    for(int i=start; i<=n; i++){  
        if(a[i]+s>len || st[i]) continue;  
        st[i]=1;  
        if(dfs(u, s+a[i], start+1)) return true;  
        st[i]=0;  
        //如果是开头或者结尾的木棍失败了, 直接放弃这种方案  
        if(!s || s+a[i]==len) return false;  
        //当前这根木棍不行, 那么与它相同长度的木棍同样也不行  
        int j=i;  
        while(j<=n && a[j]==a[i]) j++;
```

```

        i=j-1;
    }
    return false;
}

void solve() {
    while(cin>>n, n) {
        sum=0;
        memset(st, 0, sizeof st);
        for(int i=1; i<=n; i++) {
            cin>>a[i]; sum+=a[i];
        }
        len=1;
        sort(a+1, a+n+1, greater<int>()); //从大到小枚举，优化搜索顺序，选择分支小的节点先搜索
        while(1) {
            //一定要是sum的约数才行
            if(sum%len==0 && dfs(0, 0, 1)) {
                cout<<len<<'\n';
                break;
            }
            len++;
        }
    }
}

```

生日蛋糕

① h_u 表示第 u 层蛋糕的高度
 设 u 为层数， r_u 表示第 u 层蛋糕的半径 h_u 最大高度为 1
 并且有 $u \leq r_u \leq r_{u+1}-1$ 又 $n-v \geq r_u^2 \cdot 1$
 $\Rightarrow u \leq h_u \leq h_{u+1}-1$ $n-v \geq r_u^2 \cdot h_u$

② v 表示之前层已使用的体积，故剩余体积为 $n-v$

计算侧面表面积 → $S_{lmu} = \frac{2}{r_{u+1}} \cdot r_i \cdot h_i$
 的表面积 $n-v = \sum_{i=1}^u r_i^2 \cdot h_i$
 由上述两式可得： $S_{lmu} = \frac{2}{r_{u+1}} \sum_{i=1}^u r_i \cdot h_i \cdot r_{u+1} > \frac{2}{r_{u+1}} \sum_{i=1}^u r_i^2 \cdot h_i = \frac{2}{r_{u+1}} \cdot (n-v)$
 故 $S_{lmu} > \frac{2}{r_{u+1}} \cdot (n-v) \Rightarrow S_{lmu} + S_{lum} < res$

$$\boxed{S + \frac{2}{r_{u+1}} \cdot (n-v) < res}$$

③ $\boxed{\begin{cases} v + \min V(u) \leq n \\ S + \min S(u) < ans \end{cases}}$

```

const int N = 25, INF = 1e9;
int n, m;
int minv[N], mins[N];
int R[N], H[N];
int ans = INF;
//当前即将处理第u层，之前层体积是v，之前层表面积是s

```

```

void dfs(int u, int v, int s) {
    //如果已经使用的体积 + 当前层往上的最小体积 已经超过了预定体积
    //则直接剪枝
    if (v + minv[u] > n) return;
    //最优性剪枝
    if (s + mins[u] >= ans) return;
    if (s + 2 * (n - v) / R[u + 1] >= ans) return;

    if (!u) {
        if (v == n) ans = s;           //体积需要全部用完一个不剩
        return;
    }

    //枚举当前层的半径r和高度h
    for (int r = min(R[u + 1] - 1, (int)sqrt(n - v)); r >= u; --r) {
        for (int h = min(H[u + 1] - 1, (n - v) / r / r); h >= u; --h) {
            int t = 0;
            if (u == m) t = r * r;    //如果当前是最底层，额外加上顶部面积
            R[u] = r, H[u] = h;
            dfs(u - 1, v + r * r * h, s + t + 2 * r * h);
        }
    }
}

int main() {
    cin >> n >> m;
    //预处理i层蛋糕的最小体积和最小表面积
    //（例如：第i层至少半径为i，这样i-1层至少i-1，…，这样第1层才有1的半径可以用了）
    for (int i = 1; i <= m; ++i) {
        minv[i] = minv[i - 1] + i * i * i;
        mins[i] = mins[i - 1] + 2 * i * i;
    }

    //第m+1层（不存在的层）设立两个哨兵
    R[m + 1] = H[m + 1] = INF;
    dfs(m, 0, 0);      //从蛋糕底层往上dfs
    if (ans == INF) ans = 0;
    cout << ans << endl;
    return 0;
}

```

H5 迭代加深

dfs迭代加深 O(h)

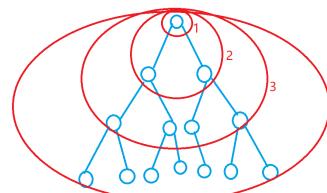
1. 应用问题
2. 基本思想
3. 区别于bfs(时间复杂度指数级别)

1. 对于搜索目标在递归树的很浅的层里，但是递归树有的分支却很深的问题
 (这类问题会使得我们在搜索的非目标分支但却很深的分支上浪费很多时间)

2. 见右图

3. bfs不会每一次都从起点开始搜判断，而迭代加深每次都
 从起点开始搜判断
 bfs是在搜的过程中会抛弃前面的点，而迭代加深不会虽然会重复搜索但是搜索前的节点个数
 相对于最后一层的节点数来说可以忽略不计，所以算法是可行的。
 (另外迭代加深可以和DA*算法结合使用)

IDA*其实就是迭代加深算法的基础上加了一个启发性的剪枝



1. 每次都要重头开始搜索判断
2. 一层一层的扩展搜索

// 迭代加深算法 一般都是解决可行性问题

```

bool dfs (int depth, int max_depth)
{
    if(depth > max_depth) return false;
    if(终止条件) return true;

    // 爆搜枚举

    return false;
}

int main ()
{
    int depth = 0;
    while(!dfs(0, depth)) depth ++ ; //*** 每一次都从0开始

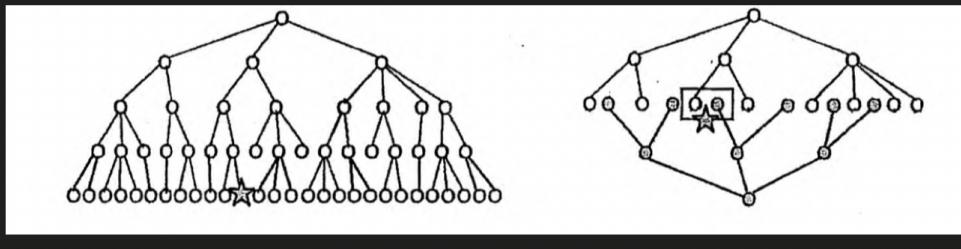
    cout << depth << endl;
}

```

h5 双向DFS

除了迭代加深之外, 双向搜索也可以避免在深层子树上浪费时间. 在一些题目中, 问题不但具有“初态”, 还具有明确的“终态”, 并且从初态开始搜索与从终态开始逆向搜索产生的搜索树都能覆盖整个状态空间. 在这种情况下, 就可以采用双向搜索---从初态和终态出发各搜索一半状态, 产生两棵深度减半的搜索树, 在中间交会, 组合成最终的答案.

如下图所示, 左侧是直接进行一次搜索产生的搜索树, 右侧是双向搜索树, 避免了层数过深时分支数量的大规模增长.



送礼物

使用空间换取时间.

折半枚举

$$\frac{N}{2} \quad | \quad 1 \quad 18 \quad 1$$

1 7 5 4

由前 $\frac{N}{2}$ 个物品能够凑出来的集合 (且最大)

里不存在当前 t + 剩下的累加和 $\leq w$



$$N_{\text{最大}} = 4^6$$

$$\frac{N}{2} = 2^3$$

每种物品最多有不选两种情况

$$t \leq w \rightarrow \boxed{1} \quad (= k)$$

△

在组合中快速输出 $\leq w \rightarrow$ 最大值放

2 的倍数

$$2^{20} \approx 10^6$$

$$2^{23} \approx 8 \times 10^6$$

这道题目就是“子集和”的拓展----从给定的 N 个数中选择几个，使他们的和最接近 w .

了解“背包”问题的人可能已经发现，这道题目也是一个“大体积”的背包问题。这类问题的直接解法就是进行“指数型”的枚举——搜索每个礼物选还是不选，时间复杂度为 $O(2^N)$ 。当然，若搜索过程中已选的礼物重量之和已经大于 w ，可以及时剪枝。

此时 $N \leq 45$, 2^{45} 的复杂度过高。这时我们就可以利用双向搜索的思想，把礼物分成两半。

首先，我们搜索出从前一半礼物中选出若干个，可能达到的 $0 \sim w$ 之间的所有重量值，存放在一个数组 A 中，并对数组 A 进行排序，去重。

然后，我们进行第二次搜索，尝试从后一半礼物中选出一些。对于每个可能达到的重量值 t ，在第一部分得到的数组 A 中二分查找 $\leq w - t$ 的数组中最大的一个，用二者的和更新答案。

这个算法的时间复杂度就只有 $O\left(2^{\frac{N}{2}} \log 2^{\frac{N}{2}}\right) = O(N * 2^{\frac{N}{2}})$ 了。我们还可以加入一些优化，进一步提高算法的效率：

1. 优化搜索顺序

把礼物按照重量降序排序后再分半，搜索。

2. 选取适当的“折半划分点”

因为第二次搜索需要再第一次搜索得到的数组中进行二分查找，效率相对较低，所以我们应该稍微增加第一次搜索的礼物数，减少第二次搜索的礼物数。经过本地随机数据实验，我们发现取第 $1 \sim \frac{N}{2} + 2$ 个礼物为“前一半”，取第 $\frac{N}{2} + 3 \sim N$ 个礼物为“后一半”时，搜索的速度最快。

```
typedef long long LL;
const int N = 1 << 25; // k最大是25，因此最多可能有  $2^{25}$  种方案
int n, m, k;
int g[50]; // 存储所有物品的重量
int weights[N]; // weights 存储能凑出来的所有的重量
int cnt = 0;
```

```

int ans; // 用ans来记录一个全局最大值
// u表示当前枚举到哪个数了， s表示当前的和
void dfs(int u, int s)
{
    // 如果我们当前已经枚举完第k个数（下标从0开始的）了， 就把当前的s， 加到weights中去
    if (u == k) {
        weights[cnt++] = s;
        return;
    }
    // 枚举当前不选这个物品
    dfs(u + 1, s);
    // 选这个物品， 做一个可行性剪枝
    if ((LL)s + g[u] <= m) { //计算和的时候转成long long防止溢出
        dfs(u + 1, s + g[u]);
    }
}
void dfs2(int u, int s)
{
    //二分查找<=v的最大值, 数据太强用stl超时了，手写二分
    if (u == n) {
        int v = m - s;
        int l = 0, r = cnt - 1;
        while (l <= r) {
            int mid = l + r >> 1;
            if (weights[mid] <= v) l = mid + 1;
            else r = mid - 1;
        }
        ans = max(ans, s + weights[l - 1]);
        return ;
    }
    // 不选择当前这个物品
    dfs2(u + 1, s);
    // 选择当前这个物品
    if ((LL)s + g[u] <= m)
        dfs2(u + 1, s + g[u]);
}
int main()
{
    cin >> m >> n;
    for (int i = 0; i < n; i++)
        cin >> g[i];
    // 优化搜索顺序（从大到小）
    sort(g, g + n);
    reverse(g, g + n);
    // 把前k个物品的重量打一个表
    k = n >> 1;
    dfs(0, 0);
    // 做完之后， 把weights数组从小到大排序
    sort(weights, weights + cnt);
    // 判重
    int t = 1;
}

```

```

        for (int i = 1; i < cnt; i++)
            if (weights[i] != weights[i - 1])
                weights[t++] = weights[i];
        cnt = t;
        // 从k开始， 当前的和是0
        dfs2(k, 0);
        cout << ans << endl;
        return 0;
    }
}

```

h5 IDA*

IDA* (迭代加深 + 估价函数)
(<= 真实值)

算法思想

像迭代加深一样搜索，当我们当前结点到根节点的距离加上估价距离(该节点到max_depth的估计距离) > max_depth, return ;

其实就相当于在迭代加深算法上多了个剪枝

IDA*算法其本质其实就是迭代加深 是迭代加深算法的一个优化

大致框架

```

// 由于迭代加深算法 一般都是解决可行性问题 所以IDA*也一般都是解决可行性问题

int f() {} // *** 估值函数

bool dfs (int depth, int max_depth)
{
    if(depth + f() > max_depth) return false;
    if(f() == 0) return true; // 这里的终止条件一般都是估值函数为0时,
    // 因为估值函数时当前状态到末状态的距离, 当等于0的时候就到了末状态了, 就结束

    // 爆搜枚举

    return false;
}

int main ()
{
    int depth = 0;
    while(!dfs(0, depth)) depth ++; //*** 每一次都从0开始

    cout << depth << endl;
}

```

> 数据结构

双端队列（滑动窗口）



```
void solve() {
    cin>>n>>k;
    for(int i=1;i<=n;i++) cin>>a[i];
    deque<int> q;
    for(int i=1;i<=n;i++) {
        while(q.size() && q.back()>a[i]) q.pop_back();
        q.push_back(a[i]);
        if(i-k>=1 && a[i-k]==q.front()) q.pop_front();
        if(i>=k) cout<<q.front()<<' ';
    }
    cout<<' \n' ;
    q.clear();
    for(int i=1;i<=n;i++) {
        while(q.size() && q.back()<a[i]) q.pop_back();
        q.push_back(a[i]);
        if(i-k>=1 && a[i-k]==q.front()) q.pop_front();
        if(i>=k) cout<<q.front()<<' ';
    }
    cout<<' \n' ;
}
```

KMP（字符串匹配）

KMP是一种高效的字符串匹配算法，用来在主字符串中查找模式字符串的位置(比如在“hello,world”主串中查找“world”模式串的位置)

```
#include <iostream>
using namespace std;
const int N = 100010, M = 10010; //N为模式串长度，M匹配串长度
int n, m;
int ne[M]; //next[]数组，避免和头文件next冲突
//next数组的含义：对next[j]，是p[1, j]串中前缀和后缀相同的最大长度（部分匹配值），即
p[1, next[j]] = p[j-ne[j]+1, j]
char s[N], p[M]; //s为模式串， p为匹配串
int main()
{
    cin >> n >> s+1 >> m >> p+1; //下标从1开始
    //求next[]数组，求前缀和后缀的匹配的最大长度，相当于与自己求一次匹配操作
    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;
    }
    //匹配操作
```

```

for(int i = 1, j = 0; i <= n; i++)
{
    // 在每次失配时, 把p串往后移动至下一次可以和前面部分匹配的位置, 这样就可以跳过大多数的
    // 失配步骤。而每次p串移动的步数就是通过查找next[]数组确定的
    while(j && s[i] != p[j+1]) j = ne[j];
    if(s[i] == p[j+1]) j++;
    if(j == m) //满足匹配条件, 打印开头下标, 从0开始
    {
        //匹配完成后的具体操作
        //如: 输出以0开始的匹配子串的首字母下标
        //printf("%d ", i - m); (若从1开始, 加1)
        j = ne[j]; //再次继续匹配
    }
}
return 0;
}

```

trie树（字典树）

高效的存储和查找字符串集合的数据结构

从根节点开始, 判断有没有该类字符, 有就向下, 没有就添加叶节点, 依次存储, 把所有结尾点标记一下, 然后用Trie高速查找某一个字符出现的次数。

qwerwer
qwe
qerer
ertq
yuyu

先是依次存储qwerwer (在字母r处标记单词结尾), 然会第二段有qwe就不用添加新的叶节点 (且在字母r处标记表示这里存在一个单词), 第三段因为e不相同, 所以要添加一个叶节点用于存储erer, 此后的基本类似, 有就存储没有就添加

```

const int N=1e5+10;
int tr[N][26], idx;
string s;
int n;
int cnt[N];
void insert() {
    //从根节点开始依次插入, 如果没有就新建一条路, 如果有就走原来存在的路
    int p=0;
    for(int i=0;s[i];i++) {
        int t=s[i]-'a';
        //如果没有找到则开一个新的下标存储
        //p表示当前节点, t表示它的儿子
        if(!tr[p][t]) tr[p][t]=++idx;
        //继续往下走
        p=tr[p][t];
    }
    cnt[p]++; //p指的是1~idx里某一个下标, 表示以当前下标(对应单词中的最后一个字母)结尾的单词有
}
int query() {
    //从根节点开始查找
}

```

```

int p=0;
for(int i=0;s[i];i++) {
    int t=s[i]-'a';
    //没有找到就说明没有这个单词，直接返回这个单词存在的次数为0
    if(!tr[p][t]) return 0;
    p=tr[p][t];
}
//返回以该单词的最后一个字母结尾的单词有多少个
return cnt[p];
}

void solve() {
    cin>>n;
    while(n--) {
        char op;
        cin>>op>>s;
        if(op=='I') {
            insert();
        }else{
            cout<<query()<<' \n';
        }
    }
}

```

最大异或对

异或性质+前缀+字典树

一个整数,是可以转化成为一个32位的二进制数,而也就可以变成长度为32位的二进制字符串.

每一次检索的时候,我们都走与当前\$A_i\$这一位相反的位置走,也就是让Xor值最大,如果说没有路可以走的话,那么就走相同的路.

这样就可以遍历所有的情况,对于每一个数字选择的都是前面和它异或产生的值最大的数字,即使当前数字是和后面的某个数字异或值最大,遍历后面那个数字的时候就会将这个数字选择出来

```

const int N=1e5+10,M=N*31;
int tr[M][2], idx;
int n, a[N];
void insert(int x) {
    int p=0;
    for(int i=31;i>=0;i--) {
        int v=x>>i&1;
        if(!tr[p][v]) tr[p][v]=++idx;
        p=tr[p][v];
    }
}
int query(int x) {
    int ans=0;
    int p=0;
    for(int i=31;i>=0;i--) {
        int v=x>>i&1;
        if(tr[p][v^1]){
            p=tr[p][v^1];
            ans=ans*2+v^1;
        }else{

```

```

        p=tr[p][v];
        ans=ans*2+v;
    }
}

return ans;
}

void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) cin>>a[i];
    int ans=0;
    //暴力是n^2, 用trie则是31*n, 边插入边查找
    //每一次找到跟a[i]异或最大的值
    for(int i=1;i<=n;i++) {
        insert(a[i]);
        int t=query(a[i]);
        ans=max(ans, t^a[i]);
    }
    cout<<ans<<'\n';
}

```

并查集

(1) 素朴并查集:

```

int p[N]; //存储每个点的祖宗节点
// 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(绑定到根节点上):

```

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:

```

```

size[find(b)] += size[find(a)];
p[find(a)] = find(b);

```

(3) 维护到祖宗节点距离的并查集（绑定到每个元素上）：

```

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

```

银河英雄传说

```

using namespace std;
typedef pair<int, int> PII;
const int N=3e4+10;
int p[N], sz[N], d[N];
int m;
int find(int x) {
    if(x!=p[x]) {
        int root=find(p[x]);
        d[x]+=d[p[x]];
        p[x]=root;
    }
    return p[x];
}
void solve() {
    cin>>m;
    for(int i=1;i<N;i++) {
        p[i]=i;
        sz[i]=1;
    }
}

```

```

    }

    for(int i=1; i<=m; i++) {
        char op;
        int a, b;
        cin>>op>>a>>b;
        if(op=='M') {
            a=find(a), b=find(b);
            if(a!=b) {
                d[a]=sz[b];
                sz[b]+=sz[a];
                p[a]=b;
            }
        } else{
            int pa=find(a), pb=find(b);
            if(pa!=pb) {
                cout<<-1<<'\n';
            } else{
                cout<<max(0, abs(d[a]-d[b])-1)<<'\n';
            }
        }
    }
}
}

```

H5 离散化+带边权或扩展域

奇偶游戏

带边权

如果我们用sum数组表示序列S的前缀和, 那么在每个回答中:

1. S[L~R] 有偶数个1, 等价于 sum[L-1]与sum[R] 奇偶性相同。
2. S[L~R] 有奇数个1, 等价于 sum[L-1]与sum[R] 奇偶性不同。

注意, 这里没有求出sum数组, 我们只是把sum看作变量。 此时本题与"程序自动分析"一题非常类似---> 都是给定若干变量和关系, 判定这些关系可满足性的问题。不同点是本题的传递关系不止一种:

1. 若x₁与x₂奇偶性相同, x₂与x₃奇偶性也相同, 那么x₁与x₃奇偶性相同。这种情况与"程序自动分析"中的等于关系一样。
2. 若x₁与x₂奇偶性相同, x₂与x₃奇偶性不同, 那么x₁与x₃奇偶性不同。
3. 若x₁与x₂奇偶性不同, x₂与x₃奇偶性也不同, 那么x₁与x₃奇偶性相同。

另外, 序列长度N很大, 但问题数M较少, 可以先用**离散化**把每个问题的**两个整数L-1和r缩小到等价的1~2M以内的范围。**

```

1 struct {
2     int l, r;
3     int ans;
4 } query[10010];
5
6 int a[20010], fa[20010], d[20010], n, m, t;
7
8 void read_discrete() {
9     cin >> n >> m;
10    for (int i = 1; i <= m; i++) {
11        char str[5];
12        cin>>query[i].l>>query[i].r,str;
13        query[i].ans = str[0] == 'o' ? 1 : 0;
14        a[++t] = query[i].l - 1;
15        a[++t] = query[i].r;
16    }
17    sort(a + 1, a + n + 1);
18    n = unique(a, a + 1, a + t + 1) - a - 1;
19 }

```

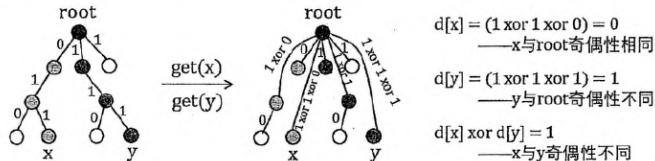
为了处理本题多种传递关系, 第一种解决方案是使用“**边带权**”的并查集。

边权 $d[x]$ 为0, 表示x与fa[x]奇偶性相同; 为1, 表示x与fa[x]奇偶性不同。路径压缩时,

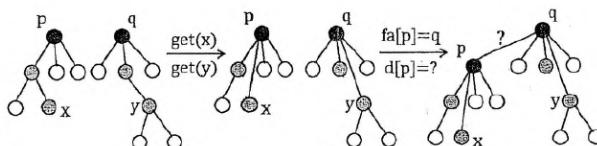
对x到树根路径上的所有边权做异或(xor)运算, 即可得到x与树根的奇偶性关系。

对于每个问题, 设在离散化后l-1和r的值分别是x和y, 设ans表示该问题的回答(0代表偶数个, 1代表奇数个)。

先检查x和y是否在同一个集合内(奇偶关系是否已知)。get(x)、get(y)都执行完成后,
 $d[x] \text{ xor } d[y]$ 即为x和y的奇偶性关系(如下图)。若 $d[x] \text{ xor } d[y] \neq \text{ans}$ (该关系与回答矛盾), 则在该问题之后即可确定小A撒谎。



若x和y不在同一集合内, 则合并两个集合。此时应该先通过get操作得到两个集合的树根(设为p和q), 令p为q的子节点。如下图所示。已知 $d[x]$ 与 $d[y]$ 分别表示路径 $x \sim p$ 与 $y \sim q$ 之间所有边权的“xor和”, $p \sim q$ 之间的边权 $d[p]$ 是待求得值。显然, 路径 $x \sim y$ 由路径 $x \sim p, p \sim q$ 与 $q \sim y$ 组成, 因此x与y得奇偶性关系 $\text{ans} = d[x] \text{ xor } d[y] \text{ xor } d[p]$ 。进行推出新连接得边权 $d[p] = d[x] \text{ xor } d[y] \text{ xor } \text{ans}$ 。

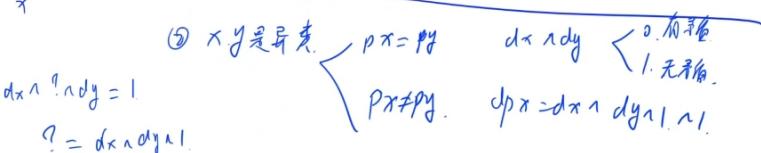
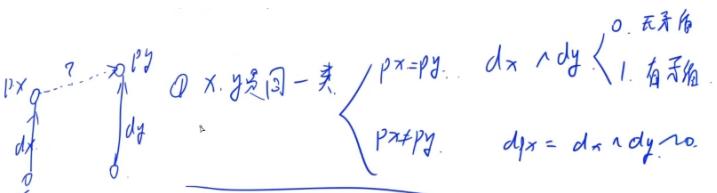
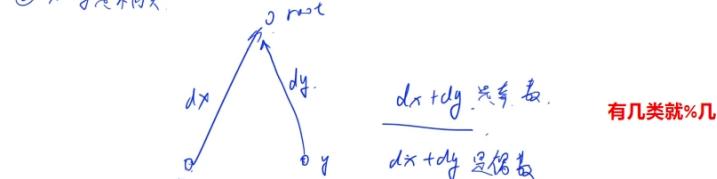


④ 带边权的并查集怎么做?

① x,y是同一类

“相对”, $d[x]$ 表示 x 与 p,q 的关系。 $\begin{cases} 0 \text{ 表示同类} \\ 1 \text{ 表示不同类} \end{cases}$

② x,y是不同类



```
using namespace std;
typedef pair<int, int> PII;
const int N=1e5+10;
int p[N];
int n, m;
int d[N];
map<int, int> mp;
int get(int x) {
    if(!mp.count(x)) mp[x]=++n;
    return mp[x];
}
```

```

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

void solve() {
    cin>>n>>m;
    n=0;
    for(int i=1;i<N;i++) {
        p[i]=i;
    }
    for(int i=1;i<=m;i++) {
        int a,b;
        string s;
        cin>>a>>b>>s;
        int t=0;
        if(s=="odd") t=1;
        a=get(a-1), b=get(b);
        int pa=find(a), pb=find(b);
        if(pa==pb) {
            if(d[a]^d[b]!=t) {
                cout<<i-1<<' \n';
                return ;
            }
        } else{
            p[pa]=pb;
            d[pa]=d[a]^d[b]^t;
        }
    }
    cout<<m<<' \n';
}

```

扩展域

第二种解决方案是使用“**扩展域**”的并查集。

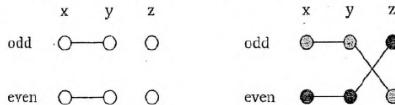
把每个变量x拆成两个节点 x_{odd} 和 x_{even} , 其中 x_{odd} 表示 $\text{sum}[x]$ 是奇数, x_{even} 表示 $\text{sum}[x]$ 是偶数。我们也经常把这两个节点称为x的“**奇数域**”和“**偶数域**”。

对于每个问题, 设在离散化后L-1和R的值分别是x和y, 设ans表示该问题的回答(0代表偶数个, 1代表奇数个)。

1. 若 $\text{ans}=0$, 则合并 x_{odd} 与 y_{odd} , x_{even} 与 y_{even} 。这表示“x为奇数”与“y为奇数”可以互相推出, “x为偶数”与“y为偶数”也可以互相推出, 他们是等价的信息。

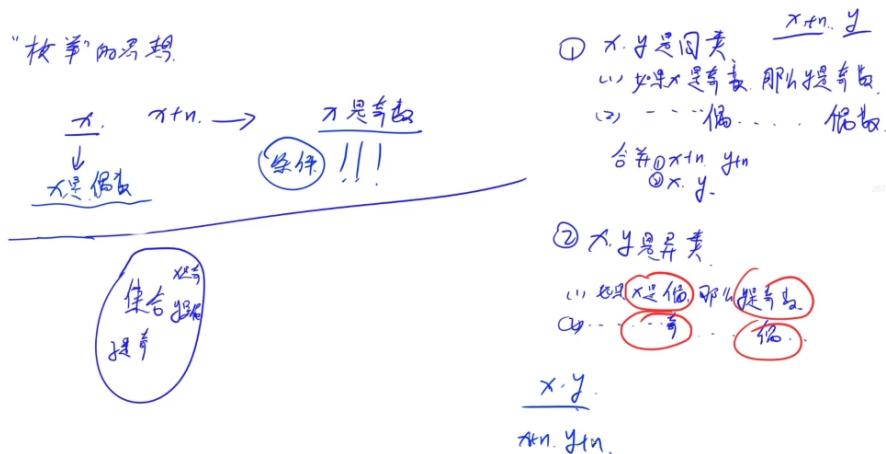
2. 若 $\text{ans}=1$, 则合并 x_{odd} 与 y_{even} , x_{even} 与 y_{odd} 。这表示“x为奇数”与“y为偶数”可以互相推出, “x为偶数”与“y为奇数”可以互相推出, 他们是等价的信息。

上述合并同时**维护了关系的传递性**。试想, 在处理完 $(x,y,0)$ 和 $(y,z,1)$ 两个回答之后, x和z之间的关系也就已知了, 如下图所示。这种做法就相当于在无向图上**维护节点之间的连通情况**, 只是扩展了多个域来应对多种传递关系。



在处理每个问题之前, 我们当然也要**检查是否存在矛盾**。若两个变量x和y对应的 x_{odd} 和 y_{odd} 节点在同一集合内, 则已知二者奇偶性相同。若两个变量x和y对应的 x_{odd} 和 y_{even} 节点在同一集合内, 则已知二者奇偶性不同。

扩展域



```
//x, y表示偶数, x+n, y+n表示奇数, 代表某一类, 而不是某一个元素
```

```
using namespace std;
typedef pair<int, int> PII;
const int N=1e5+10;
int p[N];
int n, m;
int d[N];
map<int, int> mp;
//离散化
int get(int x) {
    if(!mp.count(x)) mp[x]=++n;
    return mp[x];
}
int find(int x) {
    if(x!=p[x]) p[x]=find(p[x]);
    return p[x];
}
void solve() {
```

```

cin>>n>>m;
int x=5000;
n=0;
for(int i=1;i<N;i++) {
    p[i]=i;
}
for(int i=1;i<=m;i++) {
    int a,b;
    string s;
    cin>>a>>b>>s;
    a=get(a-1), b=get(b);
    if(s=="even") {
        //偶数奇偶性不同则矛盾
        if(find(a+x)==find(b)) {
            cout<<i-1<<' \n' ;
            return ;
        }else{
            p[find(a)]=find(b) ;
            p[find(a+x)]=find(b+x) ;
        }
    }else{
        //奇数奇偶性相同则矛盾
        if(find(a)==find(b)) {
            cout<<i-1<<' \n' ;
            return ;
        }else{
            p[find(a+x)]=find(b) ;
            p[find(a)]=find(b+x) ;
        }
    }
}
cout<<m<<' \n' ;
}

```

树状数组

支持操作: 区间查询 (求前缀和) `logn`, 单点修改(`logn`)

树状数组：

将X按二进制表示

$$x = 2^k + 2^{k-1} + \dots + 2^2 + 2^1 + 2^0$$

将X化成以下区间：

$(x-2^0, x]$	区间长度: 2^0	区间长度就是每个二进制位的值
$(x-2^1-2^0, x-2^0]$	2^1	对应二进制位的值
$(x-2^2-2^1-2^0, x-2^1-2^0]$	2^2	区间前缀和表示
\dots	\dots	\dots
$(x-2^k-2^{k-1}-\dots-2^1, x-2^1-\dots-2^0]$	2^{k-1}	$(R - \text{lowbit}(R) + 1, R]$
$(0, x-2^0-2^1-\dots-2^k]$	2^k	非对二进制前缀和的值

∴ 可以看每个 C_i

∴ 用 C_i 表示以为节点、长度为 $\text{lowbit}(k)$ 的

举例说明： $x=15 = 1111$

将15划成下列区间：

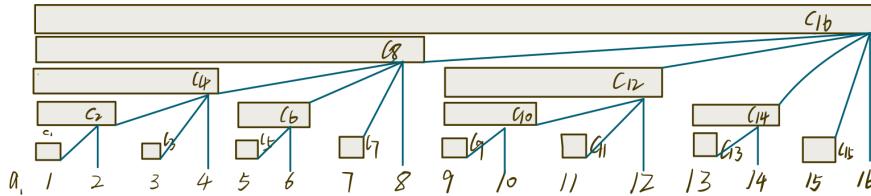
$$(15-2^0, 15] = C_{15}$$

$$(14-2^1, 14] = C_{14}$$

$$(12-2^2, 12] = C_{12}$$

$$(8-2^3, 8] = C_8$$

$$\text{若要来 } \text{sum}[1-15] = C_{15} + C_{14} + C_{12} + C_8$$



- 由节点查子节点的值 (计算 $C_{l, r}$)

$$C_{l, r} = a[x] + C_{x-1} + C_{x-1 - \text{lowbit}(x)} + C_{x+1 - \text{lowbit}(x) - \text{lowbit}(x+1)} \dots$$

- 由子节点查父节点

$$P = x + \text{lowbit}(x)$$

// 区间查询

```

int sum(int x) {
    int res=0;
    for(int i=x; i;i-=lowbit(i)) {
        res+=tr[i];
    }
    return res;
}

// 单点修改
void add(int x, int c) {
    for(int i=x; i;i+=lowbit(i)) {
        tr[i]+=c;
    }
}

// 也可以用差分数组创建树状数组，这样可以达到区间修改的操作

```

树状数组简洁精悍，考察问题的难点在于想到用树状数组去解题而不是实现代码上。

楼兰图腾

```

const int N = 2000010;
typedef long long LL;
int n;
//t[i]表示树状数组i结点覆盖的范围和
int a[N], t[N];
//Lower[i]表示左边比第i个位置小的数的个数
//Greater[i]表示左边比第i个位置大的数的个数
int Lower[N], Greater[N];
//返回非负整数x在二进制表示下最低位1及其后面的0构成的数值
int lowbit(int x)
{
    return x & -x;
}

```

```

    }

    //将序列中第x个数加上k。
    void add(int x, int k)
    {
        for(int i = x; i <= n; i += lowbit(i)) t[i] += k;
    }

    //查询序列前x个数的和
    int ask(int x)
    {
        int sum = 0;
        for(int i = x; i; i -= lowbit(i)) sum += t[i];
        return sum;
    }

    int main()
    {
        scanf("%d", &n);
        for(int i = 1; i <= n; i++) scanf("%d", &a[i]);
        //从左向右，依次统计每个位置左边比第i个数y小的数的个数、以及大的数的个数
        for(int i = 1; i <= n; i++)
        {
            int y = a[i]; //第i个数
            //在前面已加入树状数组的所有数中统计在区间[1, y - 1]的数字的出现次数
            Lower[i] = ask(y - 1);
            //在前面已加入树状数组的所有数中统计在区间[y + 1, n]的数字的出现次数
            Greater[i] = ask(n) - ask(y);
            //将y加入树状数组，即数字y出现1次
            add(y, 1);
        }
        //清空树状数组，从右往左统计每个位置右边比第i个数y小的数的个数、以及大的数的个数
        memset(t, 0, sizeof t);
        LL resA = 0, resV = 0;
        //从右往左统计
        for(int i = n; i >= 1; i--)
        {
            int y = a[i];
            resA += (LL)Lower[i] * ask(y - 1);
            resV += (LL)Greater[i] * (ask(n) - ask(y));
            //将y加入树状数组，即数字y出现1次
            add(y, 1);
        }
        printf("%lld %lld\n", resV, resA);
        return 0;
    }
}

```

H5 区间修改+单点查询（差分）

用b数组维护a数组的差分

```

using namespace std;
typedef pair<int, int> PII;
const int N=1e5+10;
int a[N], tr[N];

```

```
int n, m;
int lowbit(int x) {
    return x& -x;
}
void add(int x, int c) {
    for(int i=x; i<=n; i+=lowbit(i)) tr[i] += c;
}
int sum(int x) {
    int ans = 0;
    for(int i=x; i<=n; i-=lowbit(i)) ans += tr[i];
    return ans;
}
void solve() {
    cin >> n >> m;
    for(int i=1; i<=n; i++) cin >> a[i];
    for(int i=1; i<=n; i++) add(i, a[i] - a[i-1]); // 差分
    while(m--) {
        char op;
        int l, r, d;
        cin >> op >> l;
        if(op == 'C') {
            cin >> r >> d;
            // 差分
            add(l, d), add(r+1, -d);
        } else {
            // 单点查询即前缀和
            cout << sum(l) << '\n';
        }
    }
}
```

H5 区间修改+区间求和

用树状数组 $tr[]$ 来维护差分数组，
则求原数组的前缀和 $S[i]$ ，进行如下推导：

$$\alpha_1 = d_1$$

$$\alpha_2 = d_1 + d_2$$

$$\alpha_3 = d_1 + d_2 + d_3$$

$$\dots \quad \alpha_n = d_1 + d_2 + d_3 + \dots + d_n$$

$$S_i = \alpha_1 + \alpha_2 + \dots + \alpha_n = d_1 + d_2 + \\ d_1 + d_2 + d_3 + \dots + d_n$$

