DJ算法

// Dijkstra算法模版（洛谷）

// 静态空间实现 : 链式前向星 + 反向索引堆

// 测试链接 : https://www.luogu.com.cn/problem/P4779

// 请同学们务必参考如下代码中关于输入、输出的处理

// 这是输入输出处理效率很高的写法

// 提交以下所有代码，把主类名改成Main，可以直接通过

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 1005;

const int MAXM = 5005;

auto compare = [](const pair<int, int>& left, const pair<int, int>& right) {

    return left.second > right.second; // 注意这里是大于，因为我们想要小根堆

};

// struct compare {

//     bool operator()(const node &x, const node &y) const {

//         return x.dis > y.dis;

//     }

// };//这种是更常用的

// priority\_queue<pair<int, int>, vector<pair<int,int>>,compare>heap(compare);

priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(compare)> heap(compare);

//初始化建立小根堆

vector<vector<pair<int ,int >>>graph;

//邻接表建图

int edge[MAXM][3];

int Distance[MAXN];

//记录到源点的最小距离

bool visit[MAXN];

//记录这个点是否被访问过

int n,m,s;

int main()

{

    cin>>n>>m>>s;

    graph.resize(n+1);

    for(int i=1;i<=m;i++){

        cin>>edge[i][0]>>edge[i][1]>>edge[i][2];

        graph[edge[i][0]].push\_back({edge[i][1],edge[i][2]});

    }

    //初始化建图

    for(int i=1;i<=n;i++){

        Distance[i]=1e9;

    }

    //将数组初始化

    Distance[s]=0;

    //以s点作为源点传输信号

    heap.push({s,0});

    //小根堆储存的形式是  节点——权值  以权值建堆

    while(!heap.empty()){

        int u=heap.top().first;

        heap.pop();

        if(visit[u]){

            continue;

        }

        visit[u]=true;

        for(int i=0;i<graph[u].size();i++){

            int v=graph[u][i].first;

            int w=graph[u][i].second;

            if(!visit[v]&&(w+Distance[u])<Distance[v]){

                Distance[v]=w+Distance[u];

                heap.push({v,Distance[u]+w});

            }

        }

    }

    int ans=0;

    for(int i=1;i<=n;i++){

        if(Distance[i]==1e9){

            //表示无法到达

            cout<<-1;

            return 0;

        }

        ans=max(ans,Distance[i]);

        //取最大值  表示最远到达的点距离

    }

    cout<<ans;

    return 0;

}

Floyd算法

// Floyd算法模版（洛谷）

// 测试链接 : https://www.luogu.com.cn/problem/P2910

// 请同学们务必参考如下代码中关于输入、输出的处理

// 这是输入输出处理效率很高的写法

// 提交以下所有代码，把主类名改成Main，可以直接通过

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 101;

const int MAXM = 10001;

int n,m,ans;

int dis[MAXN][MAXN];

int path[MAXM];

void build() {

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            dis[i][j] = INT\_MAX;

        }

    }

}

void floyd() {

    // O(N^3)的过程

    // 枚举每个跳板

    // 注意，跳板要最先枚举！跳板要最先枚举！跳板要最先枚举！

    for (int bridge = 0; bridge < n; bridge++) { // 跳板

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                // i -> .....bridge .... -> j

                // distance[i][j]能不能缩短

                // distance[i][j] = min ( distance[i][j] , distance[i][bridge] + distance[bridge][j])

                if (dis[i][bridge] != INT\_MAX

                        && dis[bridge][j] != INT\_MAX

                        && dis[i][j] > dis[i][bridge] + dis[bridge][j]) {

                    dis[i][j] = dis[i][bridge] + dis[bridge][j];

                }

            }

        }

    }

}

int main()

{

    cin>>n>>m;

    //build();

    //正常情况下，是需要我们自己建立邻接矩阵的

    for(int i=1;i<=m;i++){

        cin>>path[i];

        path[i]--;