考虑把矩阵补全：

d_1	d_2	d_3	\dots	d_n
d_1	d_2	d_3	\dots	d_n
d_1	d_2	d_3	\dots	d_n
d_1	d_2	d_3	\dots	d_n
d_1	d_2	d_3	\dots	d_n
d_1	d_2	d_3	\dots	d_n
d_1	d_2	d_3	\dots	d_n

$$\text{则 } S_n = (d_1 + d_2 + \dots + d_n) \times (n+1) - (1 \cdot d_1 + 2 \cdot d_2 + \dots + n \cdot d_n)$$

$$= \sum_{i=1}^n d_i \times (n+1) - \sum_{i=1}^n i \cdot d_i$$

所以要在 $O(\log n)$ 的时间复杂度内对差分树状数组进行区间查询，则需要额外维护一个 $i \cdot d[i]$ 的差分树状数组

因此只需维护两个树状数组即可

一个是差分数组 $d[i]$ 的树状数组 $tr1[i]$ ，还有一个是 $i \cdot d[i]$ 的树状数组 $tr2[i]$

```
typedef pair<int, int> PII;
const int N=1e5+10;
int n, m, a[N], tr1[N], tr2[N];
int lowbit(int x) {
    return x&~x;
}
void add(int tr[], int x, int c) {
    for(int i=x; i<=n; i+=lowbit(i)) {
        tr[i] += c;
    }
}
```

```

int sum(int tr[], int x) {
    int ans=0;
    for(int i=x;i;i-=lowbit(i)) {
        ans+=tr[i];
    }
    return ans;
}
int get(int x) {
    return sum(tr1, x)*(x+1)-sum(tr2, x);
}
void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>a[i];
    for(int i=1;i<=n;i++) {
        int b=a[i]-a[i-1];
        add(tr1, i, b), add(tr2, i, b*i);
    }
    while(m--) {
        char op;
        int l, r, d;
        cin>>op>>l>>r;
        if(op=='Q') {
            cout<<get(r)-get(l-1)<<'\n';
        } else {
            cin>>d;
            add(tr1, l, d), add(tr1, r+1, -d);
            add(tr2, l, d*l), add(tr2, r+1, -d*(r+1));
        }
    }
}

```

H5 利用与前面元素的大小关系求解当前元素的具体值

谜一样的牛

题意：给定n头牛， $h[i]$ 表示第*i*头牛前面有 $h[i]$ 头牛比它低，求每头牛的身高。



```

//例如：对于这样的排列：
//1 2 3 4 5    (1 ~ 5之间的排列，表示五头牛可能的高度)
//那么构造一个数组表示这些数是否被选上，1表示还没选过，0表示已经被选过了，那么：
//1 1 1 1 1，这个就是要维护的数组b，并且它的前缀和表示当前位置是第几大的数，

//那么例如找一个数i它的前面由两个数比他小，那么对应的就要找第a[i] + 1大且没被用过的数，那么只要计算数组b的前缀和就能找到恰好是第a[i] + 1大的位置，用二分来进行前缀和位置的枚举即可

```

```

typedef pair<int, int> PII;
const int N=1e5+10;
int tr[N], a[N], n;
int lowbit(int x) {
    return x&-x;
}

```

```

void add(int x, int c) {
    for(int i=x; i<=n; i+=lowbit(i)) {
        tr[i] += c;
    }
}
int sum(int x) {
    int ans=0;
    for(int i=x; i; i-=lowbit(i)) ans+=tr[i];
    return ans;
}
void solve() {
    cin>>n;
    for(int i=2; i<=n; i++) {
        cin>>a[i];
    }
    for(int i=1; i<=n; i++) add(i, 1);
    vector<int> ans;
    //从最后一头牛开始
    for(int i=n; i; i--) {
        int k=a[i]+1;
        int l=1, r=n;
        while(l<=r) {
            int mid=l+r>>1;
            if(sum(mid)>=k) r=mid-1;
            else l=mid+1;
        }
        ans.push_back(r+1);
        add(r+1, -1);
    }
    reverse(ans.begin(), ans.end());
    for(int i=0; i<ans.size(); i++) {
        cout<<ans[i]<<' \n';
    }
}

```

线段树



```

struct Node {
    int l, r;
    LL sum; //这里可以维护任何满足区间加法的信息，这里就用区间求和了
} tr[4 * N]; //要开四倍空间
void push_up (int u) { //这里只有区间和，区间和就是由一个点的左右子节点的和相加
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}
void build (int u, int l, int r) { //当前正在下标为u的点，这个点表示的区间是[l, r]
    if (l == r) {
        tr[u] = {l, r, a[l]};
        return ;
    }
    int mid = (l + r) / 2;
    build(u << 1, l, mid);
    build(u << 1 | 1, mid + 1, r);
    tr[u].sum = tr[u << 1].sum + tr[u << 1 | 1].sum;
}

```

```

    }

    tr[u] = {l, r}; //记得存储当前点表示的区间，否则你会调上一整天
    int mid = l + r >> 1;
    build (u << 1, l, mid), build (u << 1 | 1, mid + 1, r); //u << 1就是u * 2, u << 1 | 1就是u * 2
+ 1
    push_up (u); //push_up函数的意思是把某一个节点的信息有他的子节点算出来
}

```



```

void modify (int u, int x, int d) { //当前这个点是下标为u的点，要把第x个数修改成d
    if (tr[u].l == tr[u].r) {
        tr[u].sum = d; //如果当前区间只有一个数，那么这个数一定是要修改的。
        return ;
    }
    int mid = tr[u].l + tr[u].r >> 1;
    if (x <= mid) modify (u << 1, x, d); //如果在左边就递归修改左半边区间
    else modify (u << 1 | 1, x, d); //如果在右边就递归修改右半边区间
    push_up (u) //记得更新信息
}

```



```

void modify (int u, int l, int r, int d) { //当前遍历到的点下标是u，要将区间[l, r]增加d
    if (tr[u].l == tr[u].r) { //叶子节点
        tr[u].sum += d;
        return ;
    }
    int mid = tr[u].l + tr[u].r >> 1; //注意是tr[u].l和tr[u].r
    if (l <= mid) modify (u << 1, l, r, d); //左边有修改区间，就递归左半边
    if (r >= mid + 1) modify (u << 1 | 1, l, r, d); //右边有修改区间，就递归右半边
    push_up (u); //要记得把这个点的信息更新一下
}

```



```

void push_down (int u) { //下传标记函数
    auto &root = tr[u], &left = tr[u << 1], &right = tr[u << 1 | 1]; //加了引用符号只要修改这个
变量就会修改他被赋值的变量
    if (root.sum_tag) { //有懒标记才下传
        left.tag += root.tag, left.sum += (LL)(left.r - left.l + 1) * root.tag; //这里的子节
点要加上懒标记，因为这里懒标记的意思是不包括当前节点
        right.tag += root.tag, right.sum += (LL)(right.r - right.l + 1) * root.tag; //同理
        root.tag = 0; //懒标记记得清零
    }
}

void modify (int u, int l, int r, int d) { //当前遍历到的点下标是u，要将区间[l, r]增加d
    if (l <= tr[u].l && tr[u].r <= r) {
        tr[u].sum += (LL)(tr[u].r - tr[u].l + 1) * d;
        tr[u].sum_tag += d; //这里我才用了总的懒标记的意思，一个点所有的子节点都加上d，而前
一行时加上根节点的数，因为不包括根节点。
        return ;
    }
}

```

```

    }

    push_down (u); //一定要分裂，只要记牢在递归左右区间之前，就要分裂
    int mid = tr[u].l + tr[u].r >> 1; //注意时tr[u].l和tr[u].r
    if (l <= mid) modify (u << 1, l, r, d); //左边有修改区间，就递归左半边
    if (r >= mid + 1) modify (u << 1 | 1, l, r, d); //右边有修改区间，就递归右半边
    push_up (u); //要记得要把这个点的信息更新一下
}

```



```

LL query_sum (int u, int l, int r) {
    if (l <= tr[u].l && tr[u].r <= r) return tr[u].sum;
    push_down (u); //在递归之前一定要分裂
    int mid = tr[u].l + tr[u].r >> 1;
    LL sum = 0;
    if (l <= mid) sum += query_sum (u << 1, l, r); //左半边有被查询到的数据，就递归左半边
    if (r >= mid + 1) sum += query_sum (u << 1 | 1, l, r); //右半边有被查询到的数据，就递归右半边
    return sum;
}

```

H5 单点修改+区间最值



```

const int N = 2e5 + 5;
typedef long long LL;
//线段树的结点，最大空间开4倍
struct Node{
    int l, r;
    int v; //最大值
} tr[N * 4];
int m, p;
//u为当前线段树的结点编号
void build(int u, int l, int r) {
    tr[u] = {l, r};
    if(l == r) return;
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
}

//查询以u为根节点，区间[l, r]中的最大值
int query(int u, int l, int r) {
    //      T1----Tr
    //      L-----R
    //1. 不必分治，直接返回
    if(tr[u].l >= l && tr[u].r <= r) return tr[u].v;

    int mid = tr[u].l + tr[u].r >> 1;
    int v = 0;
    //      T1----m---Tr
}

```

```

//          L-----R
//2. 需要在tr的左区间[Tl, m]继续分治
if(l <= mid) v = query(u << 1, l, r);

//      Tl----m----Tr
//      L-----R
//3. 需要在tr的右区间(m, Tr]继续分治
if(r > mid) v = max(v, query(u << 1 | 1, l, r));

//      Tl----m----Tr
//      L-----R
//2.3涵盖了这种情况
return v;
}

//u为结点编号，更新该结点的区间最大值
void modify(int u, int x, int v) {
    if(tr[u].l == tr[u].r) tr[u].v = v; //叶节点，递归出口
    else {
        int mid = tr[u].l + tr[u].r >> 1;
        //分治处理左右子树，寻找x所在的子树
        if(x <= mid) modify(u << 1, x, v);
        else modify(u << 1 | 1, x, v);
        //回溯，拿子结点的信息更新父节点，即pushup操作
        tr[u].v = max(tr[u << 1].v, tr[u << 1 | 1].v);
    }
}
int main()
{
    //n表示树中的结点个数，last保存上一次查询的结果
    int n = 0, last = 0;
    cin >> m >> p;
    //初始化线段树，结点的区间最多为[1, m]。
    build(1, 1, m);
    while(m--)
    {
        char op;
        cin >> op;
        if(op == 'A') //添加结点
        {
            int t;
            cin >> t;
            //在n + 1处插入
            modify(1, n + 1, ((LL)t + last) % p);
            //结点个数+1
            n++;
        }
        else
        {
            int L;
            cin >> L;
            //查询[n - L + 1, n]内的最大值，u = 1，即从根节点开始查询
            last = query(1, n - L + 1, n);
        }
    }
}

```

```

        cout << last << endl;
    }
}

return 0;
}

```

h5 单点修改+最大连续子段和



```

const int N=5e5+10;
struct Node{
    int l,r;
    int tmax, lmax, rmax, sum;
} tr[N*4];
int n, m;
int w[N];
void pushup(Node &root, Node &l1, Node &r1) {
    root.sum=l1.sum+r1.sum;
    root.tmax=max(max(l1.tmax, r1.tmax), l1.rmax+r1.lmax);
    root.lmax=max(l1.lmax, l1.sum+r1.lmax);
    root.rmax=max(r1.rmax, l1.rmax+r1.sum);
}
void pushup(int u){
    pushup(tr[u], tr[u<<1], tr[u<<1|1]);
}
void build(int u, int l, int r){
    if(l==r) tr[u]={l, r, w[l], w[l], w[l]};
    else{
        tr[u]={l, r};
        int mid=l+r>>1;
        build(u<<1, l, mid);
        build(u<<1|1, mid+1, r);
        pushup(u);
    }
}
void modify(int u, int x, int v){
    if(tr[u].l==x && tr[u].r==x){
        tr[u].tmax=tr[u].lmax=tr[u].rmax=tr[u].sum=v;
        return ;
    }
    int mid=tr[u].l+tr[u].r>>1;
    if(x<=mid){
        modify(u<<1, x, v);
    }else{
        modify(u<<1|1, x, v);
    }
    pushup(u);
}

```

```

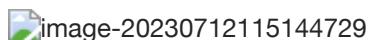
    }

Node query(int u, int l, int r) {
    if(tr[u].l>=l && tr[u].r<=r) return tr[u];
    int mid=tr[u].l+tr[u].r>>1;
    if(r<=mid) {
        return query(u<<1, l, r);
    } else if(l>mid) {
        return query(u<<1|1, l, r);
    } else{
        auto ll=query(u<<1, l, r);
        auto rr=query(u<<1|1, l, r);
        Node res;
        pushup(res, ll, rr);
        return res;
    }
}

void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>w[i];
    build(1, 1, n);
    while(m--) {
        int k, x, y;
        cin>>k>>x>>y;
        if(k==1) {
            if(x>y) swap(x, y);
            cout<<query(1, x, y).tmax<<'\n';
        } else{
            modify(1, x, y);
        }
    }
}

```

h5 区间修改+区间最大gcd



```

const int N=5e5+10;
struct Node{
    int l, r;
    int sum, d;
} tr[N*4];
int w[N];
int n, m;
void pushup(Node &x, Node &y, Node &z) {
    x.sum=y.sum+z.sum;
    x.d=__gcd(y.d, z.d);
}
void pushup(int u) {
    pushup(tr[u], tr[u<<1], tr[u<<1|1]);
}

```

```

void build(int u, int l, int r) {
    if(l==r) {
        int b=w[l]-w[l-1];//差分
        tr[u]={l, r, b, b};
        return ;
    }
    tr[u]={l, r};
    int mid=l+r>>1;
    build(u<<1, l, mid), build(u<<1|1, mid+1, r);
    pushup(u);
}

Node query(int u, int l, int r) {
    if(tr[u].l>=l && tr[u].r<=r) {
        return tr[u];
    }
    int mid=tr[u].l+tr[u].r>>1;
    if(r<=mid) {
        return query(u<<1, l, r);
    }else if(l>mid) {
        return query(u<<1|1, l, r);
    }else{
        auto tl=query(u<<1, l, r);
        auto tr=query(u<<1|1, l, r);
        Node res;
        pushup(res, tl, tr);
        return res;
    }
}
void modify(int u, int x, int v) {
    if(tr[u].l==x && tr[u].r==x) {
        int b=tr[u].sum+v;
        tr[u]={x, x, b, b};
        return ;
    }
    int mid=tr[u].l+tr[u].r>>1;
    if(x<=mid) {
        modify(u<<1, x, v);
    }else{
        modify(u<<1|1, x, v);
    }
    pushup(u);
}
void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>w[i];
    build(1, 1, n);
    while(m--) {
        char op;
        int l,r;
        cin>>op>>l>>r;
        if(op=='Q') {
            auto tl=query(1, l, r);

```

```

        Node t={0, 0, 0, 0} ;
        if(l+1<=n) t=query(1, l+1, r) ;
        cout<<abs(__gcd(t1.sum, t.d))<< '\n' ;
    }else{
        int x;
        cin>>x;
        modify(1, l, x);
        if(r+1<=n) modify(1, r+1, -x);
    }
}
}

```

h5 区间修改+区间求和（懒标记）



```

const int N=1e5+10;
struct Node{
    int l,r;
    int sum,add;
}tr[N*4];
int w[N];
int n,m;
void pushup(int u){
    tr[u].sum=tr[u<<1].sum+tr[u<<1|1].sum;
}
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+=(left.r-left.l+1)*root.add;
        right.add+=root.add, right.sum+=(right.r-right.l+1)*root.add;
        root.add=0;
    }
}
void build(int u, int l, int r){
    if(l==r){
        tr[u]={l, r, w[l], 0} ;
    }else{
        tr[u]={l, r} ;
        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 d){
    if(tr[u].l>=l && tr[u].r<=r) {
        tr[u].add+=d;
        tr[u].sum+=(tr[u].r-tr[u].l+1)*d;
    }else{
        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);
    }

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

void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>w[i];
    build(1, 1, n);
    while(m--) {
        char op;
        int l, r;
        cin>>op>>l>>r;
        if(op=='C'){
            int d;cin>>d;
            modify(1, l, r, d);
        }else{
            cout<<query(1, l, r)<<' \n' ;
        }
    }
}

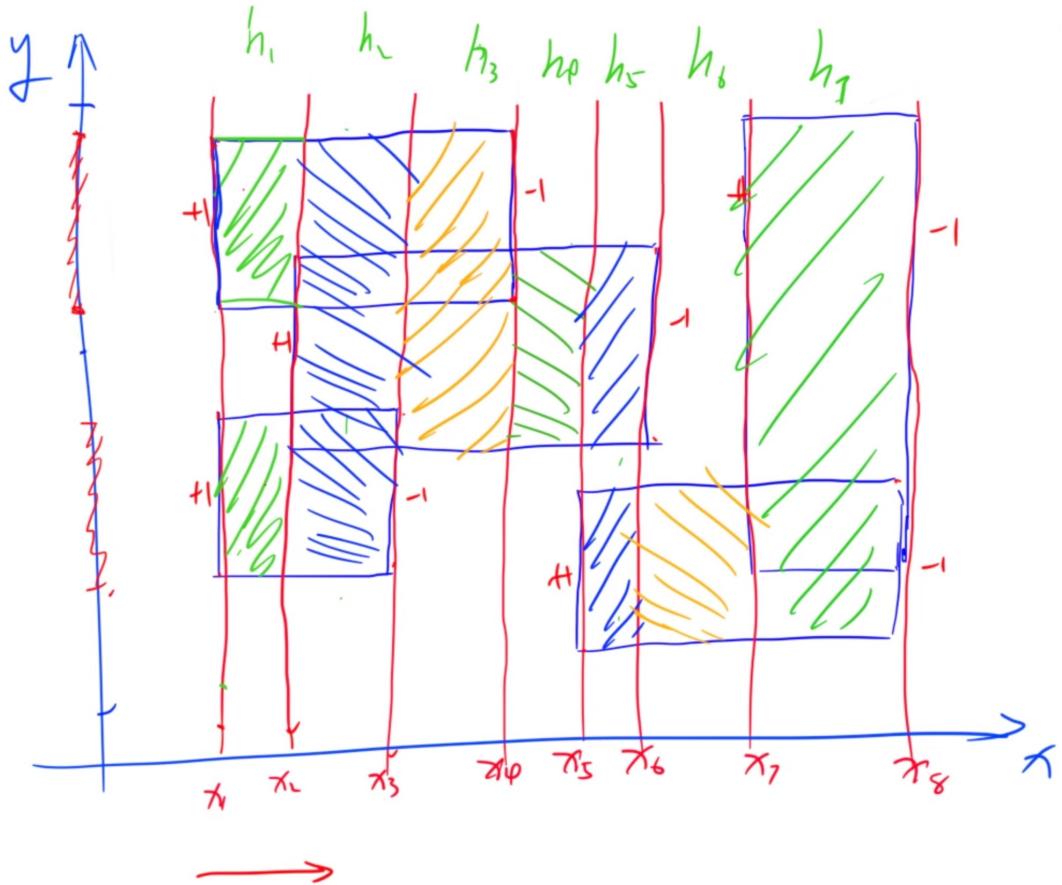
```

H5 扫描线

线段树：扫描线法

无法拓展到一般情况，扫描线法请自觉背出





```

const int N = 100010;
struct segment// 用来存线段信息
{
    double x, y1,y2;
    int d; // 区分它是该矩阵前面的线段还是后面的线段
    bool operator < (const segment&t) const
    {
        return x < t.x;
    }
} seg[N * 2];//每个矩阵需要存两个线段
// 线段树的每个节点 保存的为线段,0号点为y[0]到y[1], 以此类推
struct node
{
    int l,r;
    int cnt;// 记录这段区间出现了几次
    double len;// 记录这段区间的长度;即线段长度
} tr[N * 8];//由于线段二倍, 所以8倍空间
vector<double>ys;//用于离散化
int n;
int find(double y)
{
    // 需要返回vector 中第一个 >= y 的数的下标
    return lower_bound(ys.begin(), ys.end(), y) - ys.begin();
}
void pushup(int u)
{
    //例如: 假设tr[1].l = 0, tr[1].r = 1;
    //      y[0]为ys[0]到ys[1]的距离, y[1]为ys[1]到ys[2]的距离
}

```

```

//      tr[1].len 等于y[0]到y[1]的距离
//      y[1] = ys[tr[1].r + 1] = ys[2], y[0] = ys[tr[1].l] = ys[0]
if(tr[u].cnt) tr[u].len = ys[tr[u].r + 1] - ys[tr[u].l];//表示整个区间都被覆盖，该段长度就为右端点 + 1后在ys中的值 - 左端点在ys中的值

// 借鉴而来
// 如果tr[u].cnt等于0其实有两种情况：
// 1. 完全覆盖。这种情况由modify的第一个if进入。
//      这时下面其实等价于把“由完整的l, r段贡献给len的部分清除掉”，
//      而留下其他可能存在的子区间段对len的贡献
// 2. 不完全覆盖，这种情况由modify的else最后一行进入。
//      表示区间并不是完全被覆盖，可能有部分被覆盖，所以要通过儿子的信息来更新
else if(tr[u].l != tr[u].r)
{
    tr[u].len = tr[u << 1].len + tr[u << 1 | 1].len;
}
else tr[u].len = 0;//表示为叶子节点且该线段没被覆盖，为无用线段，长度变为0
}

void modify(int u, int l, int r, int d)//表示从线段树中l点到r点的出现次数 + d
{
    if(tr[u].l >= l && tr[u].r <= r)//该区间被完全覆盖
    {
        tr[u].cnt += d;//该区间出现次数 + d
        pushup(u);//更新该节点的len
    }
    else
    {
        int mid = tr[u].r + tr[u].l >> 1;
        if (l <= mid) modify(u << 1, l, r, d);//左边存在点
        if (r > mid) modify(u << 1 | 1, l, r, d);//右边存在点
        pushup(u);//进行更新
    }
}
}

void build(int u, int l, int r)
{
    tr[u] = {l, r, 0, 0};

    if (l != r)
    {
        int mid = l + r >> 1;
        build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
        //后面都为0， 不需更新len
    }
}
}

int main()
{
    int T = 1;
    while (cin >> n, n)//多组输入
    {
        ys.clear();//清空
        int j = 0;//一共j个线段
        for (int i = 0 ; i < n ; i ++)//处理输入
        {

```

```

        double x1, y1, x2, y2;
        cin>>x1>>y1>>x2>>y2;
        seg[j ++] = {x1, y1, y2, 1};//前面的线段
        seg[j ++] = {x2, y1, y2, -1};//后面的线段
        ys.push_back(y1), ys.push_back(y2);//y轴出现过那些点
    }

    sort(seg, seg + j); //线段按x排序
    sort(ys.begin(), ys.end()); //先排序
    ys.erase(unique(ys.begin(), ys.end()), ys.end()); //离散化去重
    //例子：假设现在有三个不同的y轴点，分为两个线段
    //y[0] ~ y[1], y[1] ~ y[2];
    //此时ys.size()为3,ys.size() - 2 为 1;
    //此时为 build(1, 0, 1);
    //有两个点0 和 1,线段树中0号点为y[0] ~ y[1],1号点为y[1] ~ y[2];
    build(1, 0, ys.size() - 2);
    double res = 0;
    for (int i = 0; i < j; i++)
    {
        //根节点的长度即为此时有效线段长度，再 * x轴长度即为面积
        if (i) res += tr[1].len * (seg[i].x - seg[i - 1].x);
        //处理一下该线段的信息，是加上该线段还是消去
        //例子：假设进行modify(1, find(10), find(15) - 1, 1);
        //      假设find(10) = 0,find(15) = 1;
        //      此时为modify(1, 0, 0, 1);
        //      表示线段树中0号点出现次数加1;
        //      而线段树中0号点刚好为线段(10 ~ 15);
        //      这就是为什么要进行find(seg[i].y2) - 1 的这个-1操作
        modify(1, find(seg[i].y1), find(seg[i].y2) - 1, seg[i].d);
    }

    printf("Test case #%d\n", T ++ );
    printf("Total explored area: %.2lf\n\n", res);
}
return 0;
}

```

H5 区间加和乘，区间求和

存储信息：

- 1.区间范围l,r
- 2.加的懒标记add
- 3.乘的懒标记mul
- 4.区间总和sum

对于x

若对懒标记的处理是先加再乘

若此次操作为乘上一个数c

可以表示为 $(n + add) * mul * c$ 即 $(n + X) * X$ 的形式

若此次操作为加上一个数c

$(n + add) * mul + c$ 不能写成 $(n + X) * X$ 的形式

\rightarrow 无法更新新的懒标记

若对懒标记的处理是先乘再加

若此次操作是加上一个数c

可以表示为 $n * \text{mul} + \text{add} + c$

→ 此时新的add即为 $\text{add} + c$

若此次操作是乘上一个数c

可以表示为 $n * \text{mul} * c + \text{add} * c$

→ 此时新的add即为 $\text{add} * c$, 新的mul即为 $\text{mul} * c$

→ 故先乘再加, 以便更新懒标记

可以把乘和加的操作都看成 $x * c + d$

→ 若是乘法, d为0

→ 若是加法, c为1

若当前x的懒标记为add和mul

→ 操作可以写成 $(x * \text{mul} + \text{add}) * c + d$

→ 即 $x * (\text{mul} * c) + (\text{add} * c + d)$

→ 新的mul为 $(\text{mul} * c)$, 新的add为 $(\text{add} * c + d)$

注意: 乘的懒标记初始为1

```
typedef long long LL;
const int N = 1e5 + 10;
int n, p, m;
int w[N];
struct Node{
    int l, r, sum, add, mul;
} tr[4 * N];
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)           //计算在该区间加或乘一个数, 数据可能会爆int
{
    t.sum = ((LL)t.sum * mul + (LL)(t.r - t.l + 1) * add) % p;
    t.mul = (LL)t.mul * mul % p;                      //根据推的公式
    t.add = ((LL)t.add * mul + add) % p;    //根据推的公式
}
void pushdown(int u)
{
    eval(tr[u << 1], tr[u].add, tr[u].mul);        //把当父区间的懒标记看作加或乘操作
    eval(tr[u << 1 | 1], tr[u].add, tr[u].mul); //tr[u].add即为在子区间加上add, mul同理
    tr[u].add = 0;                                //删去父区间懒标记
    tr[u].mul = 1;                                //删去父区间懒标记
}
void build(int u, int l, int r)
{
    tr[u].r = r, tr[u].l = l;
    if (l == r)
        tr[u].sum = w[l], tr[u].add = 0, tr[u].mul = 1;
    else
    {
```

```

        int mid = l + r >> 1;
        tr[u].add = 0, tr[u].mul = 1;
        build(u << 1, 1, mid);
        build(u << 1 | 1, mid + 1, r);
        pushup(u);           //回溯时通过子区间更新父区间信息
    }
}

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); //若当前访问区间在查询区间内
    else
    {
        pushdown(u);           //区间分裂时需先处理懒标记
        int mid = tr[u].r + tr[u].l >> 1;
        if (mid >= l) modify(u << 1, l, r, add, mul);           //递归处理左右子区间
        if (mid < r) 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;           //若当前访问区间在查询区间内
    else
    {
        pushdown(u);           //区间分裂时需先处理懒标记
        int mid = tr[u].l + tr[u].r >> 1;
        int sum = 0;
        if (mid >= l) sum = query(u << 1, l, r) % p;           //递归处理左右子区间
        if (mid < r) 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", &w[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, 1, r));
}
return 0;
}

```

平衡树---Treap



```

const int N = 100010, INF = 1e8;
struct Node { //平衡树节点
    int l, r; //左右儿子的下标
    int key, val; //key表示当前点的键值, val表示随机分配的用于更改平衡树形态的的数值
    int cnt, size; //cnt表示当前值有几个, size表示以当前点为根的子树一共有几个节点
} tr[N]; //平衡树开n个就够了, 因为一个点一个数组空间
int root, idx; //root表示根的下标, 就是1, 主要是为了方便修改, idx表示当前分配到第几个点, 类似于
Trie树
int get_node (int key) { //创建一个节点的键值是key, 返回值是当前点在数组中下标
    tr[+idx].key = key; //直接存入新的节点中
    tr[idx].val = rand (); //随机分配数值
    tr[idx].cnt = tr[idx].size = 1; //节点数只有当前点
    return idx; //返回当前点在数组中下标
}
void push_up () { //类似于线段树的push_up, 就是上传子树信息
    tr[p].size = tr[tr[p].l].size + tr[tr[p].r].size + tr[p].cnt;
}
void build_Treap () { //初始化一个平衡树
    get_node (-INF), get_node (INF); //插入-∞和+∞
    root = 1, tr[root].r = 2; //这里设-∞为根
    push_up (root);
    if (tr[root].val < tr[tr[root].r].val) left_rotate (root); //如果不满足堆性质, 就旋转, 后
面会讲
}

```



```

void right_rotate (int &p) { //注意这里是要改变数值的, 所以要加引用
    int q = tr[p].l;
    tr[p].l = tr[q].r, tr[q].r = p, p = q;
    push_up (tr[p].r).push_up (p); //要先push_up子节点, 在push_up父节点
}
void left_rotate (int &p) { //注意这里是要改变数值的, 所以要加引用
    int q = tr[p].r; //可以由right_rotate对应过来, l变成r, r变成l
    tr[p].r = tr[q].l, tr[q].l = p, p = q;
    push_up (tr[p].l).push_up (p); //要先push_up子节点, 在push_up父节点
}

```



```

void insert (int &p, int key) {
    if (!p) p = get_node (key); //这里就体现了引用的作用
    else if (tr[p].key == key) tr[p].cnt++; //如果已经存在这个数了，就cnt++;
    else if (tr[p].key > key) insert (tr[p].l, key); //如果当前点大了，就往左边插入
    else insert (tr[p].r, key); //否则就从右边插入
    push_up (p); //记得push_up一下
}

```



```

void erase (int &p, int key) { //同样道理，还是要加引用修改
    if (!p) return ; //没有这个点就直接结束
    if (tr[p].key == key) { //找到这个点了
        if (tr[p].cnt > 1) tr[p].cnt--; //当前点的个数大于1就cnt--
        else if (tr[p].l || tr[p].r) { //当前点不是叶子节点得进行旋转操作旋转到叶子节点上
            if (!tr[p].r || tr[tr[p].l].val > tr[tr[p].r].val) {
                //如果右边是空的或者右转后可以满足堆性质，就直接右旋
                right_rotate (p);
                erase (tr[p].r, key); //旋转后继续删除
            }
            else { //否则就左旋
                left_rotate (p);
                erase (tr[p].l, key);
            }
        }
        else p = 0; //是叶子节点就直接删除
    }
    else if (tr[p].key > key) erase (tr[p].l, key); //如果当前点大了就往左删除
    else erase (tr[p].r, key); //否则从右边删除
    push_up (p); //记得要push_up一下
}

```



```

void get_rank_by_key (int p, int key) {
    if (!p) return 0; //没有这个点就返回排名0
    if (tr[p].key == key) return tr[tr[p].l].size + 1; //返回最小排名
    if (tr[p].key > key) return get_rank_by_key (tr[p].l, key); //左边不用变
    return tr[tr[p].l].size + tr[p].cnt + get_rank_by_key (tr[p].r, key);
    //右边要加上当前点的数量和左子树的数量
}

```



```

int get_key_by_rank (int p, int rank) {
    if (!p) return INF; //没有当前点, 说明排名太大, 返回+∞
    if (tr[tr[p].l].size >= rank) return get_key_by_rank (tr[p].l, rank); //在左边就递归左子
树
    if (tr[tr[p].l].size + tr[p].cnt >= rank) return tr[p].key; //在当前点就直接返回
    return get_key_by_rank (tr[p].r, rank - tr[tr[p].l].size - tr[p].cnt);
    //在右边要剪掉当前点的数量和左子树的点数
}

```



```

int get_prev (int p, int key) {
    if (!p) return -INF; //没有前驱, 就返回-INF
    if (tr[p].key >= key) return get_prev (tr[p].l, key); //如果当前点和右边都不可能是答案, 就递
归左边
    return max (tr[p].key, get_prev (tr[p].r, key)); //否则从当前点的键值和右子树找出来的前驱取
max
}

```



```

int get_next (int p, int key) { //直接对应get_prev抄下来
    if (!p) return INF;
    if (tr[p].key <= key) return get_next (tr[p].r, key);
    return min (tr[p].key, get_next (tr[p].l, key));
}

```



```

const int N = 100010, INF = 1e8;
int n;
struct Node
{
    int l, r;
    int key, val;//节点键值 || 权值
    int cnt, size;//这个数出现次数 || 每个(节点)子树里数的个数
} tr[N];
int root, idx;
void pushup(int p)//儿子信息更新父节点信息
{
    tr[p].size = tr[tr[p].l].size + tr[tr[p].r].size + tr[p].cnt;
    //左儿子树 数个数+ 右儿子树 数个数+ 当前节点数出现次数(重复次数)
}
int get_node(int key)
{
    tr[++idx].key = key;
    tr[idx].val = rand();
    tr[idx].cnt = tr[idx].size = 1;//默认创建时都是叶子节点
    return idx;
}
/*          b挂x          交换pq

```

```

        x(p)           y(q)           y(p)
        / \ 右旋zig   / \           / \
y(q).c ->      a   x(p) ->    a   x(tr[p].r)
/ \           / \           / \
a   b         b   c         b   c

*/
void zig(int &p)//右旋 p指针指向的根节点
{
    int q = tr[p].l;//先把p的左儿子q存下来
    tr[p].l = tr[q].r;//b从q的右儿子变为p的左儿子
    tr[q].r = p;//最终q的右儿子是p指向的x
    p = q;//p从指向原来根节点x变为指向新根节点q
    pushup(tr[p].r), pushup(p); //先更新 右子树(x bc) 再更新根节点y
}

/*
          交换pq           b挂x
        y(p)           y(q)           x(p)
        / \           / \           / \
(tr[p].l)x  c       x(p)c       a   y(q)
/ \     <- / \     <- / \
a   b     a   b   左旋zag     b   c

*/
void zag(int &p)
{
    int q = tr[p].r;//先把p的右儿子y(q)存下来
    tr[p].r = tr[q].l;//b挂x(p)
    tr[q].l = p;//p挂y的左儿子
    p=q;//p维护指向根节点
    pushup(tr[p].l), pushup(p);
}

void build()
{
    get_node(-INF), get_node(INF);
    root = 1, tr[1].r = 2;//根节点1号点, 1号点的右儿子2号点
    //build的时候不要漏了维护树的结构
    pushup(root);
    if (tr[1].val < tr[2].val) zag(root);
}

void insert(int &p, int key)
{
    if(!p) p = get_node(key); //如果到达叶子节点fa 并进一步递归tr[fa].l or tr[fa].r == 空 新增一个key叶子节点
    else if(tr[p].key == key) tr[p].cnt++; //如果插入已有的值 cnt++
    else if(tr[p].key>key) //插入新的值 判断插入左子树还是右子树 直到叶子节点
    { //如果要插入的key比p的key小 递归tr[p]的左子树tr[p].l
        insert(tr[p].l, key);
        if(tr[tr[p].l].val > tr[p].val) zig(p); //如果左子树val大于root.val 就换上来
    }
    //维护val是根节点root.val > 子树 root.l.val or root.r.val
    else
    { //如果要插入的key比p的key大 递归tr[p]的右子树tr[p].r
        insert(tr[p].r, key);
        if(tr[tr[p].r].val > tr[p].val) zag(p); //如果右子树val大于root.val 就换上来
    }
}

```

```

    }
    pushup(p);
}

void remove(int &p, int key)
{
    if(!p) return;//如果删除的值不存在
    if(tr[p].key == key)//如果有值
    {
        if(tr[p].cnt>1) tr[p].cnt--;//如果cnt>1 cnt--
        else if(tr[p].l || tr[p].r)//否则判断是否叶子节点
        {//如果不是叶子节点
            if(!tr[p].r || tr[tr[p].l].val > tr[tr[p].r].val)//右旋把p放到右子树 递归到叶子
            节点删除
                {//如果没有右子树 肯定可以右旋 || 左子树的.val > 右子树的val
                zig(p);
                remove(tr[p].r, key);
            }
            else//否则左旋把p放到左子树 递归到叶子节点删除
            {
                zag(p);
                remove(tr[p].l, key);
            }
        }
        else p=0;//如果是叶子节点 直接删除
    }
    else if(tr[p].key > key) remove(tr[p].l, key);//如果 要删除的中序顺序key>当前节点p的tr[p].key
    去p左子树删
    else remove(tr[p].r, key);
    pushup(p);
}

int get_rank_by_key(int p, int key)//通过数值找排名 传p而不是传&p
{
    if(!p) return 0;//如果p不存在 返回0
    if(tr[p].key==key) return tr[tr[p].l].size+1;//如果key和当前节点key相等 排名=左子树数的大小
    +1(当前节点)
    if(tr[p].key>key) return get_rank_by_key(tr[p].l, key);//漏了return
    //说明要找的key比当前节点小 去当前节点左子树找
    return tr[tr[p].l].size + tr[p].cnt + get_rank_by_key(tr[p].r, key);
    //去右边找返回的是右边子树的排名, 所以还得加上根节点root数的个数以及根节点左子树总数
    root.l.size
}
int get_key_by_rank(int p, int rank)//通过排名找数值 传p而不是传&p
{
    if(!p) return INF;//p不存在 则返回不可能存在的排名
    if(tr[tr[p].l].size >=rank) return get_key_by_rank(tr[p].l, rank);//如果左子树数的个数>rank,
    则去左子树找
    if(tr[tr[p].l].size+tr[p].cnt>=rank) return tr[p].key;//左子树+当前节点个数>=rank, 则就是当前
    节点
    //左子树+当前节点个数<rank, 则递归右子树, 同时往下传的rank-左子树-当前节点个数
    return get_key_by_rank(tr[p].r, rank-tr[tr[p].l].size-tr[p].cnt);//-tr[p].cnt
}
int get_prev(int p, int key)//找到严格小于key的数中的最大数 传p而不是传&p

```

```

{
    if(!p) return -INF;
    if(tr[p].key>=key) return get_prev(tr[p].l, key); //当前这个数>=key 则要找的数在当前节点的左子
    //树里
    return max(tr[p].key, get_prev(tr[p].r, key));
    // if(tr[p].key<key) //当前节点<key max(当前节点(无右子树) || 当前节点的右子树中的最大值(有
    //右子树))
    // {
    //     if(tr[p].r) return get_prev(tr[p].r, key); //错了 有右子树 但右子树的key不一定比要找
    //的key小
    //     return tr[p].key;
    // }
}

int get_next(int p, int key)//找到严格大于key的数中的最小数 传p而不是传&p
{
    if (!p) return INF;
    if (tr[p].key <= key) return get_next(tr[p].r, key);
    return min(tr[p].key, get_next(tr[p].l, key));
}

int main()
{
    build();
    scanf("%d", &n);
    while(n--)
    {
        int op, x;
        scanf("%d%d", &op, &x);
        if(op==1) insert(root, x);
        else if (op == 2) remove(root, x);
        //有哨兵的影响，排名要-1, +1。
        else if (op == 3) printf("%d\n", get_rank_by_key(root, x) - 1);
        else if (op == 4) printf("%d\n", get_key_by_rank(root, x + 1));
        else if (op == 5) printf("%d\n", get_prev(root, x));
        else printf("%d\n", get_next(root, x));
    }
    return 0;
}
}

```



这道题就是把当前天之前的所有数都加到平衡树当中，然后每次累加与前驱和后继的差的最小差累加起来即可。

```

const int N = 40010, INF = 1e7;
int n;
struct treap_node {
    int l, r;
    int key, val;
    int size, cnt;
} tr[N];
int root, idx;
int new_node (int key) {

```

```

        tr[+idx].key = key, tr[idx].val = rand() , tr[idx].cnt = tr[idx].size = 1;
        return idx;
    }

    void push_up (int u) {
        tr[u].size = tr[tr[u].l].size + tr[tr[u].r].size + tr[u].cnt;
    }

    void left_rotate (int &u) {
        int p = tr[u].r;
        tr[u].r = tr[p].l, tr[p].l = u, u = p;
        push_up (tr[u].l), push_up (u);
    }

    void right_rotate (int &u) {
        int p = tr[u].l;
        tr[u].l = tr[p].r, tr[p].r = u, u = p;
        push_up (tr[u].r), push_up (u);
    }

    void init () {
        new_node (-INF), new_node (INF);
        root = 1, tr[root].r = 2;
        if (tr[root].val < tr[tr[root].r].val) left_rotate (root);
    }

    void insert (int &u, int key) {
        if (!u) u = new_node (key);
        else if (tr[u].key == key) tr[u].cnt++;
        else if (tr[u].key > key) {
            insert (tr[u].l, key);
            if (tr[tr[u].l].val > tr[u].val) right_rotate (u);
        }
        else {
            insert (tr[u].r, key);
            if (tr[tr[u].r].val > tr[u].val) left_rotate (u);
        }
        push_up (u);
    }

    int get_prev (int u, int key) {
        if (!u) return -INF;
        if (tr[u].key > key) return get_prev (tr[u].l, key);
        return max (tr[u].key, get_prev (tr[u].r, key));
    }

    int get_next (int u, int key) {
        if (!u) return INF;
        if (tr[u].key < key) return get_next (tr[u].r, key);
        return min (tr[u].key, get_next (tr[u].l, key));
    }

    int main () {
        init ();
        cin >> n;
        int ans;
        cin >> ans;
        insert (root, ans);
        for (int i = 2; i <= n; i++) {
            int x;

```

```

        cin >> x;
        ans += min (abs (x - get_prev (root, x)), abs (x - get_next (root, x)));
        insert (root, x);
    }
    cout << ans << endl;
    return 0;
}

```

AC自动机



```

int son[N][26], cnt[N], idx; //cnt数组记录以当前节点为结尾的字符串的数目，idx为节点编号
int n;

void insert(string& s) {
    int n = s.length();
    int p = 0;
    for(int i = 0; i<n; i++) {
        int u = s[i] - 'a';
        if(!son[p][u]) son[p][u] = ++idx;
        p = son[p][u];
    }
    cnt[p]++;
}

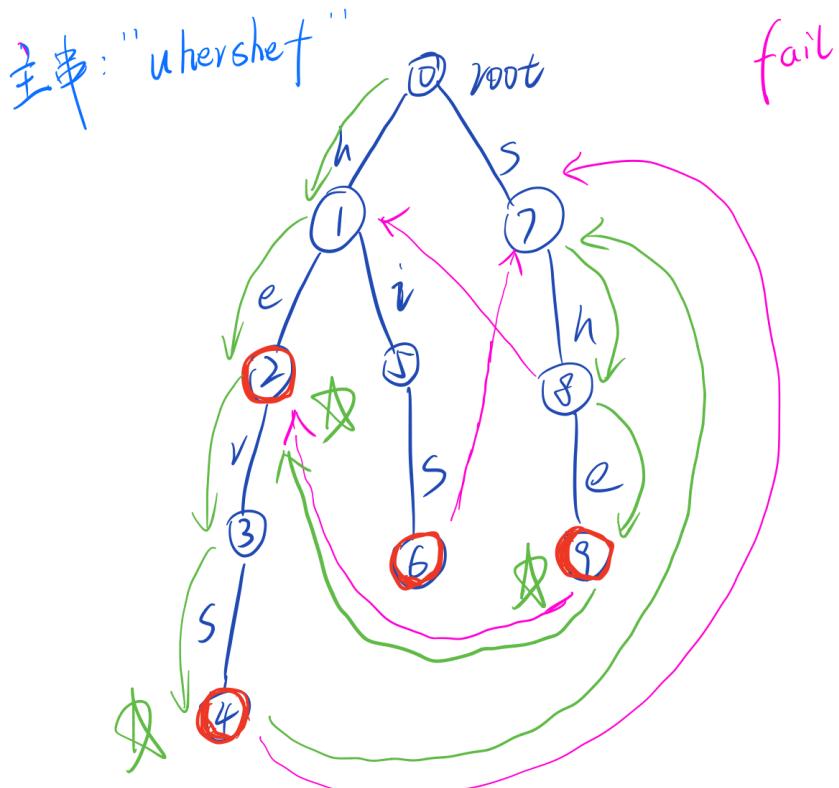
int query(string &s) {
    int p = 0;
    int n = s.length();
    for(int i = 0; i<n; i++) {
        int u = s[i] - 'a';
        if(!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}

```



有了这个fail数组后，我们模拟一个简单的样例，比如当前主串是uhershef

模式串: hers his she he



- | | |
|----------------|--------------------------|
| ① u, root无u儿子 | ④ hersh, 4号节点无h儿子 |
| ② h, root有u儿子 | ⑦失配, 走到fail(4)=7. |
| ③ he, ✓ | ⑧ sh ✓ |
| ④ her ✓ | ⑨ she ✓ 更新ans |
| ⑤ hers ✓ 更新ans | ⑩ shet e无f儿子 |
| | ⑪ 失配, 走到fail(9)=2, 更新ans |

在理解了这个过程之后，其实fail数组也就不再那么神秘了，当我们在一个分支上走不下去了之后，我们不必头铁，可以换个方向继续往下走，而fail数组的作用就是这样，当发生失配时，我们可以通过预处理出的fail数组告诉我们其他分支的信息(即满足当前字符串最长后缀的节点)，然后继续向后遍历即可。



```

void build() {
    int hh = 0, tt = -1;
    for(int i = 0; i<26; i++) {
        if(tr[0][i]) q[++tt] = tr[0][i]; //第一层的所有节点一定指向root
    }

    while(hh<=tt) {
        int t = q[hh++];
        for(int i = 0; i<26; i++) {
            int c = tr[t][i];
            //利用父节点的信息来更新自己的
        }
    }
}

```



H5 统计多个字符串出现次数



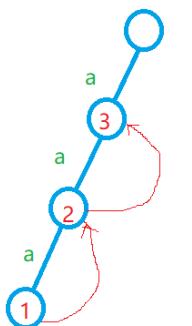
```

void solve() {
    memset(fail, 0, sizeof fail);
    memset(tr, 0, sizeof tr);
    memset(cnt, 0, sizeof cnt);
    idx=0;
    cin>>n;
    for(int i=1;i<=n;i++) {
        cin>>s;
        insert();
    }
    build();
    string ss;
    cin>>ss;
    int ans=0;
    for(int i=0, j=0;ss[i];i++) {
        int t=ss[i]-'a';
        j=tr[j][t];
        int p=j;
        while(p) {
            ans += cnt[p];//更新答案
            cnt[p] = 0; //统计出现的种类，所以清空标记
            p = fail[p]; //向上跳，不漏答案
        }
    }
    cout<<ans<<'\n';
}

```



我们用题目中的例子画一张图理解一下：



这里我把相应的cnt数组的值也写上，在这道题中cnt不仅仅只是记录每个单词出现的次数，而是记录前缀出现的次数



然后我们将Fail树拿出来，每一个节点代表一个前缀
然后我们倒序向上，将所有前驱节点的值累加到后继节点，这样就可以算出来每一个单词出现的次数了。

为什么可以这样呢，其实就是一个巧妙的转化，我们发现，要找所有单词中某个单词出现的次数，其实就是看在所有的前缀的后缀中某个单词出现的次数，这不就是fail数组的定义吗，问题也就解决了！

```

typedef pair<int, int> PII;
const int N=1e6+10;
int tr[N][26], idx;
int id[N];
int f[N];//f存下的是每一个点走过的次数
int fail[N];

```

```

int n;
vector<int> v;//存下每一个入队的点
string s;
void insert(int x) {
    int p=0;
    for(int i=0;s[i];i++) {
        int t=s[i]-'a';
        if(!tr[p][t]) tr[p][t]=++idx;
        p=tr[p][t];
        f[p]++;
    }
    id[x]=p;
}
void build() {
    queue<int> q;
    for(int i=0;i<26;i++) {
        if(tr[0][i]) q.push(tr[0][i]);
    }
    while(!q.empty()) {
        int t=q.front();
        q.pop();
        v.push_back(t);
        for(int i=0;i<26;i++) {
            int c=tr[t][i];
            if(!c) {
                tr[t][i]=tr[fail[t]][i];
            } else {
                fail[c]=tr[fail[t]][i];
                q.push(c);
            }
        }
    }
}
void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) {
        cin>>s;
        insert(i);
    }
    build();
    for(int i=idx-1;i>=0;i--) f[fail[v[i]]]+=f[v[i]];
    for(int i=1;i<=n;i++) {
        cout<<f[id[i]]<<'\\n';
    }
}

```

可持久化数据结构

h5 trie的可持久化

image-20230712171149176

image-20230712171208308

image-20230712171242251

image-20230712171259671

```
typedef pair<int, int> PII;
// 30w初始数据和30w新增，而10的7次方小于2的24次方，再加上根节点，就是说每个数最多需要25位；
const int N=6e5+10, M=N*25;
int s[N];// 前缀和序列
int n, m;
int tr[M][2];
int max_id[M], root[N], idx;// 用于记录当前根节点版本的最大id范围，更直白的说就是当前点对应要存的数据在前缀和数组s的位置(时间戳)
void insert(int k, int p, int q) {
    max_id[q]=k;
    for(int i=23;i>=0;i--) {
        int v=s[k]>>i&1;
        if(p) tr[q][v^1]=tr[p][v^1];
        tr[q][v]=++idx;
        max_id[tr[q][v]]=k;
        p=tr[p][v], q=tr[q][v];
    }
}
int query(int l, int r, int C)
{
    // 选用合适的root，就是第r-1个节点作为root(-1已在传参前完成)
    // 然后根据异或的前缀和性质才能保证在r左边
    int p = root[r];
    for (int i = 23; i >= 0; i--)
    {
        // C是s[n] ^ x，从高位到低位逐位检索二进制每一位上能跟C异或结果最大的数
        int v = C >> i & 1;
        // 自带判空功能如果没有点，max_id会为-1，那就肯定不能满足>=1（根据初始化max_id[0] = -1）
        // 如果没有这个初始化，max_id[0] 默认为0，而当l=0 的时候就会令到p误跳到空节点
        // 而max_id又同时可以限制当前的点是在l~r区间内
        // 另外，如果tr[p][v^1]为空，那么tr[p][v]就肯定不为空，并在l~r区间，因为根据
        // 插入的代码，每个节点至少有一条当前s[i]的完整路径
        // 而如果tr[p][v^1]不为空但maxid小于1，同理也能选取到tr[p][v]
        // printf("max_id[%d]: %d, p: %d, l: %d\n", tr[p][v ^ 1], max_id[tr[p][v ^ 1]], p, l);
        if (max_id[tr[p][v ^ 1]] >= 1) p = tr[p][v ^ 1];
        else p = tr[p][v];
    }
    return C ^ s[max_id[p]];
}
```

```

    }

void solve() {
    cin>>n>>m;
    // 前缀和, 初始化root[0]
    s[0]=0;
    max_id[0]=-1, root[0]=++idx;
    insert(0, 0, root[0]);
    for(int i=1;i<=n;i++) {
        int x;cin>>x;
        root[i]=++idx;
        s[i]=s[i-1]^x;
        insert(i, root[i-1], root[i]);
    }
    while(m--) {
        char op;
        cin>>op;
        if(op=='A') {
            int x;cin>>x;
            n++;
            root[n]=++idx;
            s[n]=s[n-1]^x;
            insert(n, root[n-1], root[n]);
        } else{
            int l,r,x;
            cin>>l>>r>>x;
            cout<<query(root[r-1], l-1, s[n]^x)<<'\\n' ;
        }
    }
}
}

```

h5 线段树的可持久化（主席树）



```

struct segment_tree_node {
    int l, r, sum, add;      //左子节点, 右子节点, 要维护的信息和懒标记, 这里我采用求和
} tr[4 * N + M * MAX_LOG]; //空间要充足
int root[M], idx;          //每次操作的根, 当前分配到哪个点了
void push_up (int u) {
    tr[u].sum = tr[tr[u].l].sum + tr[tr[u].r].sum;
}
void push_down (int u, int l, int r) {
    auto &root = tr[u], &left = tr[tr[u].l], &right = tr[tr[u].r];
    int mid = l + r >> 1;
    if (root.add) {
        left.add += root.add, left.sum += (mid - l + 1) * root.add;
        right.add += root.add, right.sum += (r - (mid + 1) + 1) * root.add;
        root.add = 0;
    }
}

```



```
int build (int l, int r) {    //返回[l, r]区间所组成线段树的编号。  
    int u = ++idx;    //动态开点  
    if (l == r) return u;  
    int mid = l + r >> 1;  
    tr[u] = {build (l, mid), build (mid + 1, r), 0};    //存下左儿子和右儿子的编号  
    return u;  
}
```



```
int insert (int v, int l, int r, int x, int d) {    //当前已经复制到v, 要把第x个数加上d, 当前的区间是[l, r]  
    int u = ++idx;    //动态开点  
    tr[u] = tr[v];    //拷贝节点  
    if (l == r) {  
        tr[u].sum += d;  
        return u;  
    }  
    push_down (u, l, r);  
    int mid = l + r >> 1;  
    if (x <= mid) tr[u].l = insert (tr[v].l, l, mid, x, d); //修改左子树  
    else tr[u].r = insert (tr[v].r, mid + 1, r, x, d);    //修改右儿子  
    push_up (u);  
    return u;  
}
```



```
int insert (int v, int l, int r, int L, int R, int d) {    //当前已经复制到v, 要把[L, R]加上d, 当前的区间是[l, r]  
    int u = ++idx;    //动态开点  
    tr[u] = tr[v];    //拷贝节点  
    if (L <= l && r <= R) {  
        tr[u].sum += (R - L + 1) * d;  
        tr[u].add += d;  
        return u;  
    }  
    push_down (u, l, r);  
    int mid = l + r >> 1;  
    if (L <= mid) tr[u].l = insert (tr[v].l, l, mid, x, d);  
    if (R >= mid + 1) tr[u].r = insert (tr[v].r, mid + 1, r, x, d);  
    push_up (u);  
    return u;  
}
```



```

int query (int u, int l, int r, int x) { //从第u个历史版本查询, 当前区间是[l, r], 查询x的值
    if (l == r) return tr[u].sum;
    push_down (u);
    int mid = l + r >> 1;
    if (x <= mid) return query (tr[u].l, l, mid, x);
    return query (tr[u].r, mid + 1, r, x);
}

```



```

int query (int u, int l, int r, int L, int R) { //从第u个历史版本查询, 当前区间是[l, r], 查询[L, R]的值
    if (L <= l && r <= R) return tr[u].sum;
    push_down (u);
    int mid = l + r >> 1;
    int ans = 0;
    if (L <= mid) ans += query (tr[u].l, l, mid, L, R);
    if (R >= mid + 1) ans += query (tr[u].r, mid + 1, r, L, R);
    return ans;
}

```



“可持久化线段树” 在本题目的作用实际上是指“使用多个入口指针保存多个不同版本的线段树”，类似Git，每个版本包含当前版本和之前的所有内容：每新增一个点都建一颗新树，复制前一棵树并插入新元素所造成的修改。实现上，是在build的时候只建立树的骨架而不保存元素，而对每一个点都做一次insert，在插入新点的同时，复制上一棵树，再插入新点



```

int insert(int p, int l, int r, int x)
{
    // 假设现在是从外界第一次执行insert, 那么调用的时候, p必定是根节点,
    // 那么q就相当于复制了一个根节点, 从节点q进入这棵树的时候, 也能得到之前的所有内容.
    // 同理, 再往下二分递归的时候, insert会继续复制根节点的左(右)子树, 一直递归直到l==r之前,
    // q和原来的p都是一毛一样. 直到l==r才真正插入了新点, 每次插入的时间空间复杂度都是lgk,
    // 总加起来就是lg1+lg2+...+lgn = lg(n!), 根据stirling公式, 结果为nlgn (大O)
    int q = ++idx;
    tr[q] = tr[p];
    if (l == r) // 如果区间长度为1, 说明就是放这里了
    {
        // tr[q].cnt++是表示插在这个叶节点上
        // 这个线段树只是保存的每个区间里面的元素个数
        // 每次插入都只是覆盖到的那堆区间里面的cnt发生+1
        tr[q].cnt++;
        return q;
    }
}

```

```

        int mid = l + r >> 1;
        if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x);
        else tr[q].r = insert(tr[p].r, mid+1, r, x);
        tr[q].cnt = tr[tr[q].l].cnt + tr[tr[q].r].cnt; // 相当于pushup了
        return q;
    }
}

```



```

const int N = 100010;
int n, m;
int a[N];
vector<int> nums;
struct Node
{
    int l, r;
    int cnt;
} tr[N * 4 + N * 17]; //N * 4 + NlogN
int root[N], idx;
int find(int x)
{
    return lower_bound(nums.begin(), nums.end(), x) - nums.begin();
}
// 对左右边界建立节点并编号, build是建立好骨架, 每个版本insert改不同数据
int build (int l, int r)
{
    int p = ++idx;
    if (l == r) return p;
    int mid = l + r >> 1;
    tr[p].l = build(l, mid), tr[p].r = build(mid+1, r);
    return p;
}
// l, r是要放入的坐标范围, x是要插入的数离散化后的位置
int insert(int p, int l, int r, int x)
{
    // 假设现在是从外界第一次执行insert, 那么调用的时候, p必定是根节点,
    // 那么q就相当于复制了一个根节点, 从节点q进入这棵树的时候, 也能得到之前的所有内容.
    // 同理, 再往下二分递归的时候, insert会继续复制根节点的左(右)子树, 一直递归直到l==r之前,
    // q和原来的p都是一毛一样. 直到l==r才真正插入了新点, 每次插入的时间空间复杂度都是lgk,
    // 总加起来就是lg1+lg2+...+lgn = lg(n!), 根据stirling公式, 结果为nlgn (大O)
    int q = ++idx;
    tr[q] = tr[p];
    if (l == r) // 如果区间长度为1, 说明就是放这里了
    {
        // tr[q].cnt++是表示插在这个叶节点上
        // 这个线段树只是保存的每个区间里面的元素个数
        // 每次插入都只是覆盖到的那堆区间里面的cnt发生+1
        tr[q].cnt++;
        return q;
    }
    int mid = l + r >> 1;
    if (x <= mid) tr[q].l = insert(tr[p].l, l, mid, x);
}

```

```

        else tr[q].r = insert(tr[p].r, mid+1, r, x);
        tr[q].cnt = tr[tr[q].l].cnt + tr[tr[q].r].cnt; // 相当于pushup了
        return q;
    }

// l,r是检索范围, q是当前第r个节点root[r]能包含l~r之间所有
// p的输入是root[1~1], 作用是剔除这个根节点所包含数据的影响
int query(int q, int p, int l, int r, int k)
{
    if (l == r) return r; // 如果找到位置
    // 目标是求l~r之间的第k小
    // tr[tr[q].l].cnt - tr[tr[p].l].cnt的结果是求出在p之后插入到q这些数之后,
    // 有多少个数(cnt)插入了p的左子树, 由于p的内容肯定不能出现在l~r之间(p根节点就是root[1~1]),
    // 所以cnt就是相当于"存在q左子树里面但不存在于l~r之间的数的个数"
    int cnt = tr[tr[q].l].cnt - tr[tr[p].l].cnt;
    int mid = l + r >> 1;
    // k <= cnt说明要找的元素在q的左子树里面, 同时这里面也要剔除掉包含在p左子树的内容
    if (k <= cnt) return query(tr[q].l, tr[p].l, l, mid, k);
    else return query(tr[q].r, tr[p].r, mid+1, r, k - cnt); // 类似同上
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++)
    {
        scanf("%d", &a[i]);
        nums.push_back(a[i]); // 离散化使用
    }
    // 离散化
    sort(nums.begin(), nums.end());
    nums.erase(unique(nums.begin(), nums.end()), nums.end());
    // 构造线段树, 构造n个版本的线段树
    // 第0个版本的什么都没有就用build, build是建立好骨架, 每个版本insert改不同数据
    root[0] = build(0, nums.size() - 1);
    // 后面的每插入一个点算一个版本, 每次插入都只是比上一个版本多1个数
    // 左右参数给0和nums.size()-1是因为离散化之后的值域就是在0~nums.size()-1之间
    // 要插入必须得把这些地方全包才能保证找得到插入点
    for (int i = 1; i <= n; i++)
        root[i] = insert(root[i-1], 0, nums.size() - 1, find(a[i]));
    while (m--)
    {
        int l, r, k;
        scanf("%d%d%d", &l, &r, &k);
        printf("%d\n", nums[query(root[r], root[l-1], 0, nums.size()-1, k)]);
    }
    return 0;
}

```

> 图论

树与图的存储

树是一种特殊的图，与图的存储方式相同。

对于无向图中的边ab，存储两条有向边a->b, b->a。

因此我们可以只考虑有向图的存储

H5 邻接矩阵

$g[a][b]=1$ 存储边a->b

如果有多条邻边可以用vector存储

```
vector<int> v[N];
v[a].push_back(b);
v[b].push_back(a);
```

H5 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);
```

连线很多的时候，对应的就是稠密图，显然易见，稠密图的路径太多了，所以就用点来找，也就是抓重点，用邻接矩阵存。

点很多，但是连线不是很多的时候，对应的就是稀疏图，稀疏图的路径不多，用邻接表存。

拓扑排序



拓扑排序：

一个有向图，如果图中有入度为 0 的点，把这个点删掉，同时也删掉这个点所连的边。

一直进行上面处理，如果所有点都能被删掉，则这个图可以进行拓扑排序。

- 首先记录各个点的入度
- 然后将入度为 0 的点放入队列
- 将队列里的点依次出队列，然后找出所有出队列这个点发出的边，删除边，同时边的另一侧的点的入度 -1。
- 如果所有点都进过队列，则可以拓扑排序，输出所有顶点。否则输出-1，代表不可以进行拓扑排序

vector+queue

```

#include<bits/stdc++.h>
using namespace std;
const int N=100010;
vector<int> p[N]; //vector变长数组来写邻接表
queue<int> q; //队列操作
int d[N]; //统计入度
int n, m, cnt, ans[N]; //ans数组记录答案, ans也可以用vector
int main()
{
    cin>>n>>m;
    for(int i=1;i<=m;i++)
    {
        int x, y;
        cin>>x>>y;
        p[x].push_back(y); //构造邻接表
        d[y]++; //入度++
    }
    for(int i=1;i<=n;i++)
    {
        if(!d[i]) q.push(i); //统计最初入度, 找入度为0的点
    }
    while(q.size())
    {
        int t=q.front();
        q.pop();
        ans[cnt++]=t;
        for(int i=0;i<p[t].size();i++)
        {
            d[p[t][i]]--; //删边操作
            if(d[p[t][i]]==0) q.push(p[t][i]); //如果删完边后入度为0了, 放入队列
        }
    }
    if(cnt==n) for(int i=0;i<cnt;i++) printf("%d ", ans[i]); //存在拓扑序列打印
    else printf("-1");
}

```

邻接表+模拟队列

```

#include <iostream>
#include <cstring>
#include <algorithm>
using namespace std;
const int N = 100010;
int e[N], ne[N], idx;//邻接表存储图
int h[N];
int q[N], hh = 0, tt = -1;//队列保存入度为0的点, 也就是能够输出的点,
int n, m;//保存图的点数和边数
int d[N];//保存各个点的入度

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

```

```

void topsort() {
    for(int i = 1; i <= n; i++) {//遍历一遍顶点的入度。
        if(d[i] == 0)//如果入度为 0, 则可以入队列
            q[++tt] = i;
    }
    while(tt >= hh) {//循环处理队列中点的
        int a = q[hh++];
        for(int i = h[a]; i != -1; i = ne[i]){//循环删除 a 发出的边
            int b = e[i];//a 有一条边指向b
            d[b]--;//删除边后, b的入度减1
            if(d[b] == 0)//如果b的入度减为 0, 则 b 可以输出, 入队列
                q[++tt] = b;
        }
    }
    if(tt == n - 1){//如果队列中的点的个数与图中点的个数相同, 则可以进行拓扑排序
        for(int i = 0; i < n; i++)//队列中保存了所有入度为0的点, 依次输出
            cout << q[i] << " ";
    }
    else//如果队列中的点的个数与图中点的个数不相同, 则可以进行拓扑排序
        cout << -1;//输出-1, 代表错误
}

```

```

int main() {
    cin >> n >> m;//保存点的个数和边的个数
    memset(h, -1, sizeof h);//初始化邻接矩阵
    while (m -- ){//依次读入边
        int a, b;
        cin >> a >> b;
        d[b]++;
        add(a, b); //添加到邻接矩阵
    }
    topsort(); //进行拓扑排序
    return 0;
}

```

家谱树

```

const int N=110, M=N*N;
queue<int> q;
int d[N];
int h[N], e[M], ne[M], idx;
int n;
vector<int> ans;
void add(int a, int b) {
    e[idx]=b, ne[idx]=h[a], h[a]=idx++;
}
void topsort() {
    for(int i=1; i<=n; i++) {
        if(d[i]==0) {

```

```

        q.push(i);
    }
}

while(!q.empty()) {
    int t=q.front();
    q.pop();
    ans.push_back(t);
    for(int i=h[t];~i;i=ne[i]){
        int j=e[i];
        if(--d[j]==0){
            q.push(j);
        }
    }
}
void solve(){
    cin>>n;
    memset(h,-1,sizeof h);
    for(int i=1;i<=n;i++){
        int x;
        while(cin>>x,x){
            add(i,x);
            d[x]++;
        }
    }
    topsort();
    for(int i=0;i<ans.size();i++) cout<<ans[i]<<"\n"[i==ans.size()-1];
}

```

奖金

奖金、

设 a 比 b 工资高, b 比 c 工资高.

则有 $a > b > c$

可以建图 $\begin{array}{c} a \rightarrow b \rightarrow c \\ \downarrow \end{array}$

是拓扑序. \Rightarrow 用拓扑排序模板



① 找不到合理方案: 图中存在环.

记承拓扑序数组 $a[i]$ 中元素个数 $< n$.

② 找到合理方案:

找到拓扑序, 从 100 元开始加, 依次 +1 即可.

总和即为所求结果。

几个注意点:

① a 的奖金比 b 要高

means, b 的基础上增加奖金才为 a .

故 $b \uparrow a$

所以建边时 $add(b, a)$.

② 在最初加入所有入度为 0 的点进队列时,

是基本工资 100

用队列中的一个点发散时,
它的连边终点的入度减 1 为 0 时
应当将其奖金 +1.

```
typedef pair<int, int> PII;
const int N=1e4+10, M=2e4+10;
int n, m;
int h[N], e[M], ne[M], idx;
int d[N];
int dist[N];
void add(int a, int b) {
    e[idx]=b, ne[idx]=h[a], h[a]=idx++;
}
void solve() {
    cin>>n>>m;
    memset(h, -1, sizeof h);
    for(int i=1; i<=m; i++) {
        int a, b;
        cin>>a>>b;
        add(b, a);
        d[a]++;
    }
    queue<int> q;
    vector<int> ans;
    for(int i=1; i<=n; i++) {
        if(!d[i]) {
            q.push(i);
        }
    }
    while(!q.empty()) {
        auto t=q.front();
```

```

        q.pop();
        ans.push_back(t);
        for(int i=h[t];~i;i=ne[i]){
            int j=e[i];
            if(--d[j]==0){
                q.push(j);
            }
        }
    }
    if(ans.size()!=n){
        cout<<"Poor Xed"<<'\n';
    }else{
        for(int i=1;i<=n;i++) dist[i]=100;
        int res=0;
        for(int i=0;i<ans.size();i++){
            int j=ans[i];
            for(int k=h[j];~k;k=ne[k]){
                if(dist[e[k]]<dist[j]+1){
                    dist[e[k]]=dist[j]+1;
                }
            }
        }
        for(int i=1;i<=n;i++){
            res+=dist[i];
        }
        cout<<res<<'\n';
    }
}

```

车站分级



```

const int N=2010,M=1e6+10;//加上虚拟点总共有1000+1000=2000个
int h[N], e[M], ne[M], w[M], idx;
int n, m;
int d[N];
int dist[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++;
    d[b]++;
}
void solve(){
    cin>>n>>m;
    memset(h, -1, sizeof h);
    for(int i=1;i<=m;i++){
        int cnt;
        cin>>cnt;
        memset(st, 0, sizeof st);
        int S=n, T=1;//每一趟车次中找到起点和终点
        for(int j=1;j<=cnt;j++) {

```

```

        int x;cin>>x;
        S=min(S, x), T=max(T, x) ;
        st[x]=1;
    }

    int ver=n+i;//虚拟点
    for(int j=S;j<=T;j++) {
        if(!st[j]){
            add(j, ver, 0);
        }else{
            add(ver, j, 1);
        }
    }
}

//拓扑排序
queue<int> q;
vector<int> ans;
//注意：加上虚拟点后有n+m个点
for(int i=1;i<=n+m;i++) {
    if(!d[i]){
        q.push(i);
    }
}
while(!q.empty()){
    auto t=q.front();
    q.pop();
    ans.push_back(t);
    for(int i=h[t];~i;i=ne[i]){
        int j=e[i];
        if(--d[j]==0){
            q.push(j);
        }
    }
}
for(int i=1;i<=n;i++) dist[i]=1;
for(int i=0;i<ans.size();i++) {
    int j=ans[i];
    for(int k=h[j];~k;k=ne[k]){
        dist[e[k]]=max(dist[e[k]], dist[j]+w[k]);
    }
}
int res=0;
for(int i=1;i<=n;i++) res=max(res,dist[i]);
cout<<res<<'\n';
}

```

最短路



H5 堆优化版dijkstra (稀疏图)

时间复杂度 $O(m\log n)$, n 表示点数, m 表示边数

一号点的距离初始化为零, 其他点初始化成无穷大。

将一号点放入堆中。

不断循环, 直到堆空。每一次循环中执行的操作为:

弹出堆顶 (找到S外距离最短的点, 并标记该点的最短路径已经确定)。

用该点更新临界点的距离, 若更新成功就加入到堆中。

```
#include<iostream>
#include<cstring>
#include<queue>
using namespace std;
typedef pair<int, int> PII;
const int N = 100010;
// 稀疏图用邻接表来存, 稠密图用矩阵存
int h[N], e[N], ne[N], idx;
int w[N]; // 用来存权重
int dist[N];
bool st[N]; // 如果为true说明这个点的最短路径已经确定
int n, m;
void add(int x, int y, int c)
{
    // 有重边也不要紧, 假设1->2有权重为2和3的边, 再遍历到点1的时候2号点的距离会更新两次放入堆中
    // 这样堆中会有很多冗余的点, 但是在弹出的时候还是会弹出最小值2+x (x为之前确定的最短路径),
    // 并标记st为true, 所以下一次弹出3+x会continue不会向下执行。
    w[idx] = c;
    e[idx] = y;
    ne[idx] = h[x];
    h[x] = idx++;
}

int dijkstra()
{
    memset(dist, 0x3f, sizeof(dist));
    dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap; // 定义一个小根堆
    // 这里heap中为什么要存pair呢, 首先小根堆是根据距离来排的, 所以有一个变量要是距离,
```

// 其次在从堆中拿出来的时候要知道知道这个点是哪个点，不然怎么更新邻接点呢？所以第二个变量要存点。

```
heap.push({ 0, 1 }); // 这个顺序不能倒，pair排序时是先根据first，再根据second，  
// 这里显然要根据距离排序  
while(heap.size())  
{  
    PII k = heap.top(); // 取不在集合S中距离最短的点  
    heap.pop();  
    int ver = k.second, distance = k.first;  
  
    if(st[ver]) continue;  
    st[ver] = true;  
  
    for(int i = h[ver]; i != -1; i = ne[i])  
    {  
        int j = e[i]; // i只是个下标，e中存的是i这个下标对应的点。  
        if(dist[j] > distance + w[i])  
        {  
            dist[j] = distance + w[i];  
            heap.push({ dist[j], j });  
        }  
    }  
}  
if(dist[n] == 0x3f3f3f3f) return -1;  
else return dist[n];  
}  
  
int main()  
{  
    memset(h, -1, sizeof(h));  
    scanf("%d%d", &n, &m);  
  
    while (m--)  
    {  
        int x, y, c;  
        scanf("%d%d%d", &x, &y, &c);  
        add(x, y, c);  
    }  
  
    cout << dijkstra() << endl;  
  
    return 0;  
}
```

不能使用 Dijkstra 解决含负权图的问题

Dijkstra不能解决负权边是因为 Dijkstra要求每个点被确定后 $st[j] = \text{true}$, $dist[j]$ 就是最短距离了，之后就不能再被更新了（一锤子买卖），而如果有负权边的话，那已经确定的点的 $dist[j]$ 不一定是最短了

H5 Bellman-Ford算法

时间复杂度 $O(nm)$, n为点数, m为边数

Bellman - ford 算法是求含负权图的单源最短路径的一种算法，效率较低，代码难度较小。其原理为连续进行松弛，在每次松弛时把每条边都更新一下，若在 $n-1$ 次松弛后还能更新，则说明图中有负环，因此无法得出结果，否则就完成。

(通俗的来讲就是：假设 1 号点到 n 号点是可达的，每一个点同时向指向的方向出发，更新相邻的点的最短距离，通过循环 $n-1$ 次操作，若图中不存在负环，则 1 号点一定会到达 n 号点，若图中存在负环，则在 $n-1$ 次松弛后一定还会更新)

具体步骤

1. for n次