        //注意下标的对应

    }

    for(int i=0;i<n;i++){

        for(int j=0;j<n;j++){

            cin>>dis[i][j];

            //这道题直接给出来邻接矩阵  不需要我们建立

        }

    }

    floyd();

    for(int i=1;i<m;i++){

        ans+=dis[path[i]][path[i+1]];

    }

    cout<<ans;

    return 0;

}

SPFA算法

//P2865  很经典的spfa

//这道题解决的是一个次短路问题  可以走重复边

//这个题目没有松弛操作次数的要求 因为所有边权都是大于0的

//记录松弛次数是为了判断有没有负环

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 5005;

const int MAXM = 2e5+5;

int n,m;

int head[MAXN];

int Next[MAXM];

int to[MAXM];

int weight[MAXM];

int cnt=1;

//spfa需要 这里由于是次短路

//dis[i][0]表示的是从源点 一般是1或者n节点  到达i节点的最短路径

//dis[i][1]表示的是次短路径

int dis[MAXN][2];

bool vis[MAXN];//表示现在是否在队列中

queue<int>q;//不需要堆  一个队列即可

void addedge(int u,int v,int w){

    Next[cnt]=head[u];

    to[cnt]=v;

    weight[cnt]=w;

    head[u]=cnt++;

}

void spfa(){

    for(int i=1;i<=n;i++){

        dis[i][0]=1e9;

        dis[i][1]=1e9;

    }

    dis[n][0]=0;

    q.push(n);

    vis[n]=true;

    while(!q.empty()){

        int u=q.front();

        q.pop();

        vis[u]=false;

        for(int i=head[u];i>0;i=Next[i]){

            int v=to[i];

            int w=weight[i];

            if(dis[v][0]>dis[u][0]+w){

                dis[v][1]=dis[v][0];

                dis[v][0]=dis[u][0]+w;

                if(!vis[v]){

                    //不在队列中  才会加入  这里要注意

                    q.push(v);

                    vis[v]=true;

                }

            }

            if(dis[v][1]>dis[u][0]+w&&dis[v][0]<dis[u][0]+w){

                //这里必须要说明是  <  因为如果相等的话是不符合要求的

                dis[v][1]=dis[u][0]+w;

                if(!vis[v]){

                    q.push(v);

                    vis[v]=true;

                }

            }

            if(dis[v][1]>dis[u][1]+w){

                dis[v][1]=dis[u][1]+w;

                if(!vis[v]){

                    q.push(v);

                    vis[v]=true;

                }

            }

        }

    }

}

int main(){

    cin>>n>>m;

    for(int i=1;i<=m;i++){

        int u,v,w;

        cin>>u>>v>>w;

        addedge(u,v,w);

        addedge(v,u,w);

    }

    spfa();

    cout<<dis[1][1]<<endl;

    return 0;

}

Tarjan——缩点

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 105;

vector<int>vec[MAXN];

//dfn表示的就是dfn序  而low数组表示的是这个节点沿着边可以走到的最小的dfn序

//dfn一旦确定了 就不会改变了  而low数组可能会发生变化

int dfn[MAXN],low[MAXN];

int st[MAXN],top;

int color,Time;

int col[MAXN];

int in[MAXN],out[MAXN];

bool instack[MAXN];

int n,m;

void tarjan(int x){

    dfn[x]=++Time;

    low[x]=Time;

    st[++top]=x;

    instack[x]=true;

    for(int i=0;i<vec[x].size();i++){

        int v=vec[x][i];

        if(!dfn[v]){

            //表示这个节点没有被访问过

            tarjan(v);

            low[x]=min(low[x],low[v]);

        }

        else if(instack[v]){

            //表示这个属于是回溯了 一定是同一个环上的

            //这里之所以不用low[v],是因为这里的含义就是dfn序号,其实就算改成low也不影响

            //不过这里最好写成这样 因为其他的要求可能会导致写成low出错

            low[x]=min(low[x],dfn[v]);