```
for 所有边 a,b,w (松弛操作)
    dist[b] = min(dist[b],back[a] + w)
```

2. 注意：back[] 数组是上一次迭代后 dist[] 数组的备份，由于是每个点同时向外出发，因此需要对 dist[] 数组进行备份，若不进行备份会因此发生串联效应，影响到下一个点

3. 是否能到达n号点的判断中需要进行if(dist[n] > INF/2)判断，而并非是if(dist[n] == INF)判断，原因是 INF是一个确定的值，并非真正的无穷大，会随着其他数值而受到影响，dist[n]大于某个与INF相同数量级的数即可

bellman - ford算法擅长解决有边数限制的最短路问题

```
#include<iostream>
#include<cstring>
using namespace std;
const int N = 510, M = 10010;
struct Edge {
    int a;
    int b;
    int w;
} e[M];//把每个边保存下来即可
int dist[N];
int back[N];//备份数组防止串联
int n, m, k;//k代表最短路径最多包含k条边

int bellman_ford() {
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    for (int i = 0; i < k; i++) {//k次循环
        memcpy(back, dist, sizeof dist);
        for (int j = 0; j < m; j++) {//遍历所有边
            int a = e[j].a, b = e[j].b, w = e[j].w;
            dist[b] = min(dist[b], back[a] + w);
        }
        //使用backup:避免给a更新后立马更新b, 这样b一次性最短路径就多了两条边出来
    }
    if (dist[n] > 0x3f3f3f3f / 2) return -1;
    else return dist[n];
}
```

```

int main() {
    scanf("%d%d%d", &n, &m, &k);
    for (int i = 0; i < m; i++) {
        int a, b, w;
        scanf("%d%d%d", &a, &b, &w);
        e[i] = {a, b, w};
    }
    int res = bellman_ford();
    if (res == -1) puts("impossible");
    else cout << res;

    return 0;
}

```

H5 spfa 算法

spfa 算法是队列优化的Bellman-Ford算法

时间复杂度 O(m), 最坏情况下 O(nm) ,n表示点数, m表示边数

数据恶心就会故意卡你

```

int n;           // 总点数
int h[N], w[N], e[N], ne[N], idx;           // 邻接表存储所有边
int dist[N];           // 存储每个点到1号点的最短距离
bool st[N];           // 存储每个点是否在队列中

// 求1号点到n号点的最短路距离, 如果从1号点无法走到n号点则返回-1
int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;

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

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                if (!st[j])           // 如果队列中已存在j, 则不需要将j重复插入
                {

```

```

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

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

```

比Dijkstra快，但不稳

H5 spfa找负环

负环：存在一个环路，其边权之和<0。

```

/*
负环：一个有向图/无向图中 环路的边权和<0
因为是一个环，所以可以循环无限次，那么这些环上的点的距离就会变成-∞

求负环：
基于spfa
spfa 每入队一次 就相当于更新一次 如果入队>=n次
在bellman_ford中 每更新一次 最短距离变小 但一个点的最短距离不可能变小n次
1 统计每个点入队的次数 如果某个点入队n次
    说明存在负环
    ←o
    ↓ ↑
o→o→o...o→o 总共n个点，则对于点i到其他点最多n-1条边，入队n次说明包含n条让dist[i]变小的边
    同时因为更新原则是加上第n条边后最短路权重变小
    所以第n条边是负的 则该路径一定存在负环
2 统计当前每个点的最短路中所包含的边数，如果某个点的最短路所包含的边数>=n
    说明存在负环
n条边 则一定有n+1个点 但我们总共就n个点 所以这条最短路上一定有环
    同时因为更新原则是加上第n条边后最短路权重变小
    所以第n条边是负的 则该路径一定存在负环

```

推荐第2种方法：

考虑：

当数据如 -1

```

o←o
-1 ↓ ↑ -1
o→o
-1

```

如果用第一种方法 转完一圈之后每个点只入队一次，达到判定要求则需要转n圈

$O(n^2)$

如果用第二种方法 转完一圈之后就能达到判定要求

$O(n)$

还有一个问题：

负环不一定从起点走到

4
↓ ↑
1→... 2→3

解决方案：

将所有点入队的同时把所有点的距离初始化为0

```
q.push(node) for all_node  
dist[node] = 0 for all_node
```

why 所有点入队？（结合虚拟源点建新图理解）

1 虚拟源点0向所有点连一条长度是0的边构成一条新的图

同时以虚拟源点0作为新图的起点

2 原图中存在负环 == 新图中存在负环

而新图里所有的负环一定能从虚拟源点出发走到

3 那么我们对新图做spfa时就是把虚拟源点0加入queue

而0 pop出来后队列会把所有原图的节点加入queue

why $dist[node]=0$ ？

1 有负环 == 做完spfa后 存在点 i $dist[i] = -\infty$

2 赋的初值0也是有限值，做完spfa后都会变成 $-\infty$

3 $w[node]$ 都是有限值 则必然要更新无限次（更新次数 $>n$ ）

最后来个玄学操作

spfa 0(m) ~ 0(nm)

当spfa效率比较低的时候（一直结束不了的时候）

等价于 存在负环

可测量化：当所有点入队次数超过 $2n$ ，我们就认为图中很大可能存在负环

```
*/  
//虫洞  
typedef pair<int, int> PII;  
const int N=510, M=5210;  
int h[N], e[M], ne[M], w[M], idx;  
bool st[N];  
int n, m1, m2;  
int cnt[N];  
int dist[N];  
void add(int a, int b, int c) {  
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;  
}  
bool spfa () {  
    queue<int> q;  
    for(int i=1; i<=n; i++) q.push(i), st[i]=1;  
    memset(dist, 0, sizeof dist);  
    memset(cnt, 0, sizeof cnt);  
    while(!q.empty()) {  
        auto t=q.front();  
        q.pop();  
        st[t]=0;  
        for(int i=h[t]; i!=ne[i]; i=ne[i]) {  
            int j=e[i];  
            if(dist[j]>dist[t]+w[i]) {  
                cnt[j]=cnt[t]+1;  
                if(cnt[j]>=n) {  
                    return true;  
                }  
            }  
        }  
    }  
}
```

```

        }
        dist[j]=dist[t]+w[i];
        if(!st[j]) {
            q.push(j);
        }
    }
}

return false;
}

void solve() {
    cin>>n>>m1>>m2;
    memset(h,-1,sizeof h);
    idx=0;
    for(int i=1;i<=m1;i++) {
        int a,b,c;
        cin>>a>>b>>c;
        add(a,b,c),add(b,a,c);
    }
    for(int i=1;i<=m2;i++) {
        int a,b,c;
        cin>>a>>b>>c;
        add(a,b,-c);
    }
    if(spfa()) {
        cout<<"YES"<<'\n';
    } else{
        cout<<"NO"<<'\n';
    }
}

int main() {
    ios::sync_with_stdio(0);cin.tie(0),cout.tie(0);
    int T=1;
    cin>>T;
    while(T--) {
        solve();
    }
    return 0;
}
}

```

h5 01分数规划+spfa判最长路是否有负环

观光奶牛



```

const int N=1010,M=5010;
int wf[N];
int h[N],e[M],ne[M],wt[M],idx;
double dist[N];
bool st[N];

```

```

int cnt[N];
int n, m;
bool check(double x) {
    queue<int> q;
    for (int i=1; i<=n; i++) {
        cnt[i]=0, st[i]=1;
        q.push(i);
    }
    while (!q.empty()) {
        auto t=q.front();
        q.pop();
        st[t]=0;
        for (int i=h[t]; ~i; i=ne[i]) {
            int j=e[i];
            if (dist[j]<dist[t]+wf[t]-wt[i]*x) {
                dist[j]=dist[t]+wf[t]-wt[i]*x;
                cnt[j]=cnt[t]+1;
                if (cnt[j]>=n) {
                    return true;
                }
                if (!st[j]) {
                    st[j]=1;
                    q.push(j);
                }
            }
        }
    }
    return false;
}
void add(int a, int b, int c) {
    e[idx]=b, wt[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
void solve() {
    cin>>n>>m;
    for (int i=1; i<=n; i++) cin>>wf[i];
    memset(h, -1, sizeof h);
    for (int i=1; i<=m; i++) {
        int a, b, c;
        cin>>a>>b>>c;
        add(a, b, c);
    }
    double l=0, r=1010;
    while (r-l>1e-4) {
        double mid=(l+r)/2;
        if (check(mid)) l=mid;
        else r=mid;
    }
    printf("%.2lf\n", l);
}

```

单词环



```
//将n个字符串的首尾看成点，字符串的长度看为边
typedef pair<int, int> PII;
const int N=676, M=1e5+10;
int h[N], e[M], ne[M], w[M], idx;
double dist[N];
bool st[N];
int cnt[N];
int n;
void add(int a, int b, int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
bool check(double x) {
    queue<int> q;
    for(int i=0; i<676; i++) {
        q.push(i);
        dist[i]=0, cnt[i]=0, st[i]=1;
    }
    int count=0;
    while(!q.empty()) {
        auto t=q.front();
        q.pop();
        st[t]=0;
        for(int i=h[t]; ~i; i=ne[i]) {
            int j=e[i];
            if(dist[j]<dist[t]+w[i]-x) {
                dist[j]=dist[t]+w[i]-x;
                cnt[j]=cnt[t]+1;
                //若更新超过一个某一个数时，则武断认为有正环（玄学）
                if(++count>1e4) {
                    return true;
                }
                if(cnt[j]>=676) {
                    return true;
                }
                if(!st[j]) {
                    q.push(j);
                    st[j]=1;
                }
            }
        }
    }
    return false;
}
void solve() {
    while(cin>>n, n) {
        memset(h, -1, sizeof h);
        idx=0;
        for(int i=0; i<n; i++) {
            string s;
            cin>>s;
        }
    }
}
```

```

        if(s.size()>=2) {
            int a=(s[0]-'a')*26+(s[1]-'a');
            int b=(s[s.size()-2]-'a')*26+(s[s.size()-1]-'a');
            // cout<<a<<' '<<b<<'\n';
            add(a,b,s.size());
        }
    }

    //当mid=0时，最容易出现正环，若没有，则无解
    if(!check(0)) {
        cout<<"No solution"<<'\n';
        continue;
    }

    double l=0, r=1010;
    while(r-l>1e-4) {
        double mid=(l+r)/2;
        if(check(mid)) {
            l=mid;
        } else{
            r=mid;
        }
    }
    printf("%.4lf\n", l);
}
}

```

h5 Floyd

时间复杂度 O(n^3) , n表示点数

主要思路就是通过其他的点进行中转来求的两点之间的最短路。因为我们知道，两点之间有多条路，如果换一条路可以缩短距离的话，就更新最短距离。而它最本质的思想，就是用其他的点进行中转，从而达到求出最短路的目的。

```

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]);
}

```

多源多汇最短路，求出任意两点之间的最短路

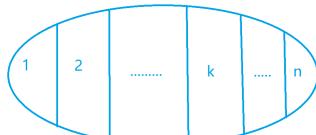
H6 传递闭包

将所有间接连接的点直接连一条边

```
for(int k = 1; k <= n; k++)
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++)
            d[i][j] |= d[i][k] & d[k][j]; //表示如果i到k可达，并且k到j可达，那么i到j可达
```

H6 找最小环（一般是正权图）

1. 集合含义：图中所有的环
2. 题目宿求：所有环中权重最小环
3. 集合划分：以环中点编号的最大编号划分 -> 不重不漏

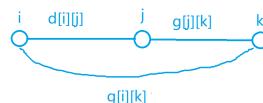


题目就可以转化为每一类的最小环再取一个min

每一类的最小环求法如下

枚举环中点编号最大为k的所有环：

枚举小于k的任意两个点i, j, 如果k和i, k和j之间有边，那么只要 $d[i][j] << inf$, 即有环。



```
for(int k = 0; k < n; k++)
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
```

此处的d数组, $d[i][j]$ 指的是经过1 - (k - 1)号点, i到j的最短距离
环的权重 = $d[i][j] + g[k][i] + g[j][k]$;
由于 $g[k][i]$ 和 $g[j][k]$ 的边长已经确定了, 所以要想求得权重最小的环就是求 $d[i][j]$ 得最小值, 而在跑floyd得时候, 就是保证了 $d[i][j]$ 此时为最小值, 这个环的权重就是这类得最小值了。

最小环权重: 所以res只需要在这些跑floyd得同时求出最小值即可。

最小环方案求法如下：

记录环中每个点, i, j, k都是已知的点, 而i -> j路径中经过了哪些点转移过来, 如下图:



```
d[i][j] = d[i][k1] + d[k1][j];
1) d[k1][j] = g[k1][j] << inf;
2) d[i][k1] = d[i][k2] + d[k2][k1];
d[i][k2] = d[i][k3] + d[k3][k2];
d[i][k3] = g[i][k3] << inf; // d[i][k3]这条边已经是最短距离了 -> pos[i][k3] = 0 没有点松弛的了他了
```

所以最小环的方案可以用递归函数求解, 将点依次放入path数组, pos数组用于记录i, j之间的最短距离的松弛点。

因为题目求最小环，所以一定没有如图所示这种情况，因为如果环为正环，舍掉这个环的话最小环的权重将会减小，如果环为负环，则会一直跑下去，最小环就一直小，也一定不会是负环。所以， $i \rightarrow j$ 的路径上必定没有环的存在。即， j 松弛点左右路径无重叠，左边的点只可能是经过 k 到右边的点，即左右的松弛点不会有同样的，即在 $\text{get_path}(i, k)$ 时， k, j 之间必定没有路径， $\text{pos}[k][j]$ 都等于0，右边的点绝不会放入 path 数组。只有当 k 回退到 i 时（左边的点放完时），才开始放右边的点。



注：i, j路径经过的点只可能是松弛点，a, b, c都是松弛点

如图: $d[i][i] \equiv d[i][b] + d[a][i] + d[a][b]$

在`get path(i, b)`时，松弛点可能为a，`get path(a, i)`就将放入右边的点而左边的点还未放完，是不符合要求的。

但题目保证了这种路径的不存在性，所以这种情况也就不可能存在了。

```

void get_path (int i, int j)
{
    // 无松弛点, 无路径
    if (pos[i][j] == 0) return ;

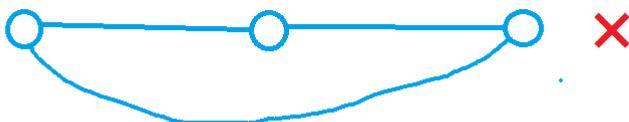
    int k = pos[i][j];
    get_path(i, k);
    path[cnt ++] = k;
    get_path(k, j);

}

```



如果只用d数组的话， j ， k 之间就会被更新为有边的，在找最小环时 i ， j ， k 将被看作一个环，可能最小环被更新，所以错误。



```
const int N = 110, inf = 0x3f3f3f3f;
int n, m;
int d[N][N]; // floyd数组 也就是dp中的f降维数组
int g[N][N]; // 存边数组
int pos[N][N]; // 松弛点数组
int path[N], cnt; // 方案数组
int res = inf; // 最小环权重
// 递归函数见图解1
void get_path (int i, int j) // 记录i到j的松弛点
{
    if(pos[i][j] == 0) return ; // 无松弛点
    int k = pos[i][j];
    get_path(i, k);
    path[cnt ++ ] = k;
    get_path(k, j); // 没有这样的路径就更没有需要加入的松弛点了
}
int main ()
{
    cin >> n >> m;
    memset(g, 0x3f, sizeof g);
}
```

```

for(int i = 1 ; i <= n ; i ++ ) g[i][i] = 0;
for(int i = 0 ; i < m ; i ++ )
{
    int a, b, c;
    cin >> a >> b >> c;
    g[a][b] = g[b][a] = min(g[a][b], c); //可能有重边
}
/*
memcpy而不只用d数组的原因：
当j与k无边，i与k有边，i可以走到j时，如果只用d数组在跑floyd的时候，d[j][k] == inf会被d[j][i] + d[i][k]更新成有边。
但实际上是没有边的。见图解2。
*/
memcpy(d, g, sizeof d);
for(int k = 1 ; k <= n ; k ++ )
{
    for(int i = 1 ; i < k ; i ++ )
        for(int j = i + 1 ; j < k ; j ++ ) // 由于是无向边所以i, j的位置交换一下也是一样的，所以j从i + 1开始
            if((long long)d[i][j] + g[i][k] + g[j][k] < res) // 由于d数组初始化为inf所以有可能爆int，这里注意
{
            res = d[i][j] + g[i][k] + g[j][k]; //求最小环
            // 存方案 顺序为k -> i -> ... -> j
            cnt = 0;
            path[cnt ++ ] = k;
            path[cnt ++ ] = i;
            get_path(i, j);
            path[cnt ++ ] = j;
        }
        for(int i = 1 ; i <= n ; i ++ )
            for(int j = 1 ; j <= n ; j ++ )
                if(d[i][j] > d[i][k] + d[k][j])
{
                    d[i][j] = d[i][k] + d[k][j];
                    pos[i][j] = k;
                }
}
if(res == inf) puts("No solution.");
else
{
    for(int i = 0 ; i < cnt ; i ++ ) cout << path[i] << ' ';
    cout << endl;
}
return 0;
}

```

H6 恰好经过k条边的最短路（倍增）

牛站

状态表示:

$d[k][i][j]$ 表示 i 走到 j 正好经过 k 条路径的最短距离;

状态转移方程:

$d[a+b][i][j] = \min(d[a+b][i][j], d[a][i][k] + d[b][k][j]);$

本题可以采取快速幂的思想:

因为要走 k 条边, 所以可以先将 k 转化为二进制数:

例如 $k=38 \rightarrow k=(100110)_2$, 设 g 数组为走过一条边的最短距离;

则可以看成是:

答案数组 = g 数组走过 2 条边的最短距离 + g 数组走过 4 条边的最短距离 + g 数组走过 32 条边的最短距离

按照此类倍增的思想即可大大降低时间复杂度, 以指数的形势逼近答案;

g 数组: (初始状态)

0x3f	1	9	0x3f
1	0x3f	8	3
9	8	5	2
0x3f	3	2	0x3f

不断倍增 →

g 数组表示的是经过一条边, 各个点之间距离的矩阵, 因为要

经过一条边, 所以 $g[i][i]$ 并不能初始化为 0, 只有当 $i \neq i$ 有一条

边的时候才能更新 $g[i][i]$;

g 数组的倍增:

只要将两个 g 数组单独分开来看即可;

例如 g 数组从经过两条边的最短距离转变为经过四条边的最短距离;

$temp[i][j] = \min(temp[i][j], g[i][k] + g[k][j]);$ 这样 g 数组可以从两条边转化为 4 条边

为什么这样做是可行的呢?

因为 $g[i][k]$ 表示的是经过 $i \rightarrow k$ 经过两条边的最短距离, $g[k][j]$ 表示的是 $k \rightarrow j$ 只经过两条边的最短距离,

所以 $g[i][k] + g[k][j]$ 表示的是 $i \rightarrow k \rightarrow j$ 经过四条边的最短的距离;

```
#include<iostream>
#include<cstring>
#include<map>

using namespace std;

const int N=210;

int res[N][N], g[N][N];
int k, n, m, S, E;
map<int, int> id;

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

```

        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
                temp[i][j]=min(temp[i][j],a[i][k]+b[k][j]);
        memcpy(c,temp,sizeof temp);
    }

void qmi()
{
    memset(res,0x3f,sizeof res);
    for(int i=1;i<=n;i++) res[i][i]=0;//经过0条边
    while(k)//更新的过程
    {
        //矩阵乘法
        if(k&1) mul(res,res,g);//res=res*g;根据k决定是否用当前g的结果去更新res
        mul(g,g,g);//g=g*g;g的更新
        k>>=1;
    }
}

int main()
{
    cin>>k>>m>>S>>E;//虽然点数较多，但由于边数少，所以我们实际用到的点数也很少，可以使用map来离散化来赋予
    //他们唯一的编号
    memset(g,0x3f,sizeof g);
    //这里我们来解释一下为什么不去初始化g[i][i]=0呢？
    //我们都知道在类Floyd算法中有严格的边数限制，如果出现了i->j->i的情况其实在i->i中我们是有2条边的
    //要是我们初始化g[i][i]=0，那样就没边了，影响了类Floyd算法的边数限制！
    if(!id.count(S)) id[S]=++n;
    if(!id.count(E)) id[E]=++n;
    S=id[S],E=id[E];
    while(m--)
    {
        int a,b,c;
        scanf("%d%d%d",&c,&a,&b);
        if(!id.count(a)) id[a]=++n;
        if(!id.count(b)) id[b]=++n;
        a=id[a],b=id[b];
        g[a][b]=g[b][a]=min(g[a][b],c);
    }
    qmi();
    cout<< res[S][E] << endl;
    return 0;
}

```

H5 文字复习总结

Dijkstra-朴素O(n^2)

1. 初始化距离数组, $dist[1] = 0$, $dist[i] = \text{inf}$;
2. for n次循环 每次循环确定一个min加入S集合中, n次之后就得出所有的最短距离
3. 将不在S中 $dist_{\text{min}}$ 的点->t
4. t->S加入最短路集合
5. 用t更新到其他点的距离

Dijkstra-堆优化O(mlogm)

1. 利用邻接表, 优先队列
2. 在priority_queue中将返回堆顶
3. 利用堆顶来更新其他点, 并加入堆中类似宽搜

Bellman_fordO(nm)

1. 注意连锁想象需要备份, struct Edge{int a,b,c} Edge[M];
2. 初始化dist, 松弛 $dist[x.b] = \min(dist[x.b], backup[x.a]+x.w)$;
3. 松弛k次, 每次访问m条边

Spfa O(n)~O(nm)

1. 利用队列优化仅加入修改过的地方
2. for k次
3. for 所有边利用宽搜模型去优化bellman_ford算法
3. 更新队列中当前点的所有出边

Floyd O(n^3)

1. 初始化d
2. k, i, j 去更新d

H5 最短路常考模型总结

DP+最短路, 二分+最短路, 反向建图, 虚拟源点, 求最短路的最大值, 求多个点的最短路, 最短路求和, 求最长路, 转化01边权双端队列BFS求最短路, 暴力枚举+最短路, 拓扑图+联通块求最短路, 正反各跑一次最短路

H6 建立虚拟源点

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
#include <map>
#include <set>
#define ll long long
#define mfp make_pair
using namespace std;
typedef pair<int, int> PII;
const int N=1010, M=1e5+10;
int h[N], e[M], ne[M], w[M], idx;
int dist[N];
```

```

bool st[N];
int n, m, T, cnt;
void add(int a, int b, int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
int dij() {
    memset(dist, 0x3f, sizeof dist);
    memset(st, 0, sizeof st);
    priority_queue<PII, vector<PII>, greater<PII>> q;
    q.push({0, 0});
    dist[0]=0;
    while(!q.empty()) {
        auto t=q.top();
        q.pop();
        int id=t.second;
        if(st[id]) continue;
        st[id]=1;
        for(int i=h[id]; i<ne[i]; i++) {
            int j=e[i];
            if(dist[j]>w[i]+dist[id]) {
                dist[j]=w[i]+dist[id];
                q.push({dist[j], j});
            }
        }
    }
    if(dist[T]==0x3f3f3f3f) return -1;
    return dist[T];
}
void solve() {
    while(cin>>n>>m>>T) {
        memset(h, -1, sizeof h);
        idx=0;
        for(int i=1; i<=m; i++) {
            int a, b, c;
            cin>>a>>b>>c;
            add(a, b, c);
        }
        cin>>cnt;
        for(int i=1; i<=cnt; i++) {
            int x; cin>>x;
            add(0, x, 0);
        }
        cout<<dij()<<'\n';
    }
}
int main() {
    ios::sync_with_stdio(0); cin.tie(0), cout.tie(0);
    int T=1;
//    cin>>T;
    while(T--) {
        solve();
    }
}

```

```
    return 0;
}
```

H6 拆点，分层图

所谓拆点 -> 就是当前的点，状态的表示方法无法区分一些本来不同的点，状态。我们通过升维对状态多增加一个约束(描述)，使得状态，点再细分，从而达到清楚描述点与状态之间的方法

拯救大兵瑞恩

```
#define x first
#define y second
using namespace std;
typedef pair<int, int> PII;
const int N=11, M=400;
const int dir[] [2]={ {1, 0}, {-1, 0}, {0, 1}, {0, -1} } ;
int h[N*N], e[M], ne[M], w[M], idx;
set<PII> edge;
int g[N][N], key[N*N];
int dist[N*N] [1<<N];
bool st[N*N] [1<<N];
int n, m, k, p;
void add(int a, int b, int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
void build() {
    for(int i=1; i<=n; i++) {
        for(int j=1; j<=m; j++) {
            int a=g[i][j];
            for(int k=0; k<4; k++) {
                int xx=i+dir[k][0];
                int yy=j+dir[k][1];
                if(xx>=1 && xx<=n && yy>=1 && yy<=m) {
                    int b=g[xx][yy];
                    if(!edge.count({a, b})) {
                        add(a, b, 0);
                    }
                }
            }
        }
    }
}
int bfs() {
    memset(dist, 0x3f, sizeof dist);
    deque<PII> q;
    q.push_back({1, 0});
    dist[1][0]=0;
    while(!q.empty()) {
        auto t=q.front();
        q.pop_front();
        if(st[t.x][t.y]) continue;
        st[t.x][t.y]=1;
```

```

        if(t.x==n*m) return dist[t.x][t.y];
        if(key[t.x]) {
            int state=t.y|key[t.x];
            if(dist[t.x][state]>dist[t.x][t.y]) {
                dist[t.x][state]=dist[t.x][t.y];
                q.push_front({t.x,state});
            }
        }
        for(int i=h[t.x];~i;i=ne[i]) {
            int j=e[i];
            if(w[i] && !(t.y>w[i]-1&1)) continue;
            if(dist[j][t.y]>dist[t.x][t.y]+1) {
                dist[j][t.y]=dist[t.x][t.y]+1;
                q.push_back({j,t.y});
            }
        }
    }
    return -1;
}
void solve() {
    cin>>n>>m>>p>>k;
    memset(h,-1,sizeof h);
    int t=1;
    for(int i=1;i<=n;i++) {
        for(int j=1;j<=m;j++) {
            g[i][j]=t;
            t++;
        }
    }
    for(int i=1;i<=k;i++) {
        int x1,y1,x2,y2,op;
        cin>>x1>>y1>>x2>>y2>>op;
        int a=g[x1][y1], b=g[x2][y2];
        edge.insert({a,b}), edge.insert({b,a});
        if(op) {
            add(a,b,op), add(b,a,op);
        }
    }
    build();
    int cnt;cin>>cnt;
    for(int i=1;i<=cnt;i++) {
        int x,y,op;
        cin>>x>>y>>op;
        key[g[x][y]] |=1<<op-1;
    }
    cout<<bfs() << '\n';
}

```

H6 求最短路径方案数

最短路计数

```
typedef pair<int, int> PII;
const int N=1e5+10, M=4e5+10;
int h[N], e[M], ne[M], idx;
int dist[N];
bool st[N];
int cnt[N];
int n, m;
void add(int a, int b) {
    e[idx]=b, ne[idx]=h[a], h[a]=idx++;
}
int dij() {
    memset(dist, 0x3f, sizeof dist);
    memset(st, 0, sizeof st);
    priority_queue<PII, vector<PII>, greater<PII> > q;
    q.push({0, 1});
    dist[1]=0;
    cnt[1]=1;
    while(!q.empty()) {
        auto t=q.top();
        q.pop();
        int id=t.second;
        if(st[id]) continue;
        st[id]=1;
        for(int i=h[id]; i<ne[id]; i++) {
            int j=e[i];
            if(dist[j]>dist[id]+1) {
                dist[j]=dist[id]+1;
                cnt[j]=cnt[id];
                q.push({dist[j], j});
            } else if(dist[j]==dist[id]+1) {
                cnt[j]+=cnt[id];
                cnt[j]%=100003;
            }
        }
    }
}
void solve() {
    cin>>n>>m;
    memset(h, -1, sizeof h);
    for(int i=1; i<=m; i++) {
        int a, b;
        cin>>a>>b;
        add(a, b);
        add(b, a);
    }
    dij();
    for(int i=1; i<=n; i++) cout<<cnt[i]<<'\n';
}
```

H6 求最短路和次短路及路径条数

观光

```
typedef pair<int, int> PII;
const int N=1010, M=1e4+10;

//状态0表示的是求最小，状态1表示求的是次小
int h[N], e[M], ne[M], w[M], idx;
int cnt[N][2]; //cnt[i][0]表示当前到达节点是i，且求的是0状态下的所有路径中最短路径的边数
int dist[N][2]; //dist[i][0]表示当前到达节点是i，且求的是0状态下的最短路径的值
bool st[N][2]; //与上面同理
int n, m, S, T;
struct ty{//小根堆，重载大于号
    int id, op, dis;//分别是编号，状态，和当前点到起点的最小或次小距离
    bool operator>(const ty &t) const{//从大到小排序
        return dis>t.dis;
    }
};
void add(int a, int b, int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
int dij() {
    memset(dist, 0x3f, sizeof dist);
    memset(cnt, 0, sizeof cnt);
    memset(st, 0, sizeof st);
    priority_queue<ty, vector<ty>, greater<ty>> q;
    q.push({S, 0, 0});
    dist[S][0]=0;
    cnt[S][0]=1;
    while(!q.empty()) {
        auto t=q.top();
        q.pop();
        int id=t.id, op=t.op, dis=t.dis, count=cnt[id][op];
        if(st[id][op]) continue;
        st[id][op]=1;
        for(int i=h[id]; ~i; i=ne[i]) {
            int j=e[i];
            if(dist[j][0]>dis+w[i]) {
                dist[j][1]=dist[j][0], cnt[j][1]=cnt[j][0];
                q.push({j, 1, dist[j][1]});
                dist[j][0]=dis+w[i], cnt[j][0]=count;
                q.push({j, 0, dist[j][0]});
            } else if(dist[j][0]==dis+w[i]) {
                cnt[j][0]+=count;
            } else if(dist[j][1]>dis+w[i]) {
                dist[j][1]=dis+w[i], cnt[j][1]=count;
                q.push({j, 1, dist[j][1]});
            } else if(dist[j][1]==dis+w[i]) {
                cnt[j][1]+=count;
            }
        }
    }
}
```

```

        int ans=cnt[T][0];
        if(dist[T][1]==dist[T][0]+1) ans+=cnt[T][1];
        return ans;
    }

void solve() {
    cin>>n>>m;
    memset(h, -1, sizeof h);
    idx=0;
    for(int i=1; i<=m; i++) {
        int a, b, c;
        cin>>a>>b>>c;
        add(a, b, c);
    }
    cin>>S>>T;
    cout<<dij()<<'\n';
}

```

最小生成树

一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。最小生成树可以用kruskal（克鲁斯卡尔）算法或prim（普里姆）算法求出。

通俗易懂的讲就是最小生成树包含原图的所有节点而只用最少的边和最小的权值距离。因为 n 个节点最少需要 $n-1$ 个边联通，而距离就需要采取某种策略选择恰当的边，且一般都是无向边。

H5 朴素版prim算法

时间复杂度 $O(n^2+m)$, n 表示点数, m 表示边数

prim 算法采用的是一种贪心的策略。

每次将离连通部分的最近的点和点对应的边加入连通部分，连通部分逐渐扩大，最后将整个图连通起来，并且边长之和最小。

```

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 (t != -1)
        {
            res += dist[t];
            for (int j = 1; j <= n; j++)
                if (!st[j] && g[t][j] != INF)
                    dist[j] = min(dist[j], g[t][j]);
        }
    }
    return res;
}

```

```

        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;
}

```

h5 Kruskal算法

时间复杂度 O(mlogm), m表示边数

算法思路:

- 将所有边按照权值的大小进行升序排序，然后从小到大一一判断。
- 如果这个边与之前选择的所有边不会组成回路，就选择这条边；反之，舍去。
- 直到具有 n 个顶点的连通网筛选出来 n-1 条边为止。
- 筛选出来的边和所有的顶点构成此连通网的最小生成树。

判断是否会产生回路的方法为：使用并查集。

- 在初始状态下给各个顶点在不同的集合中。
- 遍历过程的每条边，判断这两个顶点的是否在一个集合中。
- 如果边上的这两个顶点在一个集合中，说明两个顶点已经连通，这条边不要。如果不在一个集合中，则要这条边。

```

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

using namespace std;
const int N = 100010;
int p[N];//保存并查集

struct E{
    int a;
    int b;
    int w;
    bool operator < (const E& rhs){//通过边长进行排序
        return this->w < rhs.w;
    }
};

}edg[N * 2];
int res = 0;

int n, m;
int cnt = 0;
int find(int a){//并查集找祖宗

```

```

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

    void klskr() {
        for(int i = 1; i <= m; i++)//依次尝试加入每条边
        {
            int pa = find(edg[i].a); // a 点所在的集合
            int pb = find(edg[i].b); // b 点所在的集合
            if(pa != pb){//如果 a b 不在一个集合中
                res += edg[i].w;//a b 之间这条边要
                p[pa] = pb;// 合并a b
                cnt++; // 保留的边数量+1
            }
        }
    }

    int main()
    {

        cin >> n >> m;
        for(int i = 1; i <= n; i++) p[i] = i;//初始化并查集
        for(int i = 1; i <= m; i++) {//读入每条边
            int a, b , c;
            cin >> a >> b >>c;
            edg[i] = {a, b, c};
        }

        sort(edg + 1, edg + m + 1);//按边长排序
        klskr();
        if(cnt < n - 1) {//如果保留的边小于点数-1，则不能连通
            cout<< "impossible";
            return 0;
        }
        cout << res;
        return 0;
    }
}

```

H6 kruskal+虚拟源点

新的开始

```

//0号点为虚拟源点
const int N=410;
int p[N];
struct ty{
    int a,b,w;
    bool operator<(const ty& t) {
        return w<t.w;
    }
} e[N*N];
int n;
int find(int x){
    if(x!=p[x]) p[x]=find(p[x]);
    return p[x];
}

```

```

    }

void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) p[i]=i;
    int cnt=0;
    for(int i=1;i<=n;i++) {
        int x;cin>>x;
        e[++cnt]={0, i, x};
    }
    for(int i=1;i<=n;i++) {
        for(int j=1;j<=n;j++) {
            int x;cin>>x;
            e[++cnt]={i, j, x};
        }
    }
    sort(e+1,e+cnt+1);
    int ans=0;
    for(int i=1;i<=cnt;i++) {
        int a=e[i].a,b=e[i].b,w=e[i].w;
        a=find(a), b=find(b);
        if(a!=b) {
            p[a]=b;
            ans+=w;
        }
    }
    cout<<ans<<'\n';
}

```

H6 次小生成树

所谓严格就是该次小生成树的总边权严格大于最小生成树（以下用MST称呼）。非严格的就是边权和MST相同。



```

/*最小生成树次小生成树
将最小生成树称为为MST
次小生成树在最小生成树的邻集中。（最小生成树变一个边）
求最小生成树，统计标记每条边是否是树边；同时把最小生成树建立，权值之和为sum
预处理生成树中任意两点间的边权最大值dist1[a][b] 和长度次大dist2[a][b] （树中两点路径唯一，dfs）
依次枚举所有非MST边t，边t连接a, b, 权为w。显然a, b在MST中。
    尝试用t替换a-b的路径中最大的一条边A。t的权w >= A。（如果w < A, 直接换边就能得到更小的生成树，矛盾了）
        如果w > A, 替换后总权值是sum + w - dist1[a][b]
        否则 w = A , 不能替换，会得到非严格次小生成树（权值和MST相等）
        w = A, w > 次大值B 替换后总权值是sum + w - dist2[a][b]
*/
#include <iostream>

```

```

#include <cstring>
#include <algorithm>
using namespace std;
const int N = 510, M = 10010;
typedef long long LL;
struct Edge{
    int a, b, w;
    bool f = false;
    bool operator < (const Edge & A) const{
        return w < A.w;
    }
} edge[M];
int h[N], e[N * 2], ne[N * 2], w[N * 2], idx; // 无向树 2 * N
int cnt;
int n, m;
int dist1[N][N], dist2[N][N];// 最小和次小
int p[N];
void add(int a,int b,int c){
    e[idx] = b,w[idx] = c,ne[idx] = h[a],h[a] = idx++;
}
int find(int x){
    return x == p[x] ? x : p[x] = find(p[x]);
}
//dfs无向图技巧：记录父节点防止回走
void dfs(int u, int fa, int maxd1, int maxd2, int d1[], int d2[]){
    d1[u] = maxd1, d2[u] = maxd2;
    for (int i = h[u]; ~i; i = ne[i]){
        int j = e[i];
        if (j != fa){ //不往回搜
            int td1 = maxd1, td2 = maxd2;
            if (w[i] > td1) td2 = td1, td1 = w[i];
            else if (w[i] < td1 && w[i] > td2) td2 = w[i];
            dfs(j, u, td1, td2, d1, d2);
        }
    }
}
int main()
{
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for(int i = 0; i < m; ++ i){
        int a, b, w;
        cin >> a >> b >> w;
        edge[i] = {a, b, w};
    }
    sort(edge, edge + m);
    for(int i = 1; i <= n; ++ i)p[i] = i;
    LL sum = 0;
    for(int i = 0; i < m; ++ i){
        int a = edge[i].a, b = edge[i].b, w = edge[i].w;
        int pa = find(a), pb = find(b);
        if (pa != pb)

```

```

    {
        p[pa] = pb;
        sum += w;
        add(a, b, w), add(b, a, w); //! 合并集合，但加边是节点之间加边
        edge[i].f = true;
    }
}

for(int i = 1; i <= n; ++ i)dfs(i, -1, -1e9, -1e9, dist1[i], dist2[i]); // 生成树内搜索最长路
LL res = 1e18;
for(int i = 0; i < m; ++ i){
    int a = edge[i].a, b = edge[i].b, w = edge[i].w;
    if(!edge[i].f){ // 遍历每条外部边尝试替换
        LL t = 1e18;
        if(w > dist1[a][b]){
            t = sum + w - dist1[a][b];
        }else if(w > dist2[a][b]){ // w不是大于就是等于
            t = sum + w - dist2[a][b];
        }
        res = min(res, t);
    }
}
cout << res << endl;
return 0;
}

```

如果是非严格次小生成树，那么不需要dist[2]数组，即不需要长度次大值，非严格次小生成树的边权和可以和最小生成树的边权和相等

差分约束



差分约束

(1) 求不等式组的可行解

源点需要满足的条件：从源点出发，一定可以走到所有的边。

步骤：

- [1] 先将每个不等式 $x_i \leq x_j + c_k$ 转化成一条从 x_j 走到 x_i ，长度为 c_k 的一条边
- [2] 找一个超级源点，使得该源点一定可以遍历到所有边
- [3] 从源点求一遍单源最短路

结果1：如果存在负环，则原不等式组一定无解

结果2：如果没有负环，则 $dist[i]$ 就是原不等式组的一个可行解

(2) 如何求最大值或者最小值，这里的最值指的是每个变量的最值

结论：如果求的是最小值，则应该求最长路；如果求的是最大值，则应该求最短路；

问题：如何转化 $x_i \leq c$ ，其中 c 是一个常数，这类的不等式

方法：建立一个超级源点，0，然后建立 $0 \rightarrow i$ ，长度是 c 的边即可。

以求 x_i 的最大值为例：求所有从 x_i 出发，构成的不等式链 $x_i \leq x_j + c_1 \leq x_k + c_2 + c_1 \leq \dots \leq c_1 + c_2 + \dots$ 所计算出的上界，最终 x_i 的最大值等于所有上界的最小值。

向最大值，则求最短路，若有负环，则无解，向最小值，求最长路，若有正环，则无解

糖果

```
/*
本题
1 A = B <=> A≥B B≥A
2 A < B <=> B≥A+1
3 A≥B <=> A≥B
4 A > B <=> A≥B+1
5 B≥A <=> B≥A
x≥1
} x≥x0+1(超级源点x0=0)
x0=1
```

举例 $x[i] \geq 5$

```
x[i] ≥ 2
x[i] ≥ 3
min(x[i]) = 5
```

总共最小值 = $\min(x[i]$ 最小值) for i in range(n)

= 求 $x[i]$ 所有下界的最大值
= 求所有从0→i的路径和的最大值
= 最长路求 $dist[i]$

$0 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow \dots \rightarrow i$

c1 c3 c5 ci-1

```
x[1] ≥ x[0] + c[1]
x[3] ≥ x[1] + c[3]
x[5] ≥ x[3] + c[5]
```

...

$x[i] \geq x[i-1] + c[i-1]$

则

```
x[i] ≥ x[i-1] + c[i]
≥ x[i-3] + c[i-3] + c[i]
...
≥ x[0] + c[1] + c[3] + c[i-3] + c[i-1]
```

★ 可以发现 $\sum c[i]$ 就是从0→i的一条路径的长度

那么 求 $x[i]$ 最小值

\Leftrightarrow

求所有下界的最大值

\Leftrightarrow

求所有从0→i的路径和的最大值

\Leftrightarrow

最长路求 $dist[i]$

即：

```
if(d[j]<d[i]+w[i][j])
    d[j] = d[i] + w[i][j]
```

建立边数

最坏情况 $A = B \Leftrightarrow A \geq B B \geq A$ 2条

+超级源点和所有点xi建边

= $3n$

*/

```
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 cnt[N];//统计到当前点总共有多少条边了
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()
{
    //有时用queue判断正环或者负环会超时，用栈stack则能够大量减小运行时间，减少多余点的遍历的时间。
    //一般还是用queue
    stack<int> q;
    q.push(0);
    st[0] = true;
    //最长路 dist[j] < dist[t] + w[i] 初始化为-INF
    memset(dist, -0x3f, sizeof dist);
    dist[0] = 0;
    while(q.size())
    {
        int t = q.top();
        q.pop();
        st[t] = false;//不在栈中 状态=false
        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] < dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n + 1) return true;//有正环 无解
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }
    return false;
}
int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
    while (m -- )

```

```

int x, a, b;
cin >> x >> a >> b;
if (x == 1) add(b, a, 0), add(a, b, 0); //A=B
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
}

//每个同学都要分到糖果 x[i]≥1
//超级源点0 x[i] ≥ x[0]+1 <=> x[i] ≥ 1
for (int i = 1; i <= n; i++) add(0, i, 1);
if (spfa()) cout << "-1" << endl;
else
{
    LL res = 0;
    for (int i = 1; i <= n; i++) res += dist[i];
    cout << res << endl;
}
return 0;
}

```

最近公共祖先 (LCA)



H5 向上标记法(不常用)

方法1 向上标记法 $O(n)$

步骤

- 1 先从点1往上走到根节点, 走过的点都标记
- 2 再从点2往上走, 碰到的第一个带标记的点就是最近公共祖先

*/

H5 倍增法（在线求lca，边读入边求）

/*

预处理(nlogn) + 查询(logn)

★关键是理解二进制拼凑 在这里是怎么样体现的

即 x, y 从同一高度同时起跳后, 在 $f[x][0] \neq f[y][0]$ 的约束下 我们能跳的最多的步数跳完后 x, y 就达到了 LCA的下面一层

假定我们知道 x, y 出发点为第1层

LCA下一层为第12层

那么最多能跳的步数 $t = 12 - 1 = 11 = (1011)_2 =$ 最多能跳 $2^3 + 2^2 + 2^0$ 步

所以我们就通过从大到小枚举 k 使得我们刚好跳11步而不能跳超过12步

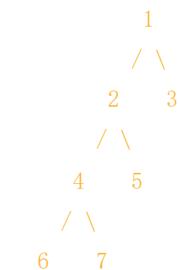
但实际上我们并不知道要跳11步, 所以我们可以通过 $f[x][0] \neq f[y][0]$ 的约束来实现

即 $f[x][\text{总共} \geq 12 \text{步}] = f[y][\text{总共} \geq 12 \text{步}]$ 那就不跳(不拼凑 2^k)

$f[x][\text{总共} < 12 \text{步}] \neq f[y][\text{总共} < 12 \text{步}]$ 那就跳(拼凑 2^k)

预处理出每个点向上走 2^k 步的父亲是谁

$f[i][j]$ 从 i 开始向上走 2^j 步所能走到的节点 $0 \leq j \leq \log n$



$f[6][0] = 4$

$f[6][1] = 2$

$f[6][2] = \text{空集}$

$j=0 f[i][j] = i$ 的父节点

$j>0 f[i][j-1]$

$i \rightarrow mid \rightarrow t$

$2^{j-1} 2^{j-1}$

$f[i][j-1] f[i][j]$

$mid = f[i][j-1]$

$t = f[i][j]$

则 $f[i][j] = f[mid][j-1] = f[f[i][j-1]][j-1]$

$\text{depth}[i]$ 表示深度/层数



步骤1 把两个点跳到同一层 把 x 跳到和 y 同一层

二进制拼凑 $2^0 \sim 2^k = t$

举例 $2^0 \sim 2^4$ 11

1 2 4 8 16 11

二进制

16>11 0
8<11 t = 11-8 = 3 1
4>3 0
2<3 t = 3-2 = 1 1
1>=1 t = 1 1

二进制 (1011)2

depth(x) - depth(y)

从x跳到y

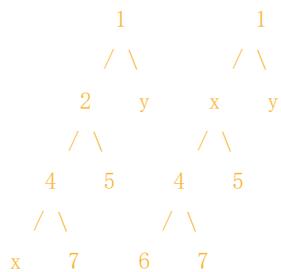
从x跳 2^k 步后的点的深度 $\text{depth}(f[x][k]) \geq \text{depth}(y)$ 时 就可以继续跳

步骤2 在 $\text{depth}(x) == \text{depth}(y)$ 后 一起往上跳 2^k (for k in [log(n), 1])

情况1 x==y 则该点就是x和y的最近公共祖先

情况2 x!=y 即他俩同层但不同

则继续让两个点同时往上跳 一直跳到它们的最近公共祖先的下一层



why 最近公共祖先的下一层 not 最近公共祖先?

方便判断

假如 $f[x][k] == f[y][k] \Leftrightarrow f[x][k]$ or $f[y][k]$ 是x和y的一个公共祖先 但不一定是最近的
举个栗子

此时 $f[x][1] == f[y][1] =$ 节点2 是x和y的一个公共祖先 但不是最近公共祖先4

, 但由于我们是从大到小拼凑的, 假如拼凑终止条件为 $f[x][k] == f[y][k]$

, 则此时会停在公共祖先2而非最近公共祖先4



步骤一 x先到y同一层 $f[x][0] != f[y][0]$ 且把k能填1的都填完了

加上约束 $f[x][k] != f[y][k]$

思考一下为啥: 因为第一个出现 $f[x][k] == f[y][k]$ 的节点只是公共祖先却不能保证是最近公共祖先
所以只要 $f[x][k] != f[y][k]$

那么 x y就还没跳到过最近公共祖先, 而在其下面层

从大往小枚举k

枚举过程中

只要 $f[x][k] != f[y][k]$

那么 x y就还没跳到过最近公共祖先, 而在其下面层, 则x, y更新为 $f[x][k]$ $f[y][k]$

这个过程中 $f[x][k] == f[y][k]$ 时

不执行跳 2^k 步跳到 $f[y][k]$ 的操作

直到k枚举完为止

枚举完之后就走到了最近公共祖先的下一层

二进制拼凑 $2^0 \sim 2^k - t$

这里的k最大为logn

t为 x和y达到同一高度(depth(起始)) - 最近公共祖先下一层深度-1(depth(祖先)-1)

举例 $2^0 \sim 2^4$ 2

1 2 4 8 16 2

二进制

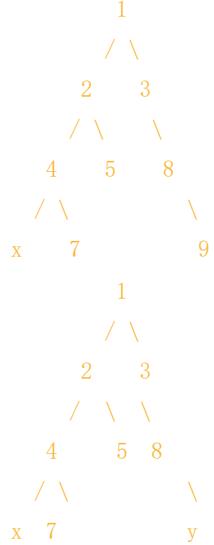
16>2 0

8>2 0

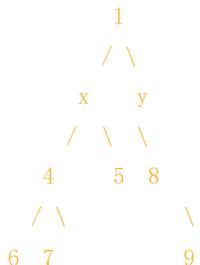
4>2 0

2=2 t = 2-2 = 0 1

1>0 0



在这个例子里 k=4 t=2 (x, y出发层)-2 (LCA下一层) [通过f[x][k] != f[y][k]约束]



此时它们的最近公共祖先就是x or y往上跳一步

即LCA = f[x][0] or f[y][0]

*/

祖孙询问

```
typedef pair<int, int> PII;
const int N=4e4+10, M=N*2;
int h[N], e[M], ne[M], idx;
int depth[N];
int fa[N][20];//往上跳2^k步后的父亲节点
int n, m;
void add(int a, int b){
    e[idx]=b, ne[idx]=h[a], h[a]=idx++;
}
void bfs(int root){//宽搜不容易因为递归层数过多爆栈
    memset(depth, 0x3f, sizeof depth);
    queue<int> q;
    q.push(root);
```

```

// 哨兵depth[0] = 0: 如果从i开始跳2^j步会跳过根节点
// fa[fa[j][k-1]][k-1] = 0
// 那么fa[i][j] = 0 depth[fa[i][j]] = depth[0] = 0
depth[0]=0, depth[root]=1;
while(!q.empty()) {
    auto t=q.front();
    q.pop();
    for(int i=h[t];~i;i=ne[i]) {
        int j=e[i];
        if(depth[j]>depth[t]+1) //说明j还没被搜索过
            depth[j]=depth[t]+1;
        q.push(j); //把第depth[j]层的j加进队列
        fa[j][0] = t; //j往上跳2^0步后就是t
        /*
        i → mid → t
        2^j-1 2^j-1
        f[i][j-1] f[i][j]
        mid = f[i][j-1]
        t = f[i][j]
        则f[i][j] = f[mid][j-1] = f[f[i][j-1]][j-1]
        */
        for(int i=1;i<=15;i++) {
            fa[j][i]=fa[fa[j][i-1]][i-1];
        }
        /*
        举个例子理解超过根节点是怎么超过的
        因为我们没有对根节点fa[1][0]赋值,那么fa[1][0] = 0;
        1
        / \
        2   3
        fa[1][0] = 0;
        fa[2][0] = 1;
        fa[2][1] = fa[fa[2][0]][0] = fa[1][0] = 0;
        */
    }
}
int lca(int a, int b)
{
    // 为方便处理 当a在b上面时 把a b 互换
    if (depth[a] < depth[b]) swap(a, b);
    //把深度更深的a往上跳到b
    for (int k = 15; k >= 0; k -- )
        //当a跳完2^k依然在b下面 我们就一直跳
        //二进制拼凑法
        //这里因为
        if (depth[fa[a][k]] >= depth[b])
            a = fa[a][k];
    //如果跳到了b
    if (a == b) return a;
    //a,b同层但不同节点
}

```

```

        for (int k = 15; k >= 0; k -- )
            // 假如a, b都跳出根节点, fa[a][k]==fa[b][k]==0 不符合更新条件
            if (fa[a][k] != fa[b][k])
            {
                a = fa[a][k];
                b = fa[b][k];
            }
        //循环结束 到达lca下一层
        //lca(a, b) = 再往上跳1步即可
        return fa[a][0];
    }

void solve() {
    cin>>n;
    memset(h, -1, sizeof h);
    int root=0;
    for(int i=1;i<=n;i++) {
        int a,b;
        cin>>a>>b;
        if(b== -1) {
            root=a;
        } else{
            add(a, b), add(b, a);
        }
    }
    bfs(root);
    cin>>m;
    for(int i=1;i<=m;i++) {
        int a,b;
        cin>>a>>b;
        int t=lca(a, b);
        if(t==a){
            cout<<1<<' \n';
        } else if(t==b){
            cout<<2<<' \n';
        } else{
            cout<<0<<' \n';
        }
    }
}

```

h5 Tarjan（离线求lca，全部读入再求）

距离

```

typedef pair<int, int> PII;
const int N=1e4+10, M=2e4+10;
int h[N], e[M], ne[M], w[M], idx;
int dist[N];
//每个点和1号点的距离

```

```

vector<PII> v[N];
//把询问存下来
// v[i][first][second] first存查询距离i的另外一个点j, second存查询编号idx
int st[N];
int p[N];
int n, m;
int ans[M];
void add(int a, int b, int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
int find(int x) {
    if(p[x]!=x) p[x]=find(p[x]);
    return p[x];
}
void dfs(int u, int fa) {
    for(int i=h[u];~i;i=ne[i]) {
        int j=e[i];
        if(j==fa) continue;
        dist[j]=dist[u]+w[i];
        dfs(j, u);
    }
}
void tarjan(int u) {
    st[u]=1;//当前路径点标记为1
    // u这条路上的根节点的左下的点用并查集合并到根节点
    for(int i=h[u];~i;i=ne[i]) {
        int j=e[i];
        if(!st[j]){
            tarjan(j);//往左下搜
            p[j] = u;//从左下回溯后把左下的点合并到根节点
        }
    }
    // 对于当前点u 搜索所有和u有关的点
    for(auto t:v[u]) {
        int y=t.first, id=t.second;
        if(st[y]==2)//如果查询的这个点已经是左下的点(已经搜索过且回溯过, 标记为2)
            int x=find(y);//y的根节点
            // x到y的距离 = d[x]+d[y] - 2*d[1ca]
            ans[id]=dist[y]+dist[u]-2*dist[x];//第id次查询的结果 ans[id]
    }
}
//点u已经搜索完且要回溯了 就把st[u]标记为2
st[u]=2;
}
void solve() {
    cin>>n>>m;
    memset(h, -1, sizeof h);
    for(int i=1;i<=n-1;i++) {
        int a, b, c;
        cin>>a>>b>>c;
        add(a, b, c), add(b, a, c);
    }
}

```

```

//存下询问
for(int i=1;i<=m;i++) {
    int a,b;
    cin>>a>>b;
    if(a!=b) {
        v[a].push_back({b,i});
        v[b].push_back({a,i});
    }
}
for(int i=1;i<=n;i++) p[i]=i;
dfs(1,-1);
tarjan(1);
for(int i=1;i<=m;i++) cout<<ans[i]<<'\n';//把每次询问的答案输出
}

```

H5 倍增lca求次小生成树 (nlogn)

```

/*
★ 非树边w的值域是一定 $\geq dist_1$  否则在当 $w < dist_1$ , 则之前kruskal求最小生成树的时候把w替换dist1连接a和b就得到一个更小的生成树(且依然能是一个生成树--原因如下图)了 与kruskal得到的是最小生成树矛盾
    dist1
    -c - d-
    a     b      →      a     b
          -           -
          w           w

```

解法0

AcWing 1148. 秘密的牛奶运输 暴力枚举

解法1

lca倍增O(logn)求新加入非树边边的两点a, b间的最大边和最小边

定理:对于一张无向图, 如果存在最小生成树和次小生成树, 那么对于任何一颗最小生成树都存在一颗次小生成树

使得这两棵树只有一条边不同



那么我们就是要在现有的最小生成树中找一条非树边w去替换w[i]

则对于每一条非树边w, 变换后的边权和 $=sum+w-w[i]$ 则我们想要边权和越小, 则对应替换的w[i]越大

即我们要做的就是找这条路径上的最大边权替换

为了防止 $w == \max(w[i] \text{ for } i \text{ in path})$ 替换后w[i]依然==w 导致不是严格最小生成树

在这种情况下 我们用w替换这条路上的次大边

所以用d1 d2存最大边和次大边

2 预处理每个点i跳2^j后的父节点f[i][j]





在x和y向 1ca[x][y]跳的过程中维护各自路径中的最大值 $d1[x \rightarrow 1ca] \setminus d1[y \rightarrow 1ca]$ 和次大值 $d2[x \rightarrow 1ca] \setminus d2[y \rightarrow 1ca]$

路径

最大值 $d1$ o o o
| | |

次大值 $d2$ o o o
| | |

假设最大值为. 则次大值从ci中找

ci . ci
| | |
o ci o
| | |

$[x \rightarrow y]$ 的最大值 = $\max(d1[i] \text{ for } i \text{ in path})$

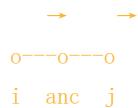
次大值 = 次大($\max(d1[i], d2[i]) \text{ for } i \text{ in path}$)

3 预处理:

每个点i跳 2^j j路径上的最大边权 $d1[i][j]$

每个点i跳 2^j j路径上的次大边权 $d2[i][j]$

$d1[i][j]$ 跳 2^j j次



$d1[i, j-1], d2[i, j-1] \leftarrow d1[anc, j-1], d2[anc, j-1]$

*/

```

typedef long long LL;
const int N = 100010, M = 300010, INF = 0x3f3f3f3f;
int n, m;
struct Edge
{
    int a, b, w;
    bool used;
    bool operator< (const Edge &t) const
    {
        return w < t.w;
    }
} edge[M];
int p[N];
int h[N], e[M], w[M], ne[M], idx;
int depth[N], fa[N][17], d1[N][17], d2[N][17]; //log2(1e5)=16 d1最大边 d2次大边
void add(int a, int b, int c)
{
    e[idx] = b, ne[idx] = h[a], w[idx] = c, h[a] = idx++;
}
  
```

```

int find(int x)
{
    if(x!=p[x]) p[x]= find(p[x]);
    return p[x];
}
LL kruskal()
{
    for (int i = 1; i <= n; i++) p[i] = i;
    sort(edge, edge + m);
    LL res = 0;
    for (int i = 0; i < m; i++)
    {
        int a = find(edge[i].a), b = find(edge[i].b), w = edge[i].w;
        if (a != b)
        {
            p[a] = b;
            res += w;
            edge[i].used = true;
        }
    }
}

return res;
}
void build()
{
    memset(h, -1, sizeof h);
    for(int i = 0;i<m;i++)
    {
        //将最小生成树建出来
        if(edge[i].used)
        {
            int a = edge[i].a,b = edge[i].b,w = edge[i].w;
            add(a,b,w),add(b,a,w);
        }
    }
}
void bfs()
{
    memset(depth, 0x3f, sizeof depth);
    depth[0] = 0, depth[1] = 1;//哨兵0 根节点1
    queue<int> q;
    q.push(1);
    while(q.size())
    {
        int t = q.front();
        q.pop();
        for(int i = h[t];~i;i=ne[i])
        {
            int j = e[i];
            // j没有被遍历过
            if(depth[j]>depth[t]+1)
            {

```

```

        depth[j] = depth[t]+1;
        q.push(j);
        fa[j][0] = t;
        d1[j][0] = w[i], d2[j][0] = -INF;
        for(int k = 1;k<=16;k++)
        {
        /*
            →      →
            o---o---o
            j   anc
            d1[i, k-1], d2[i, k-1]  d1[anc, k-1], d2[anc, k-1]
        */
        int anc = fa[j][k - 1];
        fa[j][k] = fa[anc][k - 1];
        int distance[4] = {d1[j][k - 1], d2[j][k - 1], d1[anc][k - 1],
        d2[anc][k - 1]};
        //初始化d1[j][k]和d2[j][k]
        d1[j][k] = d2[j][k] = -INF;
        for (int u = 0; u < 4; u++)
        {
            int d = distance[u];
            // 更新最大值d1和次大值d2
            if (d > d1[j][k]) d2[j][k] = d1[j][k], d1[j][k] = d;
            // 严格次大值
            else if (d < d1[j][k] && d > d2[j][k]) d2[j][k] = d;
        }
    }
}
}

// lca求出a, b之间的最大边权与次大边权
int lca(int a, int b, int w)
{
    static int distance[N * 2];//存储跳的过程中经过的边的权值
    int cnt = 0;
    // a和b中取深度更深的作为a先跳
    if (depth[a] < depth[b]) swap(a, b);
    for (int k = 16; k >= 0; k--)
    // 如果a 跳2^k后的深度比b深度大 则a继续跳
    // 直到两者深度相同 depth[a] == depth[b]
        if (depth[fa[a][k]] >= depth[b])
    {
        distance[cnt ++ ] = d1[a][k];
        distance[cnt ++ ] = d2[a][k];
        a = fa[a][k];
    }
    // 如果a和b深度相同 但此时不是同一个点 两个同时继续向上跳
    if (a != b)
    {
        for (int k = 16; k >= 0; k--)
            if (fa[a][k] != fa[b][k])

```

```

    {
        distance[cnt ++ ] = d1[a][k];
        distance[cnt ++ ] = d2[a][k];
        distance[cnt ++ ] = d1[b][k];
        distance[cnt ++ ] = d2[b][k];
        a = fa[a][k], b = fa[b][k];
    }
    // 此时a和b到lca下同一层 所以还要各跳1步=跳2^0步
    distance[cnt ++ ] = d1[a][0];
    distance[cnt ++ ] = d1[b][0];
}

// 找a,b两点距离的最大值dist1和次大值dist2
int dist1 = -INF, dist2 = -INF;
for (int i = 0; i < cnt; i ++ )
{
    int d = distance[i];
    if (d > dist1) dist2 = dist1, dist1 = d;
    else if (d != dist1 && d > dist2) dist2 = d;
}
// ★ dist1和dist2是a和b之间的最大边权和次大边权 所以可以用w替换而仍然保持生成树(包含所有节点)
// 因为加入w这条边 原来的树会形成环

// 删除环中边权最大的边 (如果最大的边和加入的边相等, 那么删去次大边)。
// 如果w>这条路的最大边 w替换dist1
if (w > dist1) return w - dist1;
// 否则w==dist1 w替换dist2
if (w > dist2) return w - dist2;
return INF;
}

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 0; i < m; i ++ )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        edge[i] = {a, b, c};
    }
    // kruskal建最小树(把用到的边标记)
    LL sum = kruskal();
    // 对标记的边建图
    build();
    bfs();
    LL res = 1e18;
    //从前往后枚举非树边
    for (int i = 0; i < m; i ++ )
        if (!edge[i].used)
    {
        int a = edge[i].a, b = edge[i].b, w = edge[i].w;
        // lca(a,b,w) 返回用w替换w[i] 的差值 = w-w[i]
    }
}

```

```

        res = min(res, sum + lca(a, b, w));
    }
    printf("%lld\n", res);
    return 0;
}

```

H5 lca倍增+树上差分

暗之连锁

题意:对一棵树，有树边和非树边，只能砍两刀，一刀树边，一刀非树边

/*

1 没有非树边 情况 很多可选项

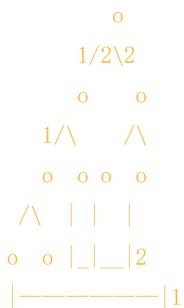


2 有1条非树边 则有环->环1要砍树上一条边+非树边



3 有2条非树边 则有环->环1和环2都要砍树上一条边+非树边

★ 此时图中数目表示由非树边i构成的环上的树边编号



1 先枚举每个非树边i

2 对非树边i构成的环上的每一个树边累加一个1 ->表示该树边砍完后还需要砍多少个非树边

★ 此时图中数目表示该树边砍完后还需要砍c个非树边

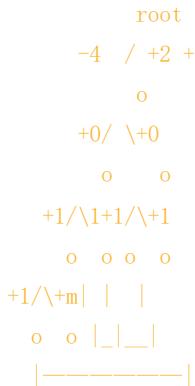


|-----|

★那么

c = 0 则第二刀可以随便砍	res+=m
c = 1 则第二刀必须砍对应的非树边	res+=1
c > 1 则起码需要3刀(≥ 2 刀非树边+当前边)	res+=0

最终方案数通过dfs回溯得到



问题来了，怎么把 $x \rightarrow y$ 上的每一条边+c?

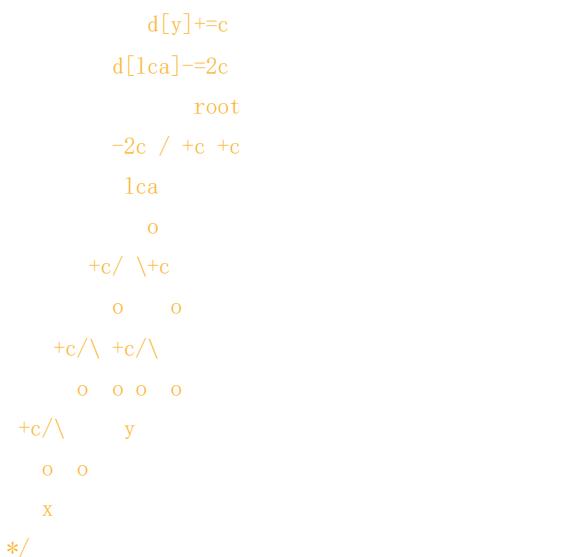
$d[x]$ $d[y]$ 会对它们到根节点上的每一条边都+c

$d[lca]$ 会对它们到根节点上的每一条边都-2c

▲ 所以最后只改变了 $x \rightarrow lca[x, y]$

$y \rightarrow lca[x, y]$ 路径上的边权

树上差分 $d[x] += c$



```
const int N = 100010, M = N * 2;
```

```

int n, m;
int h[N], e[M], ne[M], idx;
int depth[N], fa[N][17];
int d[N]; //存储每个点差分数
int ans;
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
// 求lca
void bfs()

```

```

{
    memset(depth, 0x3f, sizeof depth);
    queue<int> q;
    depth[0] = 0, depth[1] = 1;
    q.push(1);
    while(q.size())
    {
        int t = q.front();
        q.pop();
        for(int i = h[t]; ~i; i=ne[i])
        {
            int j = e[i];
            if(depth[j]>depth[t]+1)
            {
                depth[j] = depth[t]+1;
                q.push(j);
                fa[j][0] = t;
                // +2^k-1      +2^k-1
                // j → fa[j][k-1] → fa[j][k]
                for(int k = 1; k<=16; k++)
                {
                    fa[j][k] = fa[fa[j][k-1]][k-1];
                }
            }
        }
    }

    int lca(int a, int b)
    {
        // 从更低的a开始
        if (depth[a] < depth[b]) swap(a, b);
        // 把a和b提到同一个高度
        for (int k = 16; k >= 0; k --)
            if (depth[fa[a][k]] >= depth[b])
                a = fa[a][k];
        if (a == b) return a;
        // 如果 a, b不是同一个点 两个一起往上跳
        for (int k = 16; k >= 0; k --)
            if (fa[a][k] != fa[b][k])
            {
                a = fa[a][k];
                b = fa[b][k];
            }
        // lca 倍增最终停下来的位置是lca下一层 所以还要跳一步
        return fa[a][0];
    }

    // dfs 返回每一棵子树的和
    int dfs(int u, int father)
    {
        // 遍历以u为根节点的子树j的和
        int res = d[u];

```

```

        for (int i = h[u]; ~i; i = ne[i])
        {
            int j = e[i];
            if (j != father)
            {
                // 边t→j 破掉后的方案 s
                int s = dfs(j, u);
                // 如果s=0 则随便破
                if (s == 0) ans += m;
                // 如果s=1 则只能破对应的非树边
                else if (s == 1) ans++;
                // 子节点j的差分向上加给/传给 节点u
                res += s;
            }
        }
        // 如果没有子节点 即叶子节点 直接返回d[node]
        return res;
    }

int main()
{
    scanf("%d%d", &n, &m);
    memset(h, -1, sizeof h);
    //读入树边
    for (int i = 0; i < n - 1; i++)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b), add(b, a);
    }

    bfs();
    // 读入附加边==非树边
    for (int i = 0; i < m; i++)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        int p = lca(a, b);
        //树上差分, 最后用dfs求和
        d[a]++, d[b]++;
        d[p] -= 2;
    }
    dfs(1, -1);
    printf("%d\n", ans);
    return 0;
}

```

有向图的强联通分量 (SCC) (tarjan缩点+拓扑图)

h5 tarjan+拓扑图

/*

有向图

连通分量:对于分量中任意两点 u, v 必然可以从 u 走到 v 且从 v 走到 u , 即形成一个环

强连通分量:极大连通分量

有向图 \rightarrow 有向无环图(DAG)

缩点(将所有连通分量缩成一个点)

缩点举例:

$o \rightarrow o \rightarrow o \rightarrow o$

$\uparrow \downarrow$

$o \rightarrow o \rightarrow o \rightarrow o$

中间的环缩成一个点

$o \quad o$

$\searrow \nearrow$

o

$\nearrow \searrow$

$o \quad o$

应用:

求最短/最长路 递推

求强连通分量:dfs

1 树枝边(x, y)



2 前向边(x, y)

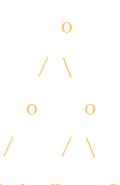


3 后向边



4横插边(往已经搜过的路径上的点继续深搜)

因为我们是从左往右搜的 所以一般是x左边分支上的点



如果往x右边边分支上的点搜 则属于树枝边

强连通分量:

情况1:

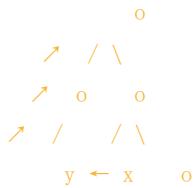
x存在后向边指向祖先结点y 直接构成环



情况2:

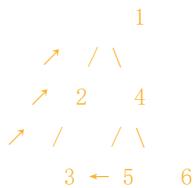
x存在横插边指向的点y有指向x和y的公共祖先节点及以上的点的边

再通过根节点往下走到x间接构成环



Tarjan 算法求强连通分量

引入 时间戳(按dfs 回溯的顺序标记)



标记上时间后:

dfn[u]dfs遍历到u的时间(如上图中的数字)

low[u]从u开始走所能遍历到的最长时间戳(上图中1, 2, 3, 4, 5都是一个环/强连通分量中的

即dfn[1]=low[1]=low[2]=low[3]=low[4]=low[5])

--即u如果在强连通分量, 其所指向的层数最高的点

u是其所在的强连通分量的最高点 (上图中dfn[1]=low[1] dfn[6]=low[6])

\Leftrightarrow

dfn[u] == low[u]

树枝边(x, y) 中dfn[y]>dfn[x] low[u]>dfn[u]

前向边(x, y) 中dfn[y]>dfn[x] low[u]>dfn[u]

后向边(x, y) 中dfn[x]>dfn[y] 后向边的终点dfn[u] == low[u]

横插边(x, y) 中dfn[x]>dfn[y]

缩点

```
for i=1;i<=n;i++  
    for i的所有邻点j  
        if i和j不在同一scc中:  
            加一条新边id[i]→id[j]
```

缩点操作后变成有向无环图

就能做topo排序了(此时连通分量编号id[]递减的顺序就是topo序了)

因为我们++scc_cnt是在dfs完节点i的子节点j后才判断low[u]==dfn[u]后才加的

那么子节点j如果是强连通分量 scc_idx[j]一定小于scc_idx[i]

*/

受欢迎的牛

```

/*
本题
当一个强连通的出度为0，则该强连通分量中的所有点都被其他强连通分量的牛欢迎
但假如存在两及以上个出度=0的牛(强连通分量) 则必然有一头牛(强连通分量)不被所有牛欢迎
见下图最右边两个强连通分量

o->o->o
      ↑
      o->o
*/
const int N=1e4+10, M=5e4+10;
int h[N], e[M], ne[M], idx;
int id[N], sz[N], dfn[N], low[N], dout[N]; //sz数组表示每个强连通分的节点个数
stack<int> q;
int scc_cnt, timestamp;
int n, m;
bool st[N];
void add(int a, int b) {
    e[idx]=b, ne[idx]=h[a], h[a]=idx++;
}
void tarjan(int u) {
    //u的时间戳
    dfn[u]=low[u]=++timestamp;
    //把当前点加到栈中 当前点在栈中
    q.push(u), st[u]=1;
    for(int i=h[u]; ~i; i=ne[i]) {
        int j=e[i];
        //j点未被遍历过
        if(!dfn[j]){//继续dfs 遍历j
            tarjan(j);
            //j也许存在反向边到达比u还高的层，所以用j能到的最小dfn序(最高点)更新u能达到的(最小dfn序)最高点
            low[u]=min(low[u], low[j]);
            //j点在栈中 说明还没出栈 是dfs序比当前点u小的
            //则其 1要么是横插边(左边分支的点)
            //
            //          o
            //          / \
            //          j ← u
            // 2要么是u的祖宗节点
            //
            //          j
            //          ↗
            //          u
            // 两种情况u的dfs序都比j大 所以用dfn[j]更新low[u]
        } else if(st[j]){
            low[u]=min(low[u], dfn[j]);
        }
        //栈代表当前未被搜完的强连通分量的所有点
    }
    // ★
    // 解释一下为什么tarjan完是逆dfs序
    // 假设这里是最高的根节点fa
    // 上面几行中 fa的儿子节点j都已经在它们的递归中走完了下面9行代码
    // 其中就包括 ++scc_cnt
}

```

```

// 即递归回溯到高层节点的时候 子节点的scc都求完了
// 节点越高 scc_id越大
// 在我们后面想求链路dp的时候又得从更高层往下
// 所以得for(int i=scc_cnt(根节点所在的scc);i;i--)开始

// 所以当遍历完u的所有能到的点后 发现u最高能到的点是自己
// 1 则u为强连通分量中的最高点,则以u为起点往下把该强连通分量所有节点都找出来
// 2 要么它就没有环,就是一个正常的往下的点
if(low[u]==dfn[u]) {
    ++scc_cnt;//强连通分量总数+
    int y;
    do{
        y=q.top();
        q.pop();
        st[y]=0;
        id[y]=scc_cnt;
        sz[scc_cnt]++;
    }while(y!=u);
}

//1 因为栈中越高的元素的dfs序越大,那么我们只需要把dfs序比u大的这些pop到u
//即因为最终会从下至上回到u 所以当y==u
//则说明点u所在的所有强连通分量都标记了id
//           → u
//           / /
//           / ne1
//           ← ne2
//           因为ne2会在u能到的dfs序里最大的,也就是此时的栈顶
//           那么我们就逐一pop出ne2和ne1
//2 要么它就是一个没有环的点 则该点单点成一个连通分量
}

void solve() {
    cin>>n>>m;
    memset(h, -1, sizeof h);
    for(int i=1;i<=m;i++) {
        int a, b;
        cin>>a>>b;
        add(a, b);
    }
    for(int i=1;i<=n;i++) {
        if(!dfn[i]) {
            tarjan(i);
        }
    }
}