        }

    }

    if(dfn[x]==low[x]){

        col[x]=++color;

        //将所有节点按照颜色分类  完成缩点

        while(st[top]!=x){

            //属于同一个强联通分量

            col[st[top]]=color;

            instack[st[top]]=false;

            top--;

        }

        instack[x]=false;

        top--;

    }

}

int main(){

    cin>>n;

    for(int i=1;i<=n;i++){

        int v;

        cin>>v;

        while(v!=0){

            vec[i].push\_back(v);

            cin>>v;

        }

    }

    for(int i=1;i<=n;i++){

        if(!dfn[i]){

            tarjan(i);

        }

    }

    for(int i=1;i<=n;i++){

        for(int j=0;j<vec[i].size();j++){

            int v=vec[i][j];

            if(col[i]!=col[v]){

                in[col[v]]++;

                out[col[i]]++;

            }

        }

    }

    int ans1=0,ans2=0;

    for(int i=1;i<=color;i++){

        if(!in[i])

            ans1++;

        if(!out[i])

            ans2++;

    }

    cout<<ans1<<endl;

    if(color==1){

        cout<<0<<endl;

    }else{

        cout<<max(ans1,ans2)<<endl;

    }

    return 0;

}

Tarjan——割点

//P3388

// 我们判断割点的方法是：

// 1.如果这个点是根节点  那么如果他有两个儿子  则他一定是割点

// 这里的两个儿子指的是在深度优先搜索的条件下的两个儿子  即这两个儿子的子树一定是互不相干的

// 否则其中一个儿子的子树一定会包括另外一个儿子

// 2.如果这个节点不是根节点，并且不是叶子结点 并且low[他的任意一个孩子]>=dfn[他]  则它是割点

// 如果是叶节点  那么不可能是割点

// 并且他的所有孩子不能到达它的上面  即他扼死了孩子向上的通道

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 2e4+5;

vector<int>vec[MAXN];

//在割点的时候  dfn还是最早的序号 可是low却不再是这个节点可以到达的最小编号了

//和缩点的时候不同

int dfn[MAXN],low[MAXN];

int Time;

bool cut[MAXN];

bool instack[MAXN];

int n,m;

// v:当前点 r：本次搜索树的root

void tarjan(int u,int r) {

    dfn[u] = low[u] = ++Time;

    int child = 0;

    for (int i = 0; i < vec[u].size(); i++) {

        int v = vec[u][i];

        if (!dfn[v]) {

            tarjan(v, r);

            low[u] = min(low[u], low[v]);

            if (low[v] >= dfn[u] && u != r)

                cut[u] = true;//不是根而且他的孩子无法跨越他回到祖先

            if (r == u){

                child++; //如果是搜索树的根，统计孩子数目

                //其实所有的节点都可以统计孩子的数量 但是非根节点统计后没有什么作用

            }

        }

        low[u] = min(low[u], dfn[v]);//这里要特别注意 不能变成low[v]

        // 举个例子 比如a-b-c-d-b  并且还存在b-e-a

        // 那么首先b是一个割点  可是如果改成low[v] 那么low[d]=dfn[a]  导致b无法成为一个割点

    }

    if (child >= 2 && u == r){

        cut[r] = true;//对应第一种情况

    }

}

int main()

{

    cin>>n>>m;

    for(int i=1;i<=m;i++){

        int u,v;

        cin>>u>>v;

        vec[u].push\_back(v);

        vec[v].push\_back(u);

    }

    for(int i=1;i<=n;i++){

        if(!dfn[i]){

            tarjan(i,i);

        }

    }

    int ans=0;

    for(int i=1;i<=n;i++){

        if(cut[i]){

            ans++;

        }

    }

    cout<<ans<<endl;

    for(int i=1;i<=n;i++){

        if(cut[i]){

            cout<<i<<' ';