//统计新图中点的出度
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];
        //a,b不为一个连通分量
        if(a!=b) {
            //a出度+1  dout[a] += i→k
            dout[a]++;
        }
    }
}

```

```

        }
    }

int cnt=0;
int ans=0;
//ans 存的所有出度为0的强连通分量的点的数量
for(int i=1;i<=scc_cnt;i++) {
    //如果第i个强连通分量出度==0
    if(!dout[i]) {
        cnt++;
        ans+=sz[i];//则加上第i个强连通分量的点的个数
    }
    if(cnt>1) {//如果有k>1个出度为0的 则会存在k-1头牛不被所有牛欢迎
        cout<<0<<' \n' ;
        return ;
    }
}
cout<<ans<<' \n' ;
}

```

h5 将整个图变成SCC

学校网络



因此将一个图变成SCC则需要最少添加max (起点数量, 终点数量) 条边

```

const int N=110, M=N*N;
int n;
int h[N], e[M], ne[M], idx;
bool st[N];
stack<int> q;
int dfn[N], low[N], din[N], dout[N], id[N], timestamp, scc_cnt;
void tarjan(int u) {
    dfn[u]=low[u]=++timestamp;
    q.push(u);
    st[u]=1;
    for(int i=h[u];~i;i=ne[i]) {
        int j=e[i];
        if(!dfn[j]) {
            tarjan(j);
            low[u]=min(low[u], low[j]);
        } else if(st[j]) {
            low[u]=min(low[u], dfn[j]);
        }
    }
    if(low[u]==dfn[u]) {
        ++scc_cnt;
        int y;
        do{

```

```

        y=q.top();
        q.pop();
        st[y]=0;
        id[y]=scc_cnt;
    }while(y!=u);
}
}

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

void solve(){
    cin>>n;
    memset(h,-1,sizeof h);
    for(int i=1;i<=n;i++){
        int x;
        while(cin>>x,x){
            add(i,x);
        }
    }
    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;j=ne[j]){
            int a=id[i],b=id[e[j]];
            if(a!=b){
                dout[a]++;
                din[b]++;
            }
        }
    }
    int cnt1=0,cnt2=0;
    for(int i=1;i<=scc_cnt;i++){
        if(!dout[i]){
            cnt2++;
        }
        if(!din[i]){
            cnt1++;
        }
    }
    cout<<cnt1<<'\n';
    if(scc_cnt==1){
        cout<<0<<'\n';
    }else{
        cout<<max(cnt1,cnt2)<<'\n';
    }
}
}

```

H5 tarjan缩点拓扑序递推dp求半联通子图最长链和方案数

```
/*
强连通:
u <=> v
半连通
u -> v
或
v -> u
则强连通分量必然是半连通

1 求最大半连通子图
方法:
先把所有强连通求出来并缩点后得到拓扑图
1 tarjan求scc
2 缩点 建图 给边判重

▲
方案数指选点的方案数
两个不同的方案==两个方案有一个点不同!=两个方案有一个边不同
根据定义:
若  $G' = (V', E')$  满足,  $E'$  是  $E$  中所有和  $V'$  有关的边
知道 如果选了两个点  $x, y$ , 且它们之间有多条边
    →
    ↑     ↓
    x → y
则必须把这三条边都选出来
即对于选中的两点, 他们之间的边要么全选, 要么一条都不选

★ 所以建图时要给边判重
不判重在找前驱时是按边来算的, 如果有多条边, 我们就会算多次
在  $f[j] + s[i] == f[i]$  的情况下
会  $g[i] += g[j]$  多次
3 按拓扑序递推
scc1 → scc2 → scc3
    ↘     ↗
        scc4
    ↗     ↘
scc5           scc6
求最大半连通子图
    <=>
求最长无分叉链(有分叉时 scc3 不能走到 scc6, scc6 也不能走到 scc3)
链上权重是连通分量里节点数量
2 统计最长链方案数
统计最长链方案数
    <=>
拓扑图求最长路(权重是结点数, 权重越大, 结点数越多)
权重最大值           $f[i]$ 
让  $f[i]$  最大的方案数  $g[i]$ 
强连通分量  $i$  结点数  $s[i]$ 
 $j \searrow$ 
 $o \rightarrow i \text{ if } f[j] + s[i] > f[i]:$ 
 $o \nearrow \quad \text{更新路径权重 } f[i] = f[j] + s[i]$ 
```

```

        更新方案数g[i]=g[j]
    if f[j]+s[i] == f[i]:
        不用更新路径权重
        只更新方案数g[i]+=g[j]

/*

```

最大半联通子图

```

const int N=1e5+10, M=2e6+10;
int h[N], hh[N], e[M], ne[M], idx;
int dfn[N], low[N], id[N], timestamp, scc_cnt;
bool st[N];
stack<int> q;
int n, m, mod;
int f[N], g[N]; //缩点后拓扑图上的最长链和方案数
int sz[N];
void add(int h[], int a, int b) {
    e[idx]=b, ne[idx]=h[a], h[a]=idx++;
}
void tarjan(int u) {
    dfn[u]=low[u]=++timestamp;
    q.push(u);
    st[u]=1;
    for(int i=h[u]; ~i; i=ne[i]) {
        int j=e[i];
        if(!dfn[j]) {
            tarjan(j);
            low[u]=min(low[u], low[j]);
        } else if(st[j]) {
            low[u]=min(low[u], dfn[j]);
        }
    }
    if(dfn[u]==low[u]) {
        scc_cnt++;
        int y;
        do{
            y=q.top();
            q.pop();
            st[y]=0;
            sz[scc_cnt]++;
            id[y]=scc_cnt;
        }while(y!=u);
    }
}
void solve() {
    cin>>n>>m>>mod;
    memset(h, -1, sizeof h);
    memset(hh, -1, sizeof hh);
    for(int i=1; i<=m; i++) {
        int a, b;

```

```

    cin>>a>>b;
    add(h, a, b);
}

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

unordered_set<ll> S;//hash映射，判重边
//建新图
for(int i=1;i<=n;i++) {
    for(int j=h[i];~j;j=ne[j]) {
        int a=id[i], b=id[e[j]];
        ll v=a*1e6+b;
        if(a!=b && !S.count(v)) {
            add(hh, a, b);
            S.insert(v);
        }
    }
}

for(int i=scc_cnt;i;i--) {
    if(!f[i]) {
        f[i]=sz[i];
        g[i]=1;
    }
}

//拓扑序递推dp
for(int j=hh[i];~j;j=ne[j]) {
    int k=e[j];
    if(f[k]<f[i]+sz[k]) {
        f[k]=f[i]+sz[k];
        g[k]=g[i];
    } else if(f[k]==f[i]+sz[k]) {
        g[k]+=g[i];
        g[k]%=mod;
    }
}
}

int maxf=0, ans=0;//找拓扑序上的最长链和方案数
for(int i=1;i<=scc_cnt;i++) {
    if(f[i]>maxf) {
        maxf=f[i];
        ans=g[i];
    } else if(f[i]==maxf) {
        ans+=g[i];
        ans%=mod;
    }
}

cout<<maxf<<'\n'<<ans<<'\n';
}

```

无向图的双联通分量（重联通分量）

1. 边的双连通分量 e-dcc 极大的不包含桥的连通块
- 2 点的双连通分量 v-dcc 极大的不包含割点的连通块

核心变量

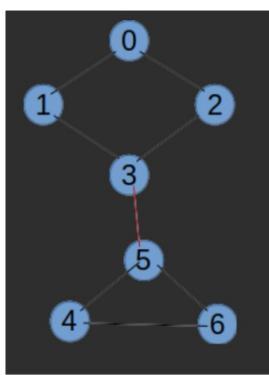
$\text{dfn}[u]$: 当前到达节点u的时间戳 (dfs序)

$\text{low}[u]$: u能达到的时间戳最小的点的时间戳

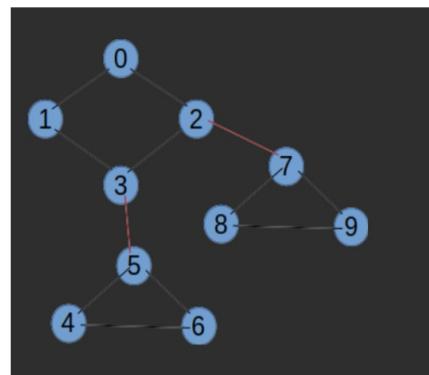
H5 边双联通 (e-DCC)

桥

对于无向图,如果删除了一条边,整个图的联通分量数量变化,则这条边称为桥



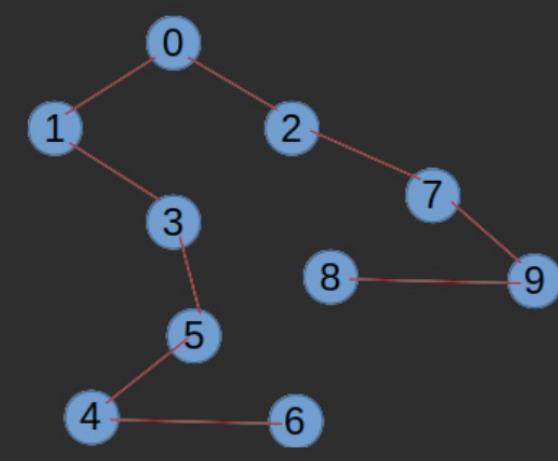
3-5这条边就被称为桥



3-5 和 2-7这两条边就被称为桥

- 一棵树的所有边都是桥

如下图,红色边都是图中的桥,一颗树中任意一条边的断开都会导致图中联通分量发生变化

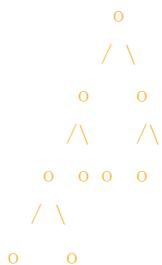


H6 求解加多少条边可以将整个图变成双连通分量

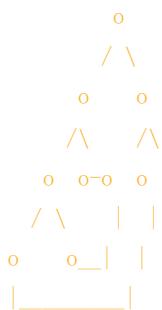
冗余路径

```
/*
```

对双连通分量做完缩点后 只剩桥和点



(叶子节点相互相连，如果是单数的话，剩余的一个就随便找一个树节点相连即可)



可以发现对左右两个叶子节点连通后，根节点连向左右叶子节点的边就可以删去了

同理 再把第2个和第4个叶子节点连通后，根节点连向第2个和第4个叶子节点的边也可以删去

第3个叶子节点随便连

给叶子节点按对称性加上边后就没有桥 \Leftrightarrow 变成边的双连通分量

这里cnt= 5 加了ans=(cnt+1)/2=3条

ans >= 下界[cnt/2]取整 == [(cnt+1)/2]取整

(其中 cnt为缩完点后度数==1的点(叶子节点)的个数)

```
*/
```



```
const int N = 5010, M = 20010;
int h[N], ne[M], e[M], idx;
int dnt[N]; //dnt[u]表示当前到达u
int low[N]; //low[u]表示从u节点出发能遍历到的最长时间戳
int id[N]; //记录节点i所在的双连通分量的编号
int d[N]; //记录节点i的度
bool is_bridge[N]; //记录边是否是桥边
stack<int> sta;
int dcc_cnt; //为双连通分量的编号
int timestamp; //时间戳
int n, m;
```

```

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

void tarjan(int u, int from) { //from记录的是当前节点由哪条边过来的(防止反向遍历)

    low[u] = dnt[u] = ++timestamp;
    sta.push(u);
    //先将图的所有节点的low和dnt都预处理出来
    for(int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if(!dnt[j]) {
            tarjan(j, i);
            low[u] = min(low[u], low[j]);
            //表示j节点永远都走不到u节点(u, j这两个点只能从u走到j), 所以边u-j(w[i])是一条桥边
            if(dnt[u] < low[j]) {
                is_bridge[i] = true;
                is_bridge[i ^ 1] = true; //反向边同样是桥边
                //因为一对正向边和反向边的编号都是成对出现, 所以直接^1即可
                // 这里i==idx 如果idx==奇数 则反向边=idx+1 = idx^1
                // 如果idx==偶数 则反向边=idx-1 = idx^1
            }
            //from ^ 1是边from的反向边, 防止再遍历回去
        } else if(i != (from ^ 1)) {
            //j节点有可能是之前遍历过的点
            low[u] = min(low[u], dnt[j]);
        }
    }
    //预处理完之后判断哪个节点的low[u] = dnt[u], 说明是双联通分量的最高点
    if(low[u] == dnt[u]) {
        int y;
        ++dcc_cnt;
        do{
            y = sta.top();
            sta.pop();
            id[y] = dcc_cnt;
        }while(y != u);
    }
}

int main() {
    memset(h, -1, sizeof h);
    cin >> n >> m;
    for(int i = 0; i < m; ++i) {
        int a, b;
        cin >> a >> b;
        add(a, b); add(b, a);
    }

    for(int i = 1; i <= n; ++i) {
        if(!dnt[i]) tarjan(i, -1); //引入from防止反向遍历回遍历过的边
    }
}

```

```

//遍历所有边，如果边i是桥边，在其所连的出边的点j所在双连通分量的度+1
for(int i = 0; i < idx; ++i){      //包含正向边和反向边，所以度为1的节点一定是叶子节点
    if(is_bridge[i]){
        d[id[e[i]]]++;
    }
}
int cnt=0;
//枚举所有双连通分量，需要加的边数就是:(度为1的节点个数 + 1) / 2
for(int i = 1; i <= dcc_cnt; ++i){
    if(d[i] == 1) cnt++;
}
cout << (cnt + 1) / 2 << endl;
return 0;
}

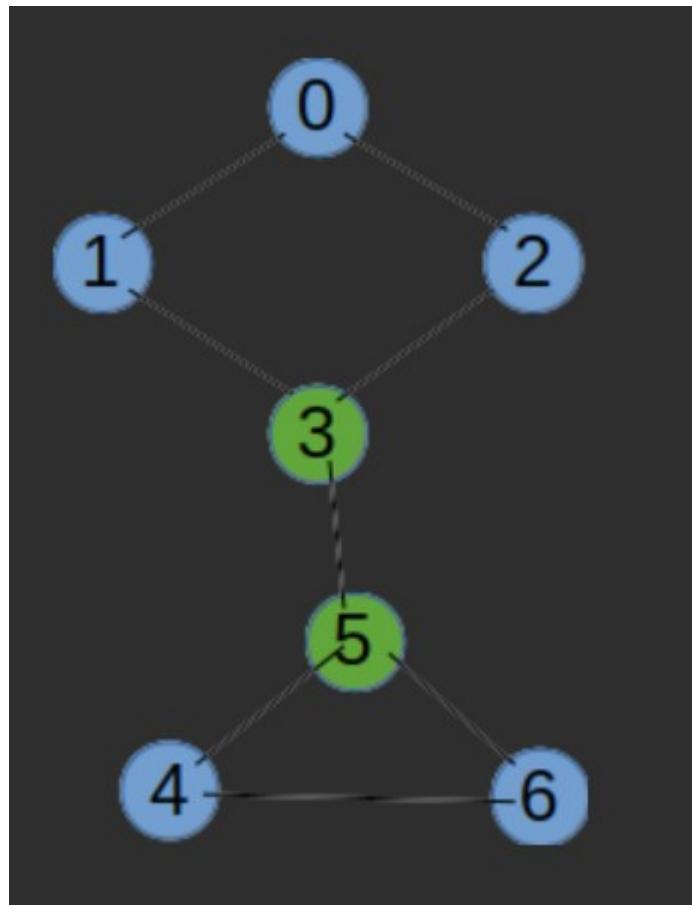
```

h5 点双联通 (v-DCC)

割点

对于无向图,如果删除了一个顶点(顶点邻边也删除),整个图的联通分量数量改变,则称这个顶点为割点。
(与桥没有关系)

如下图,顶点3和顶点5就是该图的两个割点



H6 求割点（求无向图删除一个节点之后还剩下多少联通块）



电力

```
const int N = 10010, M = 30010;

int n, m;
int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp; //dfn兼判重数组
int root; // 记录每个连通块的“根节点”
int ans; // 记录每个连通块去掉一个点形成的连通块数目的最大值
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;
    int s = 0; // 如果当前点u是割点的话，去掉该点u得到的连通分量的个数
    for (int i = h[u]; ~i; i = ne[i]) {
        int j = e[i];
        if (!dfn[j]) {
            tarjan(j);
            low[u] = min(low[u], low[j]);
            if (dfn[u] <= low[j]) // 说明u是可能是割点，u存在一棵子树(删除割点u)
                s++;
        } else low[u] = min(low[u], dfn[j]);
    }
}

//如果不是根节点
/*
    /
    u      删掉u后 除子节点yi外
    / \          还要加上父节点部分+1
    o      o
*/
//最后还要加上父节点部分1
if (u != root) s++; // 不用加上&& s的判断, 因为u不是割点的话, s要取1
ans = max(ans, s);
}

int main() {
    while (scanf("%d%d", &n, &m), n || m) {
        memset(dfn, 0, sizeof dfn); // dfn还具有判重数组的作用
        memset(h, -1, sizeof h);
        idx = timestamp = 0;
        while (m--) {
            int a, b;
            scanf("%d%d", &a, &b);
            add(a, b), add(b, a);
        }
        ans = 0; //记录删除不同割点之后形成的连通块的最大值
        int cnt = 0; // 记录连通块的数目
        //每次将其中联通块遍历, 用tarjan
```

```

        for (root = 0; root < n; root++) // 节点编号从0~n-1
            if (!dfn[root]) { // dfn数组兼判重数组, 求联通块的数量
                cnt++; // 联通块数量++
                tarjan(root);
            }
        printf("%d\n", cnt + ans - 1);
    }
    return 0;
}

```

H6 求解最少设置多少个起点使无向图无论删除哪一个点都能联通，并且输出设置的方案数

矿场搭建

题意：给一个不一定连通的无向图，问最少在几个点设置出口使得任意一个出口坏掉后，其他所有点都可以和某个出口连通。

```

typedef unsigned long long ULL;
const int N = 1010, M = 1010;
int maxn, m; //maxn记录最大的编号数, 即节点的个数
int h[N], e[M], ne[M], idx;
int dfn[N], low[N], timestamp;
stack<int> sta; //sta存点双联通分量
int dcc_cnt; //点双连通分量的编号
vector<int> dcc[N]; //记录每个点双联通分量内的所有节点
bool cut[N]; //判断节点i是否为割点
int root;
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
void tarjan(int u)
{
    low[u] = dfn[u] = ++timestamp;
    sta.push(u);
    // 1 u是孤立点-自称一个dcc
    if (u == root && h[u] == -1) //u是根节点且没有邻边
    {
        dcc_cnt++;
        dcc[dcc_cnt].push_back(u);
        return;
    }
    // 2 u不孤立
    int cnt = 0;
    for (int i = h[u]; ~i; i = ne[i])
    {
        int j = e[i];
        if (!dfn[j])
        {

```

```

        tarjan(j);
        low[u] = min(low[u], low[j]); //有可能存在反向边

        // 看j是不是能连到比u还高的地方
        if(dfn[u]<=low[j])//j最高比u高度低 说明j是u一个新的分支(如果把u删掉 多一个j连通
块)
    {
        cnt++;
        // 判断u是否割点 如果不是根节点-只要有一个分支他就是割点 || 如果是根节点
需要有两个分支才是割点
        //      root          /
        //      / \           非root(自带上面一个边, 所以只要一个下分支)
        //                  /
        if(u!=root||cnt>1)cut[u] = true;
        ++dcc_cnt;
        int y;
        //执行low[j] > dnt[u]说明一个点双连通分量已经遍历完了, 所以将他记录
        do{
            y = sta.top();
            sta.pop();
            dcc[dcc_cnt].push_back(y);
        }while(y!=j); //注意弹出栈不是弹到u为止 而是弹到j为止(u仍保留在stk中)
        // ▲ 开新分支 == u一定和新分支j组成一个dcc 也和旧连通块组成dcc
        // 那么当前最高点u还要被用在更高的包含u的旧连通块
        // 所以如果这个时候出栈了 回溯到比u高的点的时候 u就加不进旧连通块里
        dcc[dcc_cnt].push_back(u);
    }
}
else low[u] = min(low[u], dfn[j]);
}

int main()
{
    int T = 1;
    while(cin >> m, m)
    {
        for(int i=1;i<=dcc_cnt;i++)dcc[i].clear();
        while(sta.size()) sta.pop(); //栈也记得清空一下

        idx = maxn = timestamp = dcc_cnt = 0;
        memset(h, -1, sizeof h);
        memset(dfn, 0, sizeof dfn);
        memset(cut, 0, sizeof cut);
        while(m--)
        {
            int a, b;
            cin >> a >> b;
            maxn = max(maxn, b), maxn = max(maxn, a);
            add(a, b), add(b, a);
        }
        for (root = 1; root <= maxn; root++)
            if (!dfn[root])

```

```

        tarjan(root);
        int res = 0;
        ULL num = 1;
        //缩点过程, 枚举所有点的双连通分量, 记录每个双连通分量的割点个数, 并分类讨论
        for(int i = 1; i<=dcc_cnt; i++)
        {
            int cnt = 0;    //编号为i的点双连通分量的割点个数
            for(int j= 0; j<dcc[i].size(); j++)//j< 写成了i<
            {
                if(cut[dcc[i][j]])
                    cnt++;
            }
            //没有割点, 第一种情况
            if(cnt == 0)
            {
                if(dcc[i].size()>1) res+=2 , num*=dcc[i].size()*(dcc[i].size()-1)/2;
                else res++;
            }
            else if(cnt==1) res++ , num*=dcc[i].size()-1; //只有一个割点, 第二种情况
            else num *= 1;      //割点数量为2, 第三种情况
        }
        printf("Case %d: %d %llu\n", T++, res, num);
    }
    return 0;
}

```

二分图

1 二分图 不存在奇数环

\Leftrightarrow

染色法不存在矛盾

2 匈牙利算法 匹配、最大匹配、匹配点、增广路径

3 最小点覆盖、最大独立集、最小路径点覆盖、最小路径重复点覆盖

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

4 最优匹配 KM

最小费用流

5 多重匹配 每个点能匹配多个点

最大流

匹配/匹配边：与其他边没有公共节点的一条边，我们称这条边为一个匹配/匹配边。

匹配点：匹配边上的点

非匹配点：不在匹配边上的点

非匹配边：图中两点之间的边不是匹配边的边

最大匹配(匹配边数量的最大值)：最多连多少条边，使得所有的边无公共点

★ 增广路(径)：一边的非匹配点到另一边的非匹配点的一条非匹配边和匹配边交替经过的路径。

通俗：由一个未匹配的顶点开始，经过若干个匹配顶点，最后到达对面集合的一个未匹配顶点的路径，即这条路径将两个不同集合的两个未匹配顶点通过一系列匹配顶点相连。

初始：匹配边为一根线(—)，非匹配边为两根线(— —)

1 o — o 2

\\



此时 $3 \rightarrow 2 \rightarrow 1 \rightarrow 4$ 就是一条 非匹配边上的点 $3 \rightarrow$ 非匹配边上的点 4 的增广路径

通过该增广路径可以让 $3-2$ 匹配, $1-4$ 匹配从而实现最大匹配

(只要有一条增广路径就能多一组匹配, 最大匹配 \Leftrightarrow 不存在增广路径)

在一张图中, 如果能够把全部的点分到 两个集合 中, 保证两个集合内部没有任何边 , 图中的边只存在于两个集合之间, 这张图就是二分图

- 如果两个集合中的点分别染成黑色和白色, 可以发现二分图中的每一条边都一定是连接一个黑色点和一个白色点。
- 二分图不存在长度为奇数的环, 因为每一条边都是从一个集合走到另一个集合, 只有走偶数次才可能回到同一个集合。

H5 染色法判定二分图

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

using namespace std;

const int N = 100010 * 2;
int e[N], ne[N], idx;//邻接表存储图
int h[N];
int color[N];//保存各个点的颜色, 0 未染色, 1 是红色, 2 是黑色
int n, m;//点和边

void add(int a, int b)//邻接表插入点和边
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}

bool dfs(int u, int c)//深度优先遍历
{
    color[u] = c;//u的点成 c 染色

    //遍历和 u 相邻的点
    for(int i = h[u]; i != -1; i = ne[i])
    {
        int b = e[i];
        if(!color[b])//相邻的点没有颜色, 则递归处理这个相邻点
        {
            if(!dfs(b, 3 - c)) return false;// (3 - 1 = 2, 如果 u 的颜色是2, 则和 u 相邻的染成 1)
                                            // (3 - 2 = 1, 如果 u 的颜色是1, 则和 u 相邻的染成 2)
        }
        else if(color[b] && color[b] != 3 - c)//如果已经染色, 判断颜色是否为 3 - c
        {

```

```

        return false;//如果不是，说明冲突，返回
    }
}

return true;
}

int main()
{
    memset(h, -1, sizeof h);//初始化邻接表
    cin >> n >> m;
    for(int i = 1; i <= m; i++)//读入边
    {
        int a, b;
        cin >> a >> b;
        add(a, b), add(b, a);
    }
    for(int i = 1; i <= n; i++)//遍历点
    {
        if(!color[i])//如果没染色
        {
            if(!dfs(i, 1))//染色该点，并递归处理和它相邻的点
            {
                cout << "No" << endl;//出现矛盾，输出NO
                return 0;
            }
        }
    }
    cout << "Yes" << endl;//全部染色完成，没有矛盾，输出YES
    return 0;
}

```

关押罪犯

```

/*
本题
即将囚犯分为两边
所有分配方案都有一个最大值，找所有方案最大值的最小值
所以可以用二分来搜这个下界方案
如果mid值能够通过某个划分方案后变为二分图达到
就去找更小的可能的下界方案
*/
const int N=2e4+10, M=2e5+10;
int h[N], e[M], ne[M], w[M], idx;
int color[N];
int n, m;
void add(int a, int b, int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
bool dfs(int u, int c, int mid) {
    color[u]=c;
    for(int i=h[u]; ~i; i=ne[i]) {

```

```

        int j=e[i];
        if(w[i]<=mid) continue;
        if(color[j]){
            if(color[j]==c) return true;
        }else if(dfs(j, 3-c, mid)) return true;
    }
    return false;
}

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

void solve(){
    cin>>n>>m;
    memset(h, -1, sizeof h);
    for(int i=1; i<=m; i++) {
        int a, b, c;
        cin>>a>>b>>c;
        add(a, b, c), add(b, a, c);
    }
    int l=0, r=1e9;
    while(l<=r) {
        int mid=l+r>>1;
        if(check(mid)) r=mid-1;
        else l=mid+1;
    }
    cout<<r+1<<' \n';
}
}

```

H5 匈牙利算法

时间复杂度 $O(mn)$ n 表示点数, m 表示边数

二分图的最大匹配

```

#include<iostream>
#include <cstring>
#include<algorithm>
using namespace std;
int n1, n2, m;
// n1表示第一个集合中的点数, n2表示第二个集合中的点数
// 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 h[500], e[100010], ne[100010], idx = 0;
// 表示第二个集合中的每个点是否已经被遍历过
// match[x]: 和 x 编号的男生的编号
// match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

```

```

int st[510], match[510];
//存图函数
void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a]; h[a] = idx++;
}
//递归找可以匹配的点
bool find(int x) {
    // 和各个点尝试能否匹配
    for(int i = h[x]; i != -1; i = ne[i]) {
        int b = e[i];
        if(!st[b]){//打标记
            st[b] = 1;
            // 当前尝试点没有被匹配或者和当前尝试点匹配的那个点可以换另一个匹配
            if(match[b] == 0 || find(match[b])) {
                // 和当前尝试点匹配在一起
                match[b] = x;
                return true;
            }
        }
    }
    return false;
}

int main() {
    memset(h, -1, sizeof h);
    cin >> n1 >> n2 >> m;
    // 保存图, 因为只从一遍找另一边, 所以该无向图只需要存储一个方向
    for(int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        add(a, b);
    }
    int res = 0;
    //为各个点找匹配
    for(int i = 1; i <= n1; i++) {
        memset(st, 0, sizeof st);
        //找到匹配
        if(find(i)) res++;
    }
    cout << res;
    return 0;
}

```

棋盘覆盖

```

/*
本题:
结论:
一个匹配是最大匹配 <=> 不存在增广路径

o o o
o . o

```

→ 放三个小砖块

10

10

不用状压是因为N==100

所以 2^{100} 太大了

每个卡片塞2个格子

把格子看成点

把卡片看成边

则只要能放卡片的相邻两个格子就连一条边

▲ 问题转化为 \rightarrow 最多取多少条边, 满足卡片不重叠——所有选出的边没有公共点——二分图

放最多的牌——找到最多的边——二分图上最大匹配

* /

```

        }
    }
    cout<<ans<<'\n';
}

```

H5 最小点覆盖

最小点覆盖问题

定义：给我们一个图，从中选出最少的点，使得图中的每一条边至少有一个顶点被选在二分图中

最小点覆盖==最大匹配数

机器任务

```

/*
本题
a[i]=0 || b[i]=0时可以跳过
一个任务i可以被A、B机器的两种状态a[i]、b[i]完成，将一个任务看成一条边，两种状态看成两个端点，要完成一个任务就要从这两个点中选一个点(任务可以在A上执行也可以在B上执行)，对于所有任务就要从N+M-2个点中(不包含初始0状态)选出最少的点，覆盖所有的边(任务)，问题就变成求最小点覆盖问题(二分图中最小点覆盖等价于最大匹配数--匈牙利算法)。
*/
typedef pair<int, int> PII;
const int N=110;
bool g[N][N], st[N];
int n, m, k;
int match[N];
bool find(int x) {
    for(int i=1; i<m; i++) {
        if(!st[i] && g[x][i]) {
            int t=match[i];
            st[i]=1;
            if(t==0 || find(t)) {
                match[i]=x;
                return true;
            }
        }
    }
    return false;
}
void solve() {
    while(cin>>n, n) {
        cin>>m>>k;
        memset(g, 0, sizeof g);
        memset(match, 0, sizeof match);
        for(int i=1; i<=k; i++) {
            int a, b, id;
            cin>>id>>a>>b;
            if(!a || !b) continue;
            g[a][b]=1;
        }
        int ans=0;

```

```

        for(int i=1;i<n;i++) {
            memset(st, 0, sizeof st);
            if(find(i)) ans++;
        }
        cout<<ans<<'\n';
    }
}

```

H5 最大独立集

```

/*
最大独立集
选出最多的点 使得选出的点之间没有边
在二分图中 总共n个点
求最大独立集即求n-m(越大越好)则去掉的点数m越小越好
<=> 去掉最少的点(m个), 将所有边都破坏掉
<=> 找最小点覆盖所有m条边
<=> 找最大匹配m
*/

```

骑士放置

```

/*
每个不禁止放的点是一个图节点
每两个可以通过日字走到的点之间连一条边
问题为选出最多的点 使得选出来的点之间不能通过日字跳走到
<=>
转化为选出最多的点 使得选出来的点之间没有边

求最大匹配用得在二分图情况下才能用匈牙利
那么我们类似372把row+col = 奇or偶 分为 白or黑节点
    o . o . o
    . o . o .
    o . o . o
    . o . o .
    o . o . o
可以发现白节点通过日字跳只能走到黑节点—这是一个二分图
即可以用匈牙利算法做
*/

```

```

typedef pair<int, int> PII;
const int N=110;
bool g[N][N], st[N][N];
int n, m, k;
PII match[N][N];
const int dir[] [2] {{-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2}, {-1, -2}, {-2, -1}};
bool find(int x, int y) {
    for(int k=0;k<8;k++) {
        int xx=x+dir[k][0];
        int yy=y+dir[k][1];
        if(xx>=1 && xx<=n && yy>=1 && yy<=m && !st[xx][yy] && !g[xx][yy]) {
            st[xx][yy]=1;
            PII t=match[xx][yy];

```

```

        if(t.first==0 || find(t.first, t.second)) {
            match[xx][yy]={x, y} ;
            return true;
        }
    }
}

return false;
}

void solve() {
    cin>>n>>m>>k;
    for(int i=1;i<=k;i++) {
        int x, y;
        cin>>x>>y;
        g[x][y]=1;
    }
    int ans=0;
    for(int i=1;i<=n;i++) {
        for(int j=1;j<=m;j++) {
            if((i+j)%2 || g[i][j]) continue;
            memset(st, 0, sizeof st);
            if(find(i, j)) ans++;
        }
    }
    //k是坏掉的点
    cout<<n*m-k-ans<<' \n' ;
}

```

H5 最小路径（重复点）覆盖

最小路径覆盖
DAG(有向无环图)
用最少的互不相交的路径 将所有点覆盖
最大匹配数 = 总点数-最小路径覆盖

取消对路径互不相交的约束后
问题->最小路径重复点覆盖
1 先求传递闭包得到新图G`（路径无不相交）
2 原图G最小路径重复点覆盖=新图G` 最小路径覆盖

捉迷藏

```

/*
本题
最小路径cnt条重复点覆盖
答案就是cnt
*/
typedef pair<int, int> PII;
const int N=210;
bool d[N][N], st[N];
int match[N];
int n, m;
bool find(int x) {

```

```

        for(int i=1;i<=n;i++) {
            if(d[x][i] && !st[i]){
                st[i]=1;
                int t=match[i];
                if(t==0 || find(t)){
                    match[i]=x;
                    return true;
                }
            }
        }
        return false;
    }

void solve(){
    cin>>n>>m;
    for(int i=1;i<=m;i++){
        int a,b;
        cin>>a>>b;
        d[a][b]=1;
    }

    //求传递闭包
    for(int k=1;k<=n;k++) {
        for(int i=1;i<=n;i++) {
            for(int j=1;j<=n;j++) {
                d[i][j]|=d[i][k]&d[k][j];
            }
        }
    }

    int ans=0;
    for(int i=1;i<=n;i++) {
        memset(st,0,sizeof st);
        if(find(i)) ans++;
    }
    cout<<n-ans<<'\n';
}

```

欧拉回路和欧拉路径

欧拉回路:终点就是起点

一、无向图

1 存在欧拉路径的充要条件 : 度数为奇数的点只能有0或2个

2 存在欧拉回路的充要条件 : 度数为奇数的点只能有0个

二、有向图

1 存在欧拉路径的充要条件 : 要么所有点的出度均==入度;

要么除了两个点之外，其余所有点的出度==入度 剩余的两个点:一个满足出度-入度==1(起点) 一个满足入度-出度==1(终点)

2 存在欧拉回路的充要条件 : 所有点的出度均等于入度

```

/*
_0_ 环可以合并进边 或者环可以和环合并 00
    o

```



对于除了起点与终点外中间的点

由于它们的度数是偶数 则只要它从某一个过环的出度出发 则必然会走完这个环回到这个点

合并环的方法:有环先走环

```
dfs(u)
{
    for 从n出发的所有边
        dfs() 扩展
        把u加入序列 seq[]←u
}
```

最终 seq[]中存下的就是欧拉路径的倒序

有向图:

每用一条边 删掉

无向图:

每用一条边标记对应的反向边(XOR技巧)

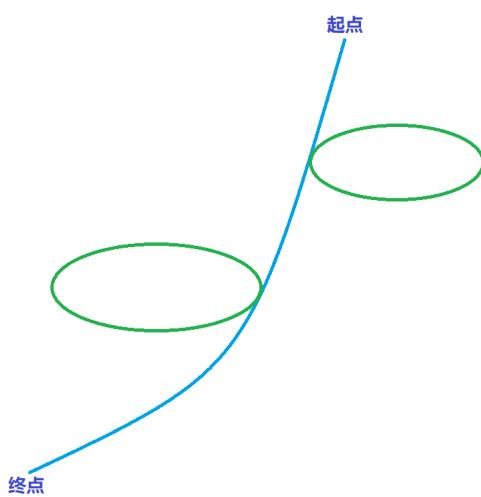
$a \rightarrow b$

$b \rightarrow a$

*/

H6 求欧拉回路或者路径（删边优化）

欧拉回路



欧拉路径:

从起点出发,可能一直沿着曲线走到终点,而两侧的绿色圈圈可能没有走过,但是从终点回溯时,绿色的圈是可以被遍历到的,所以在记录路径的过程中,可以先遍历,然后再将路径记录下来,因为是从终点记录到起点的,所以需要逆序输出,即从起点到达终点

删边优化操作:



因为每次遍历一条边时,会将所有相关的边都遍历一遍,例如上方的节点i.假设自环总数为2000
那么最开始输入的边,相关的边有2000条,第二条输入的边,相关的边有2000条,依次往复,所以在做过程中需要进行删边优化操作.

dfs 起点 删边操作的两种方式:

1.先遍历,在删边

dfs(j);

h[u]=ne[i];

这样做的话,遍历一般会先遍历到底层,当回溯到底层时,最低层的代码也就删了一条边,然后再从底层遍历到上层(因为是自环),到达最上层时还是只删除了一条边,那么依次往复,总共遍历了 $2000 + 1999 + 1998 + \dots + 1$ 次,时间复杂度爆表

2.先删边,后遍历

这样做的话,每遍历一次,就有一条边会被删掉,所以遍历到底层之后,即遍历了2000次后,2000条边也就被删光了,所以第二种方法可取:

底层

边的转化

无向图中的边的序号为:0 ~ 2m-1,而在题目中的编号是1 ~ m;
所以转化边时:0,1对应1号边,2,3对应2号边,即x号边对应x/2+1;
其中偶数是正向边,奇数是反向边,可以用 x^1 转化正反向边的关系;
如果是奇数: x^1 中,x二进制前面的数不变,最后一位变成0,即 $x^1=x-1$;
如果是偶数: x^1 中,x二进制前面的数不变,最后一位变成1,即 $x^1=x+1$;

```
typedef pair<int, int> PII;
const int N=1e5+10, M=4e5+10;
int type;
int h[N], e[M], ne[M], idx;
bool used[M];
int n, m, cnt, ans[M];
int din[N], dout[N];
void add(int a, int b) {
    e[idx]=b, ne[idx]=h[a], h[a]=idx++;
}
void dfs(int u) {
    //i一直等于h[u], 遍历到最深处然后回溯的过程每次删除一条边
    for(int i=h[u]; ~i; i=ne[u]) {
        if(used[i]) {
            h[u]=ne[i];
            continue;
        }
        used[i]=1;
        if(type==1) used[i^1]=1;
        int t;
        if(type==1) {
            t=i/2+1;
            if(i&1) t=-t;//反向边
        } else{
            t=i+1;
        }
        int j=e[i];
        h[u]=ne[i];
        dfs(j);
        ans[++cnt]=t;
    }
}
```

```

void solve() {
    cin>>type;
    cin>>n>>m;
    memset(h,-1,sizeof h);
    for(int i=1;i<=m;i++) {
        int a,b;
        cin>>a>>b;
        add(a,b);
        if(type==1) add(b,a);
        din[b]++, dout[a]++;
    }
    if(type==1) {
        for(int i=1;i<=n;i++) {
            if(din[i]+dout[i]&1) {
                cout<<"NO"<<' \n' ;
                return ;
            }
        }
    } else{
        for(int i=1;i<=n;i++) {
            if(din[i]!=dout[i]) {
                cout<<"NO"<<' \n' ;
                return ;
            }
        }
        for(int i=1;i<=n;i++) {
            if(h[i]==-1) {
                dfs(i);
                break;
            }
        }
        if(cnt<m) {
            cout<<"NO"<<' \n' ;
        } else{
            cout<<"YES"<<' \n' ;
            for(int i=cnt;i;i--) {
                cout<<ans[i]<<' ' ;
            }
            cout<<' \n' ;
        }
    }
}

```

H6 无向图欧拉路径输出字典序最小的一条路

邻接矩阵写法

```

const int N = 510;
int n = 500, m;

```

```

//邻接矩阵存图
int g[N][N];
int ans[1100], cnt;
int d[N];
void dfs(int u)
{
    //因为最后的欧拉路径的序列是ans数组逆序,
    //节点u只有在遍历完所有边之后最后才会加到ans数组里面, 所以逆序过来就是最小的字典序
    for (int i = 1; i <= n; i++)
        if (g[u][i])
    {
        //删边优化
        g[u][i] --, g[i][u] -- ;
        dfs(i);
    }
    ans[ ++ cnt] = u;
}

int main()
{
    cin >> m;
    while (m -- )
    {
        int a, b;
        cin >> a >> b;
        g[a][b] ++, g[b][a] ++ ;
        d[a] ++, d[b] ++ ;
    }
    int start = 1;
    while (!d[start]) ++start; // 较小编号作为起点, 避免欧拉回路没有度数为奇数的点, 然后没有起点了。
    //数据保证有解一定存在欧拉回路, 那么让第一条度数为奇数的点作为起点
    for (int i = 1; i <= 500; ++i) {
        if (d[i] % 2) { // 奇数点作为起点
            start = i;
            break;
        }
    }
    dfs(start);
    for (int i = cnt; i; i -- ) printf("%d\n", ans[i]);
    return 0;
}

```

邻接表写法

```

const int N= 5e3 + 10;
typedef pair<int, int> PII;
int n =500, m;
int h[N], e[N], ne[N], idx;
vector<int> ans;
bool st[N];
int d[N];

```

```

int ss[N];
void add (int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
bool cmp (PII a, PII b) {
    return a.first < b.first;
}
void dfs(int x) {
    PII temp[N];
    int cnt = 0;
    for (int i = h[x]; ~i; i = ne[i]) {
        temp[cnt++] = {e[i], i};
    } // 把所有点提取出来
    sort (temp, temp + cnt, cmp); // 从小到大排序
    for (int i = 0; i < cnt; i++) {
        int edge = temp[i].second;
        if (st[edge] || st[edge^1]) continue;
        st[edge] = st[edge^1] = true;
        dfs(temp[i].first);
    }
    ans.push_back(x);
}
int main () {
    memset (h, -1, sizeof h);
    cin >> m;
    int root = -1;
    while (m--) {
        int a, b;
        cin >> a >> b;
        root = max(root, max(a, b));
        add(a, b); add(b, a);
        d[a]++, d[b]++;
    }
    int start = 1;
    while (!d[start]) start++;
    for (int i = 1; i <= n; i++) {
        if (d[i] % 2) {
            start = i;
            break;
        }
    }
    dfs(start);
    for (int i = ans.size() - 1; i >= 0; i--) {
        cout << ans[i] << "\n";
    }
    return 0;
}

```

H6 判断有向图中是否有欧拉路径

单词游戏

- (1) 所有的边要连通;
- (2) 除了起点和终点外，其余点的入度必须等于出度;

每个单词看成一条边，首字母和尾字母看成图中的顶点。这是一个有向图。

对于 (1)，可以使用并查集解决，如果某个点有边相连，但是和并查集不连通，说明边不连通。

本题不需要将所有边存储下来，只需要判断连通性以及记录每个点的入度和出度判断是否存在答案即可。

```
typedef pair<int, int> PII;
const int N=30;
int din[N], dout[N];
int p[N];
bool st[N];// 记录某个字母是否作为单词首尾出现过
int n;
int find(int x) {
    if(x!=p[x]) p[x]=find(p[x]);
    return p[x];
}
void solve() {
    cin>>n;
    string s;
    //多组测试数据
    memset(din, 0, sizeof din);
    memset(dout, 0, sizeof dout);
    memset(st, 0, sizeof st);
    for(int i=0;i<26;i++) p[i]=i;
    for(int i=1;i<=n;i++) {
        cin>>s;
        int a=s[0]-'a', b=s[s.size()-1]-'a';
        st[a]=st[b]=1;
        din[b]++, dout[a]++;
        p[find(a)]=find(b);
    }
    bool ok=1;
    int S=0, T=0;// 起点和终点的数量，判断度数是否正确
    for(int i=0;i<26;i++) {
        if(din[i]!=dout[i]){
            if(din[i]==dout[i]+1) T++;
            else if(dout[i]==din[i]+1) S++;
            else{
                cout<<"The door cannot be opened."<<'\n';
                return ;
            }
        }
    }
    //起点和终点有的话只能有一个，要么就都没有
    if(!((S==1 && T==1) || (S==0 && T==0))) {
```

```

        cout<<"The door cannot be opened."<<'\n';
        return;
    }

    int t=-1;
    for(int i=0;i<26;i++) {
        if(st[i]){// 说明该字母在单词中出现过，所有st为true的都应该在一个集合中
            if(t== -1) t=find(i);
            else if(t!=find(i)){// 说明存在边不连通
                cout<<"The door cannot be opened."<<'\n';
                return ;
            }
        }
    }
    cout<<"Ordering is possible."<<'\n';
}

```

> 数学知识

试除法判定质数

```

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;
}

```

试除法分解质因数

```

void divide(int x)
{
    for (int i = 2; i <= x / i; i++)
        if (x % i == 0)
        {
            int s = 0;
            while (x % i == 0) x /= i, s++;
            cout << i << ' ' << s << endl;
        }
    if (x > 1) cout << x << ' ' << 1 << endl;
    cout << endl;
}

```

阶乘分解

```

typedef pair<int, int> PII;

```

```

const int N=1e6+10;
int primes[N], cnt;
bool st[N];
void init(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]=1;
            if(i%primes[j]==0) break;
        }
    }
}
void solve() {
    int n;
    cin>>n;
    init(n);
    for(int i=0; i<cnt; i++) {
        int p=primes[i];
        int s=0;
        for(int j=n; j;j/=p) s+=j/p;
        cout<<p<< ' '<<s<< '\n';
    }
}

```

朴素筛法求素数

时间复杂度 O(nlogn)

```

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)
{
    //从1~n的所有素数
    for (int i = 2; i <= n; i++)
    {
        if (st[i]) continue;
        primes[cnt ++] = i;
        //所有倍数一定不是素数
        for (int j = i + i; j <= n; j += i)
            st[j] = true;
    }
}

```

埃氏筛法

时间复杂度 O(nloglogn)

```

void get_primes() {
    for(int i=2;i<=n;i++) {
        if(!st[i]) {
            primes[cnt++]=i;
            for(int j=i;j<=n;j+=i) st[j]=true;//可以用质数就把所有的合数都筛掉;
        }
    }
}

```

线性筛法求素数

时间复杂度 O(n)

```

void get_primes() {
    //外层从2~n迭代，因为这毕竟算的是1~n中质数的个数，而不是某个数是不是质数的判定
    for(int i=2;i<=n;i++) {
        if(!st[i]) primes[cnt++]=i;
        for(int j=0;primes[j]<=n/i;j++) {//primes[j]<=n/i:变形一下得到——primes[j]*i<=n, 把大于n的合数都筛了就
            //没啥意义了
            st[primes[j]*i]=true;//用最小质因子去筛合数

            //1)当i%primes[j]!=0时, 说明此时遍历到的primes[j]不是i的质因子, 那么只可能是此时的primes[j]<i的
            //最小质因子, 所以primes[j]*i的最小质因子就是primes[j];
            //2)当有i%primes[j]==0时, 说明i的最小质因子是primes[j], 因此primes[j]*i的最小质因子也就应该是
            //prime[j], 之后接着用st[primes[j+1]*i]=true去筛合数时, 就不是用最小质因子去更新了, 因为i有最小
            //质因子primes[j]<primes[j+1], 此时的primes[j+1]不是primes[j+1]*i的最小质因子, 此时就应该
            //退出循环, 避免之后重复进行筛选。
            if(i%primes[j]==0) break;
        }
    }
}

```

二次筛法



```

typedef long long LL;

//sqrt(2^31 - 1)
const int N = 1e6 + 10;

bool st[N];
int primes[N], cnt;

```

```

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

int main() {
    int l, r;
    while (~scanf("%d%d", &l, &r)) {
        get_primes(50000);

        //把[l,r]区间内所有的合数用他们的最小质因子筛掉
        memset(st, 0, sizeof st);
        for (int i = 0; i < cnt; ++ i) {
            LL p = primes[i];
            for (LL j = max(2 * p, (l + p - 1) / p * p); j <= r; j += p)
                st[j - 1] = true;
        }

        //剩下的所有的都是素数了
        cnt = 0;
        for (int i = 0; i <= r - 1; ++ i)
            if (!st[i] && i + 1 >= 2)
                primes[cnt ++ ] = i + 1;
        if (cnt < 2) printf("There are no adjacent primes.\n");
        else {
            //计算间隔
            int minp = 0, maxp = 0;
            for (int i = 0; i + 1 < cnt; ++ i) {
                int d = primes[i + 1] - primes[i];
                if (d < primes[minp + 1] - primes[minp]) minp = i;
                if (d > primes[maxp + 1] - primes[maxp]) maxp = i;
            }
            printf("%d,%d are closest, %d,%d are most distant.\n",
                   primes[minp], primes[minp + 1],
                   primes[maxp], primes[maxp + 1]);
        }
    }
    return 0;
}

```

试除法求所有约数

```
vector<int> get_divisors(int x)
{
    vector<int> res;
    for (int i = 1; i <= x / i; i++)
        if (x % i == 0)
        {
            res.push_back(i);
            if (i != x / i) res.push_back(x / i);
        }
    sort(res.begin(), res.end()); //从小到大
    return res;
}
```

约数个数与约数之和

如果 $N = p_1^{c_1} * p_2^{c_2} * \dots * p_k^{c_k}$

约数个数: $(c_1 + 1) * (c_2 + 1) * \dots * (c_k + 1)$

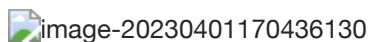
约数之和: $(p_1^0 + p_1^1 + \dots + p_1^{c_1}) * \dots * (p_k^0 + p_k^1 + \dots + p_k^{c_k})$

约数个数推导



求约数个数

```
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;
const int mod = 1e9 + 7;
int main() {
    int n, x;
    LL ans = 1;
    unordered_map<int, int> hash;
    cin >> n;
    while(n--) {
        cin >> x;
        for(int i = 2; i <= x/i; ++i) {
            while(x % i == 0) {
                x /= i;
                hash[i]++;
            }
        }
        if(x > 1) hash[x]++;
    }
    for(auto i : hash) ans = ans*(i.second + 1) % mod;
    cout << ans;
    return 0;
}
```



```
while (b -- ) t = (t * a + 1) % mod
```

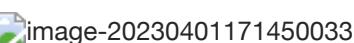
求约数之和

```
#include <iostream>
#include <unordered_map>
using namespace std;
typedef long long ll;
const int mod = 1e9 + 7;
unordered_map<int, int> primes;
int main() {
    int n, x;
    cin >> n;
    while (n--) {
        cin >> x;
        for (int i = 2; i <= x / i; i++) {
            while (x % i == 0) {
                primes[i]++;
                x /= i;
            }
        }
        if (x > 1) primes[x]++;
    }
    ll res = 1;
    for (auto p : primes) {
        ll a = p.first, b = p.second;
        ll t = 1;
        while (b--) t = (t * a + 1) % mod;
        res = res * t % mod;
    }
    cout << res << endl;
    return 0;
}
```

欧几里得算法（辗转相除法）

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

欧拉函数

朴素版

```

int phi(int x)
{
    int res = x;
    for (int i = 2; i <= x / i; i++)
        if (x % i == 0)
        {
            res = res / i * (i - 1);
            while (x % i == 0) x /= i;
        }
    if (x > 1) res = res / x * (x - 1);

    return res;
}

```

筛法求欧拉函数

1) 质数 i 的欧拉函数即为 $\phi[i] = i - 1$, $1 \sim i-1$ 均与 i 互质, 共 $i-1$ 个。

2) $\phi[\text{primes}[j] * i]$ 分为两种情况:

① $i \% \text{primes}[j] == 0$: $\text{primes}[j]$ 是 i 的最小质因子, 也是 $\text{primes}[j] * i$ 的最小质因子, 因此 $1 - 1 / \text{primes}[j]$ 这一项在 $\phi[i]$ 中计算过了, 只需将基数 N 修正为 $\text{primes}[j]$ 倍, 最终结果为 $\phi[i] * \text{primes}[j]$ 。

② $i \% \text{primes}[j] != 0$: $\text{primes}[j]$ 不是 i 的质因子, 只是 $\text{primes}[j] * i$ 的最小质因子, 因此不仅需要将基数 N 修正为 $\text{primes}[j]$ 倍, 还需要乘上 $\frac{\text{prime}[j]-1}{\text{primes}[j]}$ 这一项, 因此最终结果 $\phi[i] * (\text{primes}[j] - 1)$ 。

```

#include<bits/stdc++.h>
using namespace std;
const int N = 1000010;
int n;
int p[N], st[N], cnt;//筛选质数 p[]存放已经找到的质数, st[]标记某个数是不是质数, cnt记录已经找到的质
                      //数数量
int phi[N]; //欧拉函数 Φ(x)
long long int res;
void ola_primes(int x)
{
    for(int i=2;i<=x;i++)
    {
        if(!st[i])
        {
            p[cnt++]=i;
            //从1到质数i中与i互质的数个数为i-1
            phi[i]=i-1;
        }

        for(int j=0;p[j]<=x/i;j++)
        {
            st[p[j]*i]=1;
            if(i%p[j]==0)
            {
                phi[i*p[j]]=p[j]*phi[i];
            }
        }
    }
}

```

```

        break;
    }
    phi[i*p[j]]=(p[j]-1)*phi[i];
}
}
int main()
{
    cin>>n;
    phi[1]=1;
    ola_primes(n);
    for(int i=1;i<=n;i++) res+=phi[i];
    cout<<res<<endl;
    return 0;
}

```

快速幂

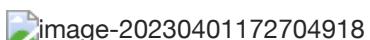
```

//求 m^k mod p, 时间复杂度 O(logk)。
//利用二进制
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;
}

```



扩展欧几里得算法



```

// 求x, y, 使得ax + by = gcd(a, b)
#include<bits/stdc++.h>
using namespace std;
int exgcd(int a, int b, int &x, int &y) {//返回gcd(a,b) 并求出解(引用带回)
    if(b==0) {
        x = 1, y = 0;
        return a;
    }
    int g = exgcd(b, a%b, y, x);
    y -= a/b*x;
    return g;
}

```

```

        return a;
    }

    int x1, y1, gcd;
    gcd = exgcd(b, a%b, x1, y1);
    //回溯
    x = y1, y = x1 - a/b*y1;
    return gcd;
}

int main() {
    int n, a, b, x, y;
    cin>>n;
    while(n--) {
        cin>>a>>b;
        exgcd(a, b, x, y);
        cout<<x<<" "

```

高斯消元



算法步骤

枚举每一列c

1. 找到当前列绝对值最大的一行
2. 用初等行变换(2) 把这一行换到最上面 (未确定阶梯型的行，并不是第一行)
3. 用初等行变换(1) 将该行的第一个数变成 1 (其余所有的数字依次跟着变化)
4. 用初等行变换(3) 将下面所有行的当且列的值变成 0

```

#include <iostream>
#include <algorithm>
#include <cmath>

using namespace std;

const int N = 110;
const double eps = 1e-6;

int n;
double a[N][N];

int gauss()
{
    int c, r;// c 代表列 col , r 代表行 row
    for (c = 0, r = 0; c < n; c++)
    {
        int t = r;// 先找到当前这一列，绝对值最大的一个数字所在的行号
        for (int i = r; i < n; i++)

```

```

        if (fabs(a[i][c]) > fabs(a[t][c]))
            t = i;

        if (fabs(a[t][c]) < eps) continue;// 如果当前这一列的最大数都是 0，那么所有数都是 0,
        就没必要去算了，因为它的约束方程，可能在上面几行

        for (int i = c; i < n + 1; i++) swap(a[t][i], a[r][i]);/// 把当前这一行，换到最上面
        (不是第一行，是第 r 行) 去
        for (int i = n; i >= c; i--) a[r][i] /= a[r][c];// 把当前这一行的第一个数，变成 1，
        方程两边同时除以 第一个数，必须要到着算，不然第一个数直接变1，系数就被篡改，后面的数字没法算
        for (int i = r + 1; i < n; i++)// 把当前列下面的所有数，全部消成 0
        if (fabs(a[i][c]) > eps)// 如果非0 再操作，已经是 0就没必要操作了
            for (int j = n; j >= c; j--)// 从后往前，当前行的每个数字，都减去对应列
* 行首非0的数字，这样就能保证第一个数字是 a[i][0] -= 1*a[i][0];
            a[i][j] -= a[r][j] * a[i][c];

        r ++ ;// 这一行的工作做完，换下一行
    }

    if (r < n)// 说明剩下方程的个数是小于 n 的，说明不是唯一解，判断是无解还是无穷多解
    {// 因为已经是阶梯型，所以 r ~ n-1 的值应该都为 0
        for (int i = r; i < n; i++)
        if (fabs(a[i][n]) > eps)// a[i][n] 代表 b_i ,即 左边=0，右边=b_i, 0 != b_i， 所以
        无解。
        return 2;
        return 1;// 否则， 0 = 0，就是r ~ n-1的方程都是多余方程
    }
    // 唯一解 ↓，从下往上回代，得到方程的解
    for (int i = n - 1; i >= 0; i--)
        for (int j = i + 1; j < n; j++)
            a[i][n] -= a[j][n] * a[i][j];//因为只要得到解，所以只用对 b_i 进行操作，中间的
    值，可以不用操作，因为不用输出

    return 0;
}

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

    int t = gauss();

    if(t == 0) // 有唯一解
    {
        for (int i = 0; i < n; i++)
        {
            if(fabs(a[i][n]) < eps) a[i][n] = 0.00; // 避免输出-0.00
            printf("%.2lf\n", a[i][n]);
    }
}

```

```

        }
    }

    else if (t == 1) puts("Infinite group solutions");
    else puts("No solution");

    return 0;
}

```

求组合数

H5 递推打表，模数 $1e9+7$



```

// c[a][b] 表示从a个苹果中选b个的方案数
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]) % mod;

```

通过预处理逆元的方式求组合数（模数是质数，如 $1e9+7$ ）



首先预处理出所有阶乘取模的余数fact[N]，以及所有阶乘取模的逆元infact[N]

如果取模的数是质数，可以用费马小定理求逆元

```

int qmi(int a, int k, int p)      // 快速幂模板
{
    int res = 1;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

// 预处理阶乘的余数和阶乘逆元的余数
fact[0] = infact[0] = 1;
for (int i = 1; i < N; i++)
{
    fact[i] = (LL)fact[i - 1] * i % mod;
    infact[i] = (LL)infact[i - 1] * qmi(i, mod - 2, mod) % mod;
}

```

H5 Lucas定理

若p是质数，则对于任意整数 $1 \leq m \leq n$, 有:

$$C(n, m) = C(n \% p, m \% p) * C(n / p, m / p) \pmod{p}$$

```
int qmi(int a, int k, int p) // 快速幂模板
{
    int res = 1 % p;
    while (k)
    {
        if (k & 1) res = (LL)res * a % p;
        a = (LL)a * a % p;
        k >>= 1;
    }
    return res;
}

int C(int a, int b, int p) // 通过定理求组合数C(a, b)
{
    if (a < b) return 0;

    LL x = 1, y = 1; // x是分子, y是分母
    for (int i = a, j = 1; j <= b; i --, j ++ )
    {
        x = (LL)x * i % p;
        y = (LL) y * j % p;
    }

    return x * (LL)qmi(y, p - 2, p) % p;
}

int lucas(LL a, LL b, int p)
{
    if (a < p && b < p) return C(a, b, p);
    return (LL)C(a % p, b % p, p) * lucas(a / p, b / p, p) % p;
}
```

H5 分解质因数法求组合数

当我们需要求出组合数的真实值，而非对某个数的余数时，分解质因数的方式比较好用：

1. 筛法求出范围内的所有质数
2. 通过 $C(a, b) = a! / b! / (a - b)!$ 这个公式求出每个质因子的次数。 $n!$ 中p的次数是 $n / p + n / p^2 + n / p^3 + \dots$
3. 用高精度乘法将所有质因子相乘

```
//没有说要取模
int primes[N], cnt; // 存储所有质数
int sum[N]; // 存储每个质数的次数
bool st[N]; // 存储每个数是否已被筛掉

void get_primes(int n) // 线性筛法求素数
{
```

```

        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;
        }
    }
}

int get(int n, int p)           // 求n! 中的次数
{
    int res = 0;
    while (n)
    {
        res += n / p;
        n /= p;
    }
    return res;
}

vector<int> mul(vector<int> a, int b)           // 高精度乘低精度模板
{
    vector<int> c;
    int t = 0;
    for (int i = 0; i < a.size(); i++)
    {
        t += a[i] * b;
        c.push_back(t % 10);
        t /= 10;
    }

    while (t)
    {
        c.push_back(t % 10);
        t /= 10;
    }

    return c;
}

get_primes(a); // 预处理范围内的所有质数

for (int i = 0; i < cnt; i++)           // 求每个质因数的次数
{
    int p = primes[i];
    sum[i] = get(a, p) - get(b, p) - get(a - b, p);
}

```

```

vector<int> res;
res.push_back(1);

for (int i = 0; i < cnt; i++) // 用高精度乘法将所有质因子相乘
    for (int j = 0; j < sum[i]; j++)
        res = mul(res, primes[i]);

```

卡特兰数

给定n个0和n个1，它们按照某种顺序排成长度为2n的序列，满足任意前缀中0的个数都不少于1的个数的序列的数量为： $\text{Cat}(n) = C(2n, n) / (n + 1)$

容斥原理

容斥原理：

$$\begin{aligned}
& |s_1 \cup s_2 \cup s_3 \cup \dots \cup s_n| \\
&= |s_1| + |s_2| + |s_3| + \dots + |s_n| - (|s_1 \cap s_2| + |s_1 \cap s_3| + \dots + |s_{n-1} \cap s_n|) \\
&\quad + (|s_1 \cap s_2 \cap s_3| + |s_1 \cap s_2 \cap s_4| + \dots + |s_{n-2} \cap s_{n-1} \cap s_n|) - \dots - (-1)^n |s_1 \cap s_2 \cap \dots \cap s_n|
\end{aligned}$$

例题



```

#include<iostream>
using namespace std;
typedef long long LL;

const int N = 20;
int p[N], n, m;

int main() {
    cin >> n >> m;
    for(int i = 0; i < m; i++) cin >> p[i];

    int res = 0;
    // 枚举从1到 1111... (m个1) 的每一个集合状态，(至少选中一个集合)
    for(int i = 1; i < 1 << m; i++) {
        int t = 1; // 选中集合对应质数的乘积
        int s = 0; // 选中的集合数量

        // 枚举当前状态的每一位
        for(int j = 0; j < m; j++) {
            // 选中一个集合
            if(i >> j & 1) {
                // 乘积大于n，则n/t = 0，跳出这轮循环
                if((LL)t * p[j] > n) {

```

```

        t = -1;
        break;
    }
    s++; //有一个1，集合数量+1
    t *= p[j];
}
}

if(t == -1) continue;

if(s & 1) res += n / t; //选中奇数个集合，则系数应该是1，n/t为当前
这种状态的集合数量
else res -= n / t; //反之则为 -1
}

cout << res << endl;
return 0;
}

```

博弈论

H5 NIM游戏

给定N堆物品，第*i*堆物品有A_i个。两名玩家轮流行动，每次可以任选一堆，取走任意多个物品，可把一堆取光，但不能不取。取走最后一件物品者获胜。两人都采取最优策略，问先手是否必胜。

我们把这种游戏称为NIM博弈。把游戏中面临的状态称为局面。整局游戏第一个行动的称为先手，第二个行动的称为后手。若在某一局面下无论采取何种行动，都会输掉游戏，则称该局面必败。

所谓采取最优策略是指，若在某一局面下存在某种行动，使得行动后对面面临必败局面，则优先采取该行动。同时，这样的局面被称为必胜。我们讨论的博弈问题一般都只考虑理想情况，即两人均无失误，都采取最优策略行动时游戏的结果。

NIM博弈不存在平局，只有先手必胜和先手必败两种情况。

定理：NIM博弈先手必胜，当且仅当 A₁ ^ A₂ ^ ... ^ A_n != 0

H5 公平组合游戏ICG

若一个游戏满足：

1. 由两名玩家交替行动；
2. 在游戏进程的任意时刻，可以执行的合法行动与轮到哪名玩家无关；
3. 不能行动的玩家判负；

则称该游戏为一个公平组合游戏。

NIM博弈属于公平组合游戏，但城建的棋类游戏，比如围棋，就不是公平组合游戏。因为围棋交战双方分别只能落黑子和白子，胜负判定也比较复杂，不满足条件2和条件3。

H5 有向图游戏

给定一个有向无环图，图中有一个唯一的起点，在起点上放有一枚棋子。两名玩家交替地把这枚棋子沿有向边进行移动，每次可以移动一步，无法移动者判负。该游戏被称为有向图游戏。

任何一个公平组合游戏都可以转化为有向图游戏。具体方法是，把每个局面看成图中的一个节点，并且从每个局面向沿着合法行动能够到达的下一个局面连有向边。

H5 Mex运算

设S表示一个非负整数集合。定义mex(S)为求出不属于集合S的最小非负整数的运算，即：

$mex(S) = \min\{x\}$, x 属于自然数，且 x 不属于S

H5 SG函数

在有向图游戏中，对于每个节点x，设从x出发共有k条有向边，分别到达节点 y_1, y_2, \dots, y_k ，定义 $SG(x)$ 为x的后继节点 y_1, y_2, \dots, y_k 的SG函数值构成的集合再执行mex(S)运算的结果，即：

$SG(x) = mex(\{SG(y_1), SG(y_2), \dots, SG(y_k)\})$

特别地，整个有向图游戏G的SG函数值被定义为有向图游戏起点s的SG函数值，即 $SG(G) = SG(s)$ 。

例题

有向图游戏的和

设 G_1, G_2, \dots, G_m 是m个有向图游戏。定义有向图游戏G，它的行动规则是任选某个有向图游戏 G_i ，并在 G_i 上行动一步。 G 被称为有向图游戏 G_1, G_2, \dots, G_m 的和。

有向图游戏的和的SG函数值等于它包含的各个子游戏SG函数值的异或和，即：

$SG(G) = SG(G_1) \wedge SG(G_2) \wedge \dots \wedge SG(G_m)$

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

using namespace std;

const int N=110, M=10010;
int n, m;
int f[M], s[N];//s存储的是可供选择的集合, f存储的是所有可能出现过的情况的sg值

int sg(int x)
{
    if(f[x]!=-1) return f[x];
    //因为取石子数目的集合是已经确定了的, 所以每个数的sg值也都是确定的, 如果存储过了, 直接返回即可
    set<int> S;
    //set代表的是有序集合(注:因为在函数内部定义, 所以下一次递归中的S不与本次相同)
    for(int i=0; i<m; i++)
    {
        int sum=s[i];
        if(x>=sum) S.insert(sg(x-sum));
        //先延伸到终点的sg值后, 再从后往前排查出所有数的sg值
    }

    for(int i=0;; i++)
    //循环完之后可以进行选出最小的没有出现的自然数的操作
    if(!S.count(i))
        return f[x]=i;
}

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

```

for(int i=0;i<m;i++)
    cin>>s[i];

cin>>n;
memset(f,-1,sizeof(f)); //初始化f均为-1,方便在sg函数中查看x是否被记录过

int res=0;
for(int i=0;i<n;i++)
{
    int x;
    cin>>x;
    res^=sg(x);
    //观察异或值的变化,基本原理与Nim游戏相同
}

if(res) printf("Yes");
else printf("No");

return 0;
}

```

H5 树上博弈



树上博弈加换根



换根后就要考虑公式如何转化，推出转化方程



```

const int N=2e5+10,mod=1e9+7;
int f1[N],f2[N];
int h[N],e[N*2],ne[N*2],idx;
int n,q;
void add(int a,int b){
    e[idx]=b,ne[idx]=h[a],h[a]=idx++;
}
//求 m^k mod p, 时间复杂度 O(logk)。
//利用二进制
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;
    }

void dfs1(int u, int fa) {
    int now=0;
    for(int i=h[u]; ~i; i=ne[i]) {
        int v=e[i];
        if(v==fa) continue;
        dfs1(v, u);
        now^=(f1[v]-1);
    }
    f1[u]=now;
}

void dfs2(int u, int fa) {
    for(int i=h[u]; ~i; i=ne[i]) {
        int v=e[i];
        if(v==fa) continue;
        f2[v]=f1[v]^((f2[u]^ (f1[v]+1))+1);
        dfs2(v, u);
    }
}
void solve() {
    cin>>n;
    memset(h, -1, sizeof h);
    idx=0;
    for(int i=0; i<n-1; i++) {
        int u, v;
        cin>>u>>v;
        add(u, v), add(v, u);
    }
    dfs1(1, -1);
    f2[1]=f1[1];
    dfs2(1, -1);
    int cnt=0;
    for(int i=1; i<=n; i++) {
        if(f2[i]) cnt++;
    }
    //模数是质数，直接用快速幂求逆元得到概率
    int ans=(111*cnt*qmi(n, mod-2, mod)+mod)%mod;
    cout<<ans<<'\n';
}
}

```

h5 补充

有向图游戏的某个局面必胜，当且仅当该局面对应节点的SG函数值大于0。
有向图游戏的某个局面必败，当且仅当该局面对应节点的SG函数值等于0。

> 动态规划

复杂度计算：状态数量*状态计算量

数字三角形模型（路径dp）

LIS模型

H5 O(n^2)解法

```
void solve() {
    cin>>n;
    for(int i=1;i<=n;i++)
        cin>>a[i];
    for(int i=1;i<=n;i++) {
        dp[i]=1;
        for(int j=1;j<i;j++) {
            if(a[i]>a[j]) {
                dp[i]=max(dp[i], dp[j]+1);
            }
        }
    }
    int ans=-INF;
    for(int i=1;i<=n;i++) {
        ans=max(ans, dp[i]);
    }
    cout<<ans<<endl;
}
```

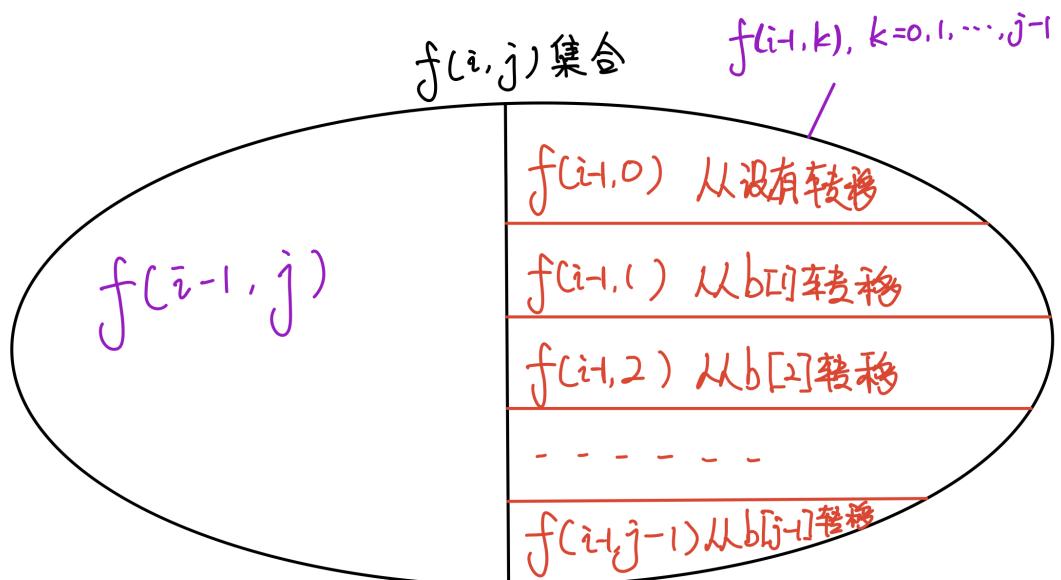
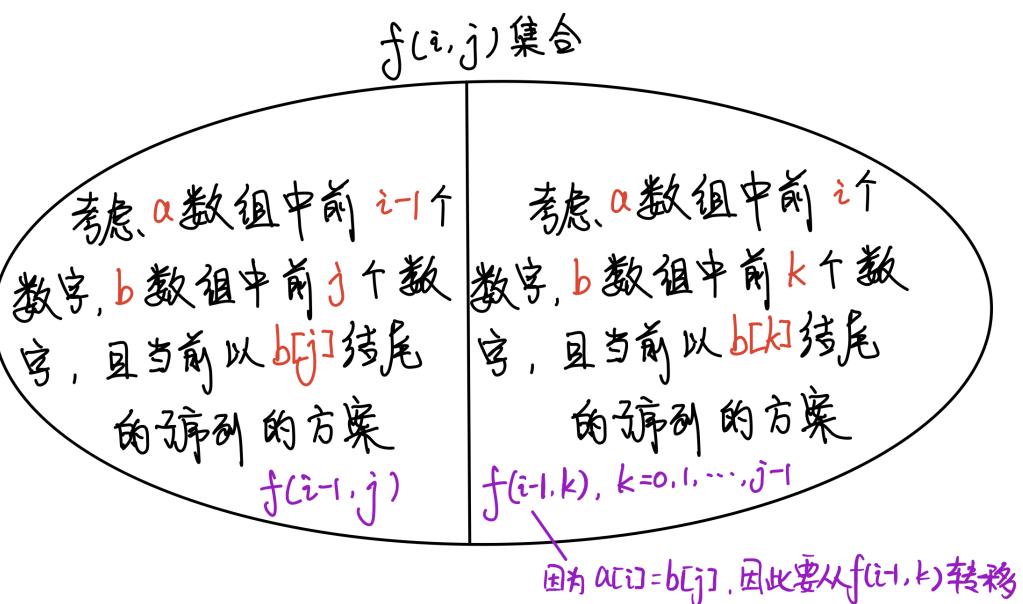
H5 O(nlogn)解法

```
void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) cin>>a[i];
    int len=1;q[len]=a[1];
    for(int i=2;i<=n;i++) {
        if(a[i]>q[len]) {
            q[++len]=a[i];
        } else{
            int pos=lower_bound(q+1, q+len+1, a[i])-q;
            q[pos]=a[i];
        }
    }
    cout<<len<<' \n' ;
}
```