        }

    }

    return 0;

}

Tarjan——桥

//P1656

// 桥其实和tarjan没有什么本质上的区别  都是一样的

// 桥的判断方法是对于一条边  如果下面的点的low大于上面点的dfn  说明是桥

// 如果这条边是回溯的  那不算

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 155;

const int MAXM = 5005;

int n,m;

vector<pair<int,int>>bridges;

int dfn[MAXN];

int low[MAXN];

int dfncnt;

int cnt=1;

int head[MAXN];

int nxt[MAXM<<1];

int to[MAXM<<1];

int weight[MAXM<<1];//记录这条边的编号

void addedge(int u,int v,int i){

    nxt[cnt]=head[u];

    to[cnt]=v;

    weight[cnt]=i;

    head[u]=cnt++;

}

void tarjan(int u,int id){

    //表示当前来到了点u  是通过真实编号id这条边来的

    dfn[u]=low[u]=++dfncnt;

    for(int i=head[u];i;i=nxt[i]){

        int v=to[i];

        int w=weight[i];

        if(!dfn[v]){

            tarjan(v,w);

            low[u]=min(low[u],low[v]);

            if(low[v]>dfn[u]){

                bridges.push\_back({u,v});

            }

        }

        else if(w!=id){

            //不能走回头路

            low[u]=min(low[u],dfn[v]);

        }

    }

}

int main()

{

    cin>>n>>m;

    for(int i=1;i<=m;i++){

        int u,v;

        cin>>u>>v;

        addedge(u,v,i);

        addedge(v,u,i);

    }

    for(int i=1;i<=n;i++){

        if(!dfn[i]){

            tarjan(i,0);

        }

    }

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

    for (auto &p : bridges) {

        cout << p.first << " " << p.second << endl;

    }

    return 0;

}

拓扑排序

// 拓扑排序模版（Leetcode）

// 邻接表建图（动态方式）

// 课程表II

// 现在你总共有 numCourses 门课需要选，记为 0 到 numCourses - 1

// 给你一个数组 prerequisites ，其中 prerequisites[i] = [ai, bi]

// 表示在选修课程 ai 前 必须 先选修 bi

// 例如，想要学习课程 0 ，你需要先完成课程 1 ，我们用一个匹配来表示：[0,1]

// 返回你为了学完所有课程所安排的学习顺序。可能会有多个正确的顺序

// 你只要返回 任意一种 就可以了。如果不可能完成所有课程，返回 一个空数组

// 测试链接 : https://leetcode.cn/problems/course-schedule-ii/

#include<bits/stdc++.h>

using namespace std;

#define int long long

const int MAXN = 5005;

const int MAXM = 500005;

const int MOD = 80112002;

int n,m,cnt=1;

int head[MAXN];

int Next[MAXM];

int to[MAXM];

int edge[MAXM][2];

int indegree[MAXN];

int line[MAXN];

int ans=0;

queue<int >pq;

signed main()

{

    cin>>n>>m;

    for(int i=1;i<=m;i++){

        cin>>edge[i][0]>>edge[i][1];

    }

    for(int i=1;i<=m;i++){

        //连边 建图

        Next[cnt]=head[edge[i][0]];

        to[cnt]=edge[i][1];

        head[edge[i][0]]=cnt++;

        indegree[edge[i][1]]++;

    }

    for(int i=1;i<=n;i++){

        if(indegree[i]==0){

            pq.push(i);

            line[i]=1;

        }

        //printf("%3d %3d %3d\n",indegree[i],line[i],head[i]);

    }

    while(!pq.empty()){

        int cur=pq.front();

        pq.pop();

        if(head[cur]==0){

            ans=(ans+line[cur])%MOD;

        }else{

            for(int i=head[cur];i>0;i=Next[i]){

                indegree[to[i]]--;

                line[to[i]]=(line[cur]+line[to[i]])%MOD;

                if(indegree[to[i]]==0){

                    pq.push(to[i]);