LCS模型

```
void solve() {
    cin>>n>>m;
    cin>>a+1>>b+1;
    for(int i=1;i<=n;i++) {
        for(int j=1;j<=m;j++) {
            dp[i][j]=max(dp[i-1][j], dp[i][j-1]);
            if(a[i]==b[j]) {
                dp[i][j]=max(dp[i][j], dp[i-1][j-1]+1);
            }
        }
    }
    cout<<dp[n][m]<<endl;
}
```

LCS+LIS



朴素版本

```

void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) cin>>a[i];
    for(int i=1;i<=n;i++) cin>>b[i];
    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]) {
                f[i][j]=max(f[i][j], 1);
                for(int k=1;k<j;k++) {
                    if(b[k]<b[j]) f[i][j]=max(f[i][j], f[i-1][k]+1);
                }
            }
        }
    }
    int ans=0;
    for(int i=1;i<=n;i++) ans=max(ans, f[n][i]);
    cout<<ans<<'\n';
}

```



优化版本

```

void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) cin>>a[i];
    for(int i=1;i<=n;i++) cin>>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(b[j]<a[i]) maxv=max(maxv, f[i-1][j]+1);
        }
    }
    int ans=0;
    for(int i=1;i<=n;i++) ans=max(ans, f[n][i]);
    cout<<ans<<'\n';
}

```

背包问题



01背包：物品只选一次 完全背包：物品选无数次 多重背包：物品选有限次 分组背包：每组中选一个物品

H5 01背包

朴素版本

```
void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>v[i]>>w[i];
    for(int i=1;i<=n;i++) {
        for(int j=0;j<=m;j++) {
            f[i][j]=f[i-1][j];
            if(j>=v[i]) f[i][j]=max(f[i][j], f[i-1][j-v[i]]+w[i]);
        }
    }
    cout<<f[n][m]<<' \n' ;
}
```

优化版本

```
void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>v[i]>>w[i];
    for(int i=1;i<=n;i++) {
        for(int j=m;j>=v[i];j--) {
            f[j]=max(f[j], f[j-v[i]]+w[i]);
        }
    }
    cout<<f[m]<<' \n' ;
}
```

H5 完全背包



推出 $f[i, j] = \max(f[i-1, j], f[i, j-v]+w)$

```
void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++)
        cin>>v[i]>>w[i];
    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;
}
```

注：与01背包优化版本的不同是完全背包体积从小到大枚举，01背包体积从大到小枚举

H5 多重背包

H6 二进制优化版本



```
const int N=1e5+10, M=2010;
int n, m, v[N], w[N], f[M];
void solve() {
    cin>>n>>m;
    int cnt=0;
    for(int i=1;i<=n;i++) {
        int a, b, c;cin>>a>>b>>c;
        int k=1;
        while(k<=c) {
            cnt++;v[cnt]=a*k;w[cnt]=b*k;
            c-=k;
            k*=2;
        }
        if(c) {
            cnt++;v[cnt]=a*c;w[cnt]=b*c;
        }
    }
    n=cnt;
    //01背包板子
    for(int i=1;i<=n;i++) {
        for(int j=m;j>=v[i];j--) {
            f[j]=max(f[j], f[j-v[i]]+w[i]);
        }
    }
    cout<<f[m]<<'\n';
}
```

H6 单调队列优化版本

二维朴素版本

```
const int N=1010, M=2e4+10;
int n, m, v[N], w[N], s[N], q[M], f[N][M];
void solve() {
    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 r=0;r<v[i];r++) {
            int hh=0, tt=-1;
            for(int j=r;j<=m;j+=v[i]) {
                while(hh<=tt && j-q[hh]>s[i]*v[i]) hh++;
                while(hh<=tt && f[i-1][q[tt]]+(j-q[tt])/v[i]*w[i]<=f[i-1][j]) tt--;
                q[++tt]=j;
                f[i][j]=f[i-1][q[hh]]+(j-q[hh])/v[i]*w[i];
            }
        }
    }
}
```

```

        }
    }
}

cout<<f[n][m]<<' \n' ;
}

```

一维优化版本(仅与前一层有关, 滚动数组优化)

```

const int N=1010, M=2e4+10;
int n, m, v[N], w[N], s[N], q[M], f[2][M];
void solve() {
    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 r=0;r<v[i];r++) {
            int hh=0, tt=-1;
            for(int j=r;j<=m;j+=v[i]) {
                while(hh<=tt && j-q[hh]>s[i]*v[i]) hh++;
                while(hh<=tt && f[(i-1)&1][q[tt]]+(j-q[tt])/v[i]*w[i]<=f[(i-1)&1][j]) tt--;
                q[++tt]=j;
                f[i&1][j]=f[(i-1)&1][q[hh]]+(j-q[hh])/v[i]*w[i];
            }
        }
    }
    cout<<f[n&1][m]<<' \n' ;
}

```

H5 分组背包

```

int v[N][N], w[N][N], s[N];
int dp[N];
int n, m;
void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) {
        cin>>s[i];
        for(int j=1;j<=s[i];j++) {
            cin>>v[i][j]>>w[i][j];
        }
    }
    for(int i=1;i<=n;i++) { //枚举组
        for(int j=m;j>=0;j--) { //枚举体积
            for(int k=0;k<=s[i];k++) { //枚举决策
                if(j>=v[i][k]) {
                    dp[j]=max(dp[j], dp[j-v[i][k]]+w[i][k]);
                }
            }
        }
    }
    cout<<dp[m]<<endl;
}

```

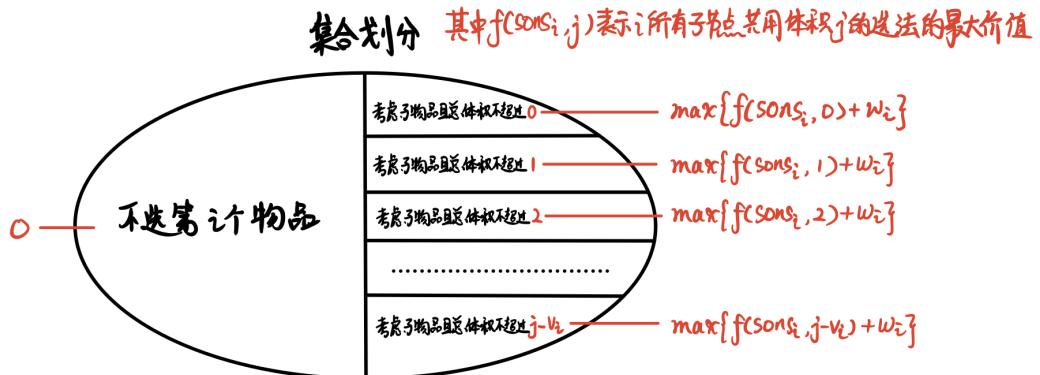
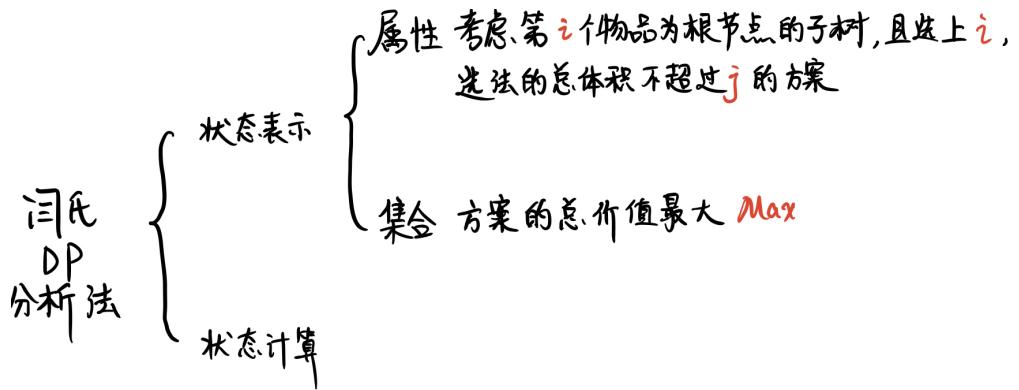
```
}
```

H5 混合背包问题

```
int f[N], n, m;
void solve() {
    cin>>n>>m;
    for(int i=1; i<=n; i++) {
        int v, w, s;
        cin>>v>>w>>s;
        //s为0是完全背包
        if(s==0) {
            for(int j=v; j<=m; j++) {
                f[j]=max(f[j], f[j-v]+w);
            }
        } else {
            //s为-1为01背包，01背包是特殊的多重背包
            //s>0表示多重背包，s表示个数
            if(s== -1) s=1;
            int k=1;
            while(k<=s) {
                for(int j=m; j>=v*k; j--) {
                    f[j]=max(f[j], f[j-v*k]+w*k);
                }
                s-=k;
                k*=2;
            }
            if(s) {
                for(int j=m; j>=v*s; j--) {
                    f[j]=max(f[j], f[j-v*s]+w*s);
                }
            }
        }
    }
    cout<<f[m]<<'\n';
}
```

H5 有依赖的背包问题

物品之间的依赖以树的形式连接



```

vector<int> g[N];
int f[N][N];
int n, m, v[N], w[N];
void dfs(int u) {
    for (int j=v[u]; j<=m; j++) f[u][j]=w[u];
    for (int i=0; i<g[u].size(); i++) {
        dfs(g[u][i]);
        for (int j=m; j>=v[u]; j--) {
            for (int k=0; k<=j-v[u]; k++) {
                f[u][j]=max(f[u][j], f[u][j-k]+f[g[u][i]][k]);
            }
        }
    }
}
void solve() {
    cin>>n>>m;
    int root;
    for (int i=1; i<=n; i++) {
        int p;
        cin>>v[i]>>w[i]>>p;
        if (p==1) root=i;
        else g[p].push_back(i);
    }
    dfs(root);
    cout<<f[root][m]<<'\n';
}

```

H5 二维费用的背包问题

以01背包为例，其他背包类似

```
int n, V, M;
int f[N][N];
void solve() {
    cin>>n>>V>>M;
    for(int i=1; i<=n; i++) {
        int v, m, w;
        cin>>v>>m>>w;
        for(int j=V; j>=v; j--) {
            for(int k=M; k>=m; k--) {
                f[j][k]=max(f[j][k], f[j-v][k-m]+w);
            }
        }
    }
    cout<<f[V][M]<<'\n';
}
```

H5 求方案数的背包问题

求所有满足最大价值的方案数

```
const int N=1010, mod=1e9+7;
int n, m;
int f[N], g[N];
//用g数组存储数量
void solve() {
    cin>>n>>m;
    g[0]=1;
    for(int i=1; i<=n; i++) {
        int v, w;
        cin>>v>>w;
        for(int j=m; j>=v; j--) {
            int maxv=max(f[j], f[j-v]+w);
            int cnt=0;
            if(maxv==f[j]) cnt+=g[j];
            if(maxv==f[j-v]+w) cnt+=g[j-v];
            f[j]=maxv;
            g[j]=cnt%mod;
        }
    }
    int maxv=0;
    for(int i=0; i<=m; i++) {
        maxv=max(maxv, f[m]);
    }
    int ans=0;
    for(int i=0; i<=m; i++) {
        if(f[i]==maxv) {
            ans+=g[i];
            ans%=mod;
        }
    }
}
```

```
    }
    cout<<ans<<' \n' ;
}
```

H5 求具体方案的背包问题

记录方案的过程就是记录每一次决策的过程

```
// 求具体方案且按字典序最小输出
int n, m, f[N][N], v[N], w[N];
void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) cin>>v[i]>>w[i];
    for(int i=n;i>=1;i--) {
        for(int j=0;j<=m;j++) {
            f[i][j]=f[i+1][j];
            if(j>=v[i]) f[i][j]=max(f[i][j], f[i+1][j-v[i]]+w[i]);
        }
    }
    int j=m;
    for(int i=1;i<=n;i++) {
        if(j>=v[i] && f[i][j]==f[i+1][j-v[i]]+w[i]) {
            cout<<i<<' ';
            j-=v[i];
        }
    }
    cout<<' \n' ;
}
```

状态机模型

特点：描述的是一个过程，不止有选还是不选，状态不好考虑的时候，可以把状态根据题目所给的限制条件再分离为多个状态进行转移（通过添加维度划分类型），从而简化问题，更容易理解和思考，在状态机模型中我们强调并记录当前事物的状态，当某个事件的状态影响到后面事件的决策的时候，我们需要在状态定义的时候记录事件状态，即状态机DP

状态压缩

状态压缩：通常用一个二进制数表示状态，用每一位上的数值代表对应物品的选择(放置/经过/...)情况。

H5 棋盘式（基于连通性）

通常是在棋盘/方格中摆放物品，且一个物品对周围其他物品的摆放有限制

有时为了优化空间可以用滚动数组

玉米田，求方案数

```
const int N=14, mod=1e8;
int n, m, f[N][1<<N];
vector<int> state;//存储合法状态
vector<int> head[1<<N];//存储状态可以由哪些其他状态转换而来
int g[N];//存储荒废的土地（不能种玉米）的状态
```

```

bool check(int x) {
    //玉米地不能相邻
    for(int i=0;i<m;i++) {
        if((x>>i&1) && (x>>(i+1))&1) {
            return false;
        }
    }
    return true;
}

void solve() {
    cin>>n>>m;
    for(int i=1;i<=n;i++) {
        for(int j=1;j<=m;j++) {
            int x;cin>>x;
            if(!x) g[i]+=1<<(j-1);
        }
    }
    for(int i=0;i<1<<m;i++) {
        if(check(i)) {
            state.push_back(i);
        }
    }
    for(int i=0;i<state.size();i++) {
        for(int j=0;j<state.size();j++) {
            int a=state[i],b=state[j];
            //上下两行不相交
            if(! (a&b)) {
                head[a].push_back(b);
            }
        }
    }
    f[0][0]=1;
    for(int i=1;i<=n+1;i++) {
        for(int j=0;j<state.size();j++) {
            int a=state[j];
            if(a&g[i]) continue;
            for(int k=0;k<head[a].size();k++) {
                int b=head[a][k];
                f[i][a]=(f[i][a]+f[i-1][b])%mod;
            }
        }
    }
    //上面多枚举了一行，偷懒求答案时不需要再枚举
    cout<<f[n+1][0]<<'\n';
}

```

预处理出满足摆放限制的合法状态，并且还要预处理出在限制条件下，哪些状态之间可以进行转移，dp乱搞一下

H5 集合

维护某些元素是否在集合当中

区间DP

一般用两种方式实现，一是迭代式（一维常用），二是记忆化搜索

H5 环形区间

将环转化成链，常用优化方法：复制一遍添在数组的后面，达成链的效果

环形石子合并，求合并的最小花费和最大花费

```
const int N=410, INF=0x3f3f3f3f;
int f[N][N], g[N][N];
//f求合并的最大值，g求合并的最小值
int n, w[N], s[N];
void solve() {
    cin>>n;
    //复制一遍接在后面
    for(int i=1;i<=n;i++) {
        cin>>w[i];
        w[i+n]=w[i];
    }
    //前缀和
    for(int i=1;i<=n*2;i++) {
        s[i]=s[i-1]+w[i];
    }
    memset(f,-0x3f,sizeof f);
    memset(g,0x3f,sizeof g);
    //枚举区间，然后枚举左端点，得到右端点后枚举l~r之间的合并位置
    for(int len=1;len<=n;len++) {
        for(int l=1;l+len-1<=n*2;l++) {
            int r=l+len-1;
            if(len==1) g[1][r]=f[1][r]=0;//长度为1不需要合并，花费为0
            else{
                for(int k=l;k<r;k++) {
                    f[1][r]=max(f[1][r], f[1][k]+f[k+1][r]+s[r]-s[l-1]);
                    g[1][r]=min(g[1][r], g[1][k]+g[k+1][r]+s[r]-s[l-1]);
                }
            }
        }
    }
    int maxv=-INF, minv=INF;
    for(int i=1;i<=n;i++) {
        maxv=max(maxv, f[i][i+n-1]);
        minv=min(minv, g[i][i+n-1]);
    }
    cout<<minv<<' \n' <<maxv<<' \n';
}
```

H5 记录方案数

加分二叉树



```
const int N=35;
int f[N][N], g[N][N];//g数组存储路径
int n, w[N];
void dfs(int l, int r) {
    if(l>r) return ;
    int root=g[l][r];
    cout<<root<<' ';
    dfs(l, root-1);
    dfs(root+1, r);
}
void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) cin>>w[i];
    for(int len=1;len<=n;len++) {
        for(int l=1;l+len-1<=n;l++) {
            int r=l+len-1;
            if(len==1) {
                f[1][r]=w[1];
                g[1][r]=1;
            } else {
                for(int k=l;k<=r;k++) {
                    int left=((k==l)?1:f[1][k-1]);
                    int right=((k==r)?1:f[k+1][r]);
                    if(left*right+w[k]>f[1][r]) {
                        f[1][r]=left*right+w[k];
                        g[1][r]=k;
                    }
                }
            }
        }
    }
    cout<<f[1][n]<<' \n' ;
    dfs(1, n);
}
```

H5 区间DP+高精度

凸边形的划分

```
const int N=55;
vector<int> f[N][N];
int n, w[N];
bool cmp(vector<int> a, vector<int> b) {
    if(a.size()!=b.size()) return a.size()<b.size();
    for(int i=a.size()-1;i>=0;i--) {
        if(a[i]!=b[i]) return a[i]<b[i];
    }
}
```

```

        return true;
    }

vector<int> add(vector<int> a, vector<int> b) {
    if(b.size()>a.size()) return add(b,a);
    int t=0;
    vector<int> c;
    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%10);
    return c;
}

vector<int> mul(vector<int> a, ll b) {
    vector<int> c;
    ll t=0;
    for(int i=0;i<a.size();i++) {
        t+=a[i]*b;
        c.push_back(t%10);
        t/=10;
    }
    while(t) {
        c.push_back(t%10);
        t/=10;
    }
    return c;
}

void solve() {
    cin>>n;
    for(int i=1;i<=n;i++) cin>>w[i];
    for(int len=3;len<=n;len++) {
        for(int l=1;l+len-1<=n;l++) {
            int r=l+len-1;
            for(int k=l+1;k<r;k++) {
                auto t=mul({w[l]},w[k]),w[r]);
                t=add(add(t,f[l][k]),f[k][r]);
                if(f[l][r].empty() || cmp(t,f[l][r])) {
                    f[l][r]=t;
                }
            }
        }
    }
    for(int i=f[1][n].size()-1;i>=0;i--) {
        cout<<f[1][n][i];
    }
    cout<<' \n' ;
}

```

H5 二维区间DP+（二维前缀和）

棋盘分割

```
const int N=15, m=9;
int n;
double f[m][m][m][m][N];
int s[m][m];//二维前缀和
double X;//平均值
int get_sum(int x1, int y1, int x2, int y2) {
    return s[x2][y2]-s[x1-1][y2]-s[x2][y1-1]+s[x1-1][y1-1];
}
double get(int x1, int y1, int x2, int y2) {
    double sum=double(get_sum(x1, y1, x2, y2))-X;
    return double(sum*sum)/n;
}
double dp(int x1, int y1, int x2, int y2, int k) {
    double &v=f[x1][y1][x2][y2][k];
    if(v>=0) return v;
    if(k==1) return v=get(x1, y1, x2, y2);
    v=1e9;
    //横着切
    for(int i=x1;i<x2;i++) {
        v=min(v, get(x1, y1, i, y2)+dp(i+1, y1, x2, y2, k-1));
        v=min(v, get(i+1, y1, x2, y2)+dp(x1, y1, i, y2, k-1));
    }
    //竖着切
    for(int i=y1;i<y2;i++) {
        v=min(v, get(x1, y1, x2, i)+dp(x1, i+1, x2, y2, k-1));
        v=min(v, get(x1, i+1, x2, y2)+dp(x1, y1, x2, i, k-1));
    }
    return v;
}
void solve() {
    cin>>n;
    for(int i=1;i<=m-1;i++) {
        for(int j=1;j<=m-1;j++) {
            cin>>s[i][j];
            s[i][j]+=s[i-1][j]+s[i][j-1]-s[i-1][j-1];
        }
    }
    X=(double)s[m-1][m-1]/n;
    memset(f, -1, sizeof f);
    printf("%.3lf\n", sqrt(dp(1, 1, 8, 8, n)));
}
```

树形DP

h5 求树的直径

常用做法

image-20230620160731958

```
void dfs(int x, int father)
{
    for(int i=h[x]; i!=-1; i=ne[i]) //用链式前向星存树
    {
        int j=e[i];
        if(j==father) continue; //树是一种有向无环图，只要搜索过程中不返回父亲节点即可保证不会重复遍历同一个点。
        d[j]=d[x]+w[i]; //更新每个点到起始点的距离（在树中任意两点的距离是唯一的）
        dfs(j, x); //继续搜索下一个节点
    }
}
11 ans=-1;
for(int i=1; i<=n; i++)
{
    if(d[i]>ans)
    {
        ans=d[i];
        e=i; //记录与搜索点距离最远的点
    }
}
memset(d, 0, sizeof d);
dfs(e, 0);
for(int i=1; i<=n; i++)
{
    if(d[i]>ans)
        ans=d[i];
}
printf("%lld", ans);
```

树形dp求直径

树的最长路径

枚举每一点作为根节点时，求出子树的最大和次大路径值，这里边权值可以为负数，如果是普通的无边权的，边权设为1即可

```
const int N=1e4+10, M=N*2;
int h[N], e[M], ne[M], w[M], idx;
int n, ans;
void add(int a, int b, int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
int dfs(int u, int fa) {
    int d1=0, d2=0;
    int maxv=0; //表示往下走的最大距离
    for(int i=h[u]; ~i; i=ne[i]) {
        int j=e[i];
        if(j==fa) continue; //因为是无向边，为了防止重复访问造成死循环
```

```

        int d=dfs(j,u)+w[i];
        maxv=max(maxv,d);
        if(d>=d1) d2=d1, d1=d;
        else if(d>d2) d2=d;
    }
    //以u为根节点时，求得的两颗最大和次大的边权值的子树
    ans=max(ans,d1+d2);
    return maxv;
}
void solve() {
    cin>>n;
    memset(h,-1,sizeof h);
    for(int i=1;i<n;i++) {
        int a,b,c;cin>>a>>b>>c;
        add(a,b,c),add(b,a,c);
    }
    dfs(1,-1);
    cout<<ans<<'\n';
}

```

h5 树的中心

树的中心：该点到树中其他结点的最远距离最近

这里求的是树的中心到其他点的最远距离，若要求出树的中心，则遍历一遍判断哪个点即可

这个问题是树形DP的一类经典模型，常被称作换根DP



```

const int N=1e4+10, M=N*2, INF=0x3f3f3f3f;
int h[N], e[M], ne[M], w[M], idx;
int n, d1[N], d2[N], up[N], p1[N], p2[N];
//d1, d2表示每个点向下走的路径最大和次大值，up表示每个点向上走的路径最大值
//p1, p2: 存下i节点向下走的最长路径和次长是从哪一个节点下去的
void add(int a,int b,int c) {
    e[idx]=b, w[idx]=c, ne[idx]=h[a], h[a]=idx++;
}
int dfs_d(int u,int fa) {
    d1[u]=d2[u]=-INF;
    for(int i=h[u];~i;i=ne[i]) {
        int j=e[i];
        if(j==fa) continue;
        int d=dfs_d(j,u)+w[i];
        if(d>=d1[u]) {
            d2[u]=d1[u], p2[u]=p1[u];
            d1[u]=d, p1[u]=j;
        } else if(d>d2[u]) {
            d2[u]=d, p2[u]=j;
        }
    }
    if(d1[u]==-INF) d1[u]=d2[u]=0;
    return d1[u];
}

```

```

    }

void dfs_u(int u, int fa) {
    for(int i=h[u]; i<ne[i]; i++) {
        int j=e[i];
        if(j==fa) continue;
        if(j==p1[u]) up[j]=max(up[u], d2[u])+w[i];
        else up[j]=max(up[u], d1[u])+w[i];
        dfs_u(j, u);
    }
}

void solve() {
    cin>>n;
    memset(h, -1, sizeof h);
    for(int i=1; i<=n-1; i++) {
        int a, b, c;
        cin>>a>>b>>c;
        add(a, b, c), add(b, a, c);
    }
    dfs_d(1, -1);
    dfs_u(1, -1);
    int ans=INF;
    for(int i=1; i<=n; i++) {
        ans=min(ans, max(d1[i], up[i]));
    }
    cout<<ans<<'\n';
}

```

H5 滑稽总结

树形DP总结：根据题目分析出性质，将题目抽象成树形DP，根据题目要求的限制，用状态机的思想分离出多种状态类型，然后跑dfs枚举所有情况，常用邻接表存储树，也可以用vector存储，常常是枚举根节点，然后分左边和右边？上面和下面？选或者不选？

数位DP

问题特点：常常问的是在某一个区间里面，满足某种性质的数的个数（在限制条件下的满足的个数）

对应的数位dp问题有相应的解题技巧：

1.利用前缀和，比如求区间[x,y]中的个数，转化成求[0,y]的个数-[0,x-1]的个数。

2.利用树的结构来考虑（按位分类讨论）

H5 一般做法(伪代码)

```

int dfs(int pos, int pre, int lead, int limit) {
    if (!pos) {
        边界条件
    }
    if (!limit && !lead && dp[pos][pre] != -1) return dp[pos][pre];
    int res = 0, up = limit ? a[pos] : 无限制位;
    for (int i = 0; i <= up; i++) {
        if (不合法条件) continue;

```

```

        res += dfs(pos - 1, 未定参数, lead && !i, limit && i == up);
    }
    return limit ? res : (lead ? res : dp[pos][sum] = res);
}

int cal(int x) {
    memset(dp, -1, sizeof dp);
    len = 0;
    while (x) a[++len] = x % 进制, x /= 进制;
    return dfs(len, 未定参数, 1, 1);
}

int main() {
    cin >> l >> r;
    cout << cal(r) - cal(l - 1) << endl;
}

```

单调队列优化

滑动窗口求最值 (求固定窗口内的区间最值)

斜率优化

一些做题的经验总结

> 最短路求双向距离

先从起点跑一遍dij求出所有点到起点的距离，然后转置矩阵，再跑一遍dij求起点到所有点的距离，此时求出来的距离实际上就是所有点到起点的最短距离。这样来回的距离就求出来了

POJ 3268



题意

有编号为1 - N的牛，它们之间存在一些单向的路径。给定一头牛的编号，其他牛要去拜访它并且拜访完之后要返回自己原来的位置，求这些牛中所花的最长的来回时间是多少。

```

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
#include <map>
#define ll long long
#define mfp make_pair
using namespace std;
typedef pair<int, int> PII;

```

```

const int N=1010, INF=0x3f3f3f3f;
int n, m, x;
int ans[N], dist[N], vis[N];
int g[N][N];
void dij(int s) {
    memset(vis, 0, sizeof vis);
    for(int i=1; i<=n; i++) {
        dist[i]=g[s][i];
    }
    dist[s]=0;
    vis[s]=1;
    for(int i=1; i<=n; i++) {
        int mn=INF, pos=0;
        for(int j=1; j<=n; j++) {
            if(!vis[j] && mn>dist[j]) {
                mn=dist[j], pos=j;
            }
        }
        vis[pos]=1;
        for(int j=1; j<=n; j++) {
            dist[j]=min(dist[j], dist[pos]+g[pos][j]);
        }
    }
}
void solve() {
    cin>>n>>m>>x;
    // n个点, m条单向边, x为所有牛都要到的点
    memset(g, INF, sizeof g);
    for(int i=1; i<=m; i++) {
        int a, b, c;
        cin>>a>>b>>c;
        g[a][b]=c;
    }
    dij(x);
    for(int i=1; i<=n; i++) {
        ans[i]=dist[i];
    }
    // 转置矩阵
    for(int i=1; i<=n; i++) {
        for(int j=1; j<i; j++) {
            swap(g[i][j], g[j][i]);
        }
    }
    dij(x);
    int res=0;
    for(int i=1; i<=n; i++) {
        ans[i]+=dist[i];
        res=max(res, ans[i]);
    }
    cout<<res<<'\n';
}
int main() {

```

```

ios::sync_with_stdio(0); cin.tie(0), cout.tie(0);
int T=1;
// cin>>T;
while(T--) {
    solve();
}
return 0;
}

```

> 算贡献

通过计算每段的贡献然后加在一起得到答案，有时不一定是正着算贡献，也可以先算出总共的然后减去所有没有价值的从而得到最后的有价值的贡献

例题

h5 D題 Petya, Petya, Petr, and Palindromes



其中 $(t+i)/2 \geq 1+k/2$, $(t+i)/2 \leq n-k/2$ 指的是中点坐标合法

```

#include <bits/stdc++.h>
#define int long long
using namespace std;
const int N=2e5+10;
vector<int> v[N][2];
int n, k;
int cal(vector<int> &a, int l, int r) {
    return upper_bound(a.begin(), a.end(), r)-lower_bound(a.begin(), a.end(), l);
}
void solve() {
    cin>>n>>k;
    int x;
    for(int i=1; i<=n; i++) {
        cin>>x;
        v[x][i&1].push_back(i);
        //x值在分别在奇偶位置时的位置
    }
    if(k==1) {
        cout<<0<<endl;
        return ;
    }
    int ans=(n-(k-1))*(k/2);

```

```

for(int i=1;i<=2e5;i++) { //注: 写成i<=N, 没有输出
    for(int j=0;j<=1;j++) {
        for(auto t:v[i][j]) {
            //判断合法区间
            int l=max(1LL, max(t-(k-1), 2+k/2*2-t));
            int r=min(t-1, 2*n-k/2*2-t);
            if(l<=r) {
                ans+=cal(v[i][j], l, r);
            }
        }
    }
    cout<<ans<<endl;
}
signed main() {
    ios::sync_with_stdio(0); cin.tie(0), cout.tie(0);
    int T=1;
//    cin>>T;
    while(T--) {
        solve();
    }
    return 0;
}

```

> 寻找树的直径

树的直径做法：先随便找一个点，跑一个 `dfs`，记录每一个点的深度，然后找到深度最大的那个点，他就是直径的一个端点，然后从这个点再跑一遍，找到深度最大的点，这个点就是另一个端点，两个点之间的距离就是直径

例题



```

#include <bits/stdc++.h>

using namespace std;
const int N=1e5+10;
int n, m;
int dist[N], d[N];
vector<int> e[N];

void dfs(int u, int fa) {
    dist[u] = dist[fa] + 1;
    for(auto it : e[u]) {
        if(it == fa) continue;
        dfs(it, u);
    }
}

```

```

void solve()
{
    int a, b;
    cin >> n;
    m = n - 1;
    for(int i = 1; i <= m; i++) {
        cin >> a >> b;
        e[a].push_back(b), e[b].push_back(a);
    }
    dfs(1, 1);
    int pos = 0, ma = 0;
    for(int i = 1; i <= n; i++) {
        if(dist[i] > ma) ma = dist[i], pos = i;
    }
    // 此时的 pos 是直径的一个端点
    dist[pos] = 0;
    dfs(pos, pos);
    ma = 0;
    for(int i = 1; i <= n; i++) {
        d[i] = dist[i];
        if(dist[i] > ma) ma = dist[i], pos = i;
    }
    // pos 是直径的另一个端点
    dist[pos] = 0;
    dfs(pos, pos);
    for(int i = 1; i <= n; i++) {
        d[i] = max(d[i], dist[i]) - 1;
    }
    // 每个点的距离我们取距直径两端的最大值
    int ans = 0, now = 1;
    sort(d + 1, d + n + 1);
    for(int i = 1; i <= n; i++) {
        while(now <= n && d[now] < i) {
            now++, ans++;
        }
        // 如果 d[now] < i, 说明这个点无法加入直径两端所在连通块, 所以连通块数量++
        cout << min(ans + 1, n) << "\n"[i == n];
        // ans 是有多少点是孤立点, 最后答案还要加上直径两端所在的连通块
    }
}

signed main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    int t = 1;
    while(t--) solve();
    return 0;
}

```

将问题转化为图来解决,分类讨论

H5 例题



题意

给定长度为 n 的数组 a ，且 $-n \leq a_i \leq n$ 。

你需要进行以下的游戏。在数轴上，如果当前位置为 i ：

- 当 $1 \leq i \leq n$ 时，下一步可以到 $i+a_i$ 。
- 否则，游戏结束。

你可以修改一个数组元素值（依然需要使得 $-n \leq a_i \leq n$ ，修改后的值和原先值相同也是可以的），使得从数轴上的 1 出发，能够在有限回合结束游戏。

分析

把每次转移看成一条边，把 终止条件 看作结点 0

我们观察组成的图一定满足以下性质.

- 所有能终止的结点一定在以 0 为根的树上.
- 所有无法终止的结点一定在某个基环树上.

然后讨论初始状态 1 能否终止

若初始条件 1 可以到达终点，那么显然 1 在以 0 为根的树上。首先除了 1 到 根节点 路径上的点，其他点随便修改 1 也能到达终点。

所以我们主要考虑 1 到根节点路径上的点，我们可以发现这些点连到本身的子树内时或者连到了其他无法终止的基环树上，条件就无法满足了，我们可以考虑用总数减去不满足要求的答案。

若初始条件 1 不可以到达终点，我们必须修改 1 本身或者 1 能到达的结点，这些结点只要能连到以 0 为根的树上就可以满足条件，找到所有 1 能到达的结点统计即可。

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
const int N=2e5+10;
bool vis[N];
vector<int> g[N];
int p[N], n, a[N], sz[N];
void dfs(int u) {
    sz[u]=1;
    for(auto t:g[u]){
        dfs(t);
        sz[u]+=sz[t];
    }
}
void solve() {
    cin>>n;
    for(int i=0;i<=n;i++){
        g[i].clear();
        sz[i]=0;
    }
}
```

```

        vis[i]=0;
    }
    for(int i=1;i<=n;i++) {
        cin>>a[i];
        int t=i+a[i];
        if(t<=0 || t>n) t=0;
        p[i]=t;
        g[t].push_back(i);
    }
    dfs(0);
    if(sz[1]) {
        int ans=n*(2*n+1);
        int u=1;
        while(u) {
            ans-=sz[u]+(n+1-sz[0]);
            u=p[u];
        }
        cout<<ans<<' \n' ;
    } else{
        int ans=0;
        int u=1;
        //遍历所有1路径上的点
        while(!vis[u]) {
            vis[u]=1;
            ans+=sz[0]-1+n+1;//可以选择sz[0]除了0上的点以及n+1个外面的点
            u=p[u];
        }
        cout<<ans<<' \n' ;
    }
}
signed main() {
    ios::sync_with_stdio(0);cin.tie(0),cout.tie(0);
    int T=1;
    cin>>T;
    while(T--) {
        solve();
    }
    return 0;
}

```

点与点之间有联系，通过建图利用搜索，然后分类讨论求解

一些不能犯的傻事（总结+笔记）

不要读错题意

千万不要理解错题意，多读几遍，发现思路对了但是怎么也A不了，那么有可能就是题意读错了，或者有些corner case没有处理，先看是否读错题意，若没有再去处理细节

VP

- VP时千万不要提前去搜题解，要养成自己独立思考的习惯，这样才能锻炼出解题的直觉
- 要想各种各样的思路和解法去解题，不要放弃思考

不要越界！

十年oi一场空，不开long long见祖宗，不开long long 是傻逼

观察数据范围，看是否超int，别白白wa一发，输不起罚时

不要过于慌张

看到题有许多人过了，且是个简单题，自己没有A出来，不要慌张，冷静思考，一定不要心急，一定一定不要心急

不要把一些题想的太难，可能就是暴力枚举

审题要仔细啊，傻逼

对于模拟题和暴力题

一定要仔细读题，尽可能的抓住所有的细节，考虑所有的细节和case corner，多拿几组数据测试一下，细节细节细节

输入输出

一定一定要注意输入和输出的格式，是换行输出还是隔着空格输出！！！

注意边界

如是否取等号，特殊情况等等

不要死磕

一道题暂时没有思路千万不要死磕，一种方法不行的时候要及时换一种方法

超时问题

如果没有超int就用int，不超就不用#define int long long，否则会慢许多从而导致TLE

千万不要停止思考

比赛还没有结束就不要停止思考，不要摆烂，想各种方法求解答案，正是这思考的过程才是培养思维直觉的过程

优化

优化往往是对代码进行优化，思路一般不变，对代码做等价变形