                }

            }

        }

    }

    cout<<ans;

    return 0;

}

扩展欧几里得

// 同余方程

// 求关于x的同余方程 ax ≡ 1(mod b) 的最小正整数解

// 题目保证一定有解，也就是a和b互质

// 2 <= a、b <= 2 \* 10^9

// 测试链接 : https://www.luogu.com.cn/problem/P1082

// 提交以下的code，提交时请把类名改成"Main"，可以通过所有测试用例

//在exgcd之后 求出来的特解x实际上是a在%b意义下的逆元

#include<bits/stdc++.h>

using namespace std;

long long d, x, y, px, py;

void exgcd(long long a, long long b) {

    if (b == 0) {

        //表示已经求出了最大公约数d      x y分别是1 0

        d = a;//最大公约数设置为a

        x = 1;

        y = 0;

    } else {

        exgcd(b, a % b);

        //这个过程在向上的时候逐步更新

        //d=x'\*a'+y'\*b'

        // =x'\*b+y'\*(a-(a/b)\*b)

        // =y'\*a+(x'-(a/b)\*y')b

        // =x\*a+y\*b

        //x=y'   y=x'-(a/b)\*y'

        px = x;

        py = y;

        x = py;

        y = px - py \* (a / b);

    }

}

int main()

{

    long long a,b;

    cin>>a>>b;

    exgcd(a,b);

    cout<<((x % b + b) % b);

    return 0;

}

求格点公式：gcd(abs(x1 - x2), abs(y1 - y2)) + 1

中国剩余定理（CRT）

// 中国剩余定理模版

// 给出n个同余方程，求满足同余方程的最小正数解x

// 一共n个同余方程，x ≡ ri(% mi)

// 1 <= n <= 10

// 0 <= ri、mi <= 10^5

// 所有mi一定互质

// 所有mi整体乘积 <= 10^18

// 测试链接 : https://www.luogu.com.cn/problem/P1495

// 提交以下的code，提交时请把类名改成"Main"，可以通过所有测试用例

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 11;

long long m[MAXN];

long long r[MAXN];

long long d, x, y, px, py;

// 原理来自，讲解033，位运算实现乘法

// a \* b的过程自己实现，每一个中间过程都%mod

// 这么写目的是防止溢出，也叫龟速乘

long long multiply(long long a, long long b, long long mod) {

    // 既然是在%mod的意义下，那么a和b可以都转化成非负的

    // 本题不转化无所谓，但是其他题目可能需要转化

    // 尤其是b需要转化，否则while循环会跑不完

    a = (a % mod + mod) % mod;

    b = (b % mod + mod) % mod;

    long long ans = 0;

    while (b != 0) {

        if ((b & 1) != 0) {

            ans = (ans + a) % mod;

        }

        a = (a + a) % mod;

        b >>= 1;

    }

    return ans;

}

void exgcd(long long a, long long b) {

    if (b == 0) {

        d = a;

        x = 1;

        y = 0;

    } else {

        exgcd(b, a % b);

        px = x;

        py = y;

        x = py;

        y = px - py \* (a / b);

    }

}

// 中国剩余定理模版  只适用于模数互质的情况

long long crt(int n) {

    long long lcm = 1;

    for (int i = 1; i <= n; i++) {

        lcm = lcm \* m[i];

        //因为m是互质的  所以最小公倍数就是他们的累乘

    }

    long long ai, ci, ans = 0;

    for (int i = 1; i <= n; i++) {

        // ai = lcm / m[i]

        // 即 m1 \* m2 \*```\* mi-1 \* mi+1 \*```\* mn-1 \* mn

        ai = lcm / m[i];

        // ai逆元，在%m[i]意义下的逆元

        exgcd(ai, m[i]);

        // ci = (ri \*   ai \* ai逆元(即x)  ) % lcm

        //这里之所以要%lcm  是因为我们要求的是最小的符合要求的正整数  所有符合要求的可以表示为特解加上任意个lcm

        ci = multiply(r[i], multiply(ai, x, lcm), lcm);

        ans = (ans + ci) % lcm;

    }

    return ans;

}

int main()

{

    int n;

    cin>>n;

    for(int i=1;i<=n;i++){

        cin>>m[i]>>r[i];

    }

    cout<<crt(n)<<endl;

    return 0;

}

扩展中国剩余定理（exCRT）

// 扩展中国剩余定理模版

// 给出n个同余方程，求满足同余方程的最小正数解x

// 一共n个同余方程，x ≡ ri(% mi)

// 1 <= n <= 10^5

// 0 <= ri、mi <= 10^12

// 所有mi不一定互质

// 所有mi的最小公倍数 <= 10^18

// 测试链接 : https://www.luogu.com.cn/problem/P4777

// 测试链接 : https://www.luogu.com.cn/problem/P1495

// 提交以下的code，提交时请把类名改成"Main"，可以通过所有测试用例

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 100001;

long long m[MAXN],r[MAXN];

long long d, x, y, px, py;

long long multiply(long long a, long long b, long long mod) {

    // 既然是在%mod的意义下，那么a和b可以都转化成非负的

    // 本题不转化无所谓，但是其他题目可能需要转化

    // 尤其是b需要转化，否则while循环会跑不完

    a = (a % mod + mod) % mod;

    b = (b % mod + mod) % mod;

    long long ans = 0;

    while (b != 0) {

        if ((b & 1) != 0) {

            ans = (ans + a) % mod;

        }

        a = (a + a) % mod;

        b >>= 1;

    }

    return ans;

}

void exgcd(long long a, long long b) {

    if (b == 0) {

        d = a;

        x = 1;

        y = 0;

    } else {

        exgcd(b, a % b);

        px = x;

        py = y;

        x = py;

        y = px - py \* (a / b);

    }

}

// 扩展中国剩余定理模版 可以用于解决模数不是互质的情况  适用范围更广

long long excrt(int n) {

    long long tail = 0, lcm = 1, tmp, b, c, x0;

    // ans = lcm \* x + tail

    for (int i = 1; i <= n; i++) {

        // ans = m[i] \* y + ri

        // lcm \* x + m[i] \* y = ri - tail

        // a = lcm

        // b = m[i]

        // c = ri - tail

        b = m[i];

        c = ((r[i] - tail) % b + b) % b;

        exgcd(lcm, b);

        if (c % d != 0) {

            return -1;

        }

        // ax + by = gcd(a,b)，特解是，x变量

        // ax + by = c，特解是，x变量 \* (c/d)

        // ax + by = c，最小非负特解x0 = (x \* (c/d)) % (b/d) 取非负余数

        //特解是x0 通解 = x0 + (b/d) \* n

        x0 = multiply(x, c / d, b / d);

        // ans = lcm \* x + tail，带入通解

        // ans = lcm \* (x0 + (b/d) \* n) + tail

        // ans = lcm \* (b/d) \* n + lcm \* x0 + tail

        // tail' = tail' % lcm'

        tmp = lcm \* (b / d);

        tail = (tail + multiply(x0, lcm, tmp)) % tmp;

        lcm = tmp;

    }

    return tail;

}

int main(){

    int n;

    cin>>n;

    for(int i=1;i<=n;i++){

        cin>>m[i]>>r[i];

    }

    cout<<excrt(n)<<endl;

    return 0;

}

质数筛

// 计数质数

// 给定整数n，返回小于非负整数n的质数的数量

// 测试链接 : https://leetcode.cn/problems/count-primes/

//这里介绍两种质数筛：埃氏筛 欧拉筛

//埃氏筛很容易理解

//欧拉筛的扩展功能很强  可以记录每个数字的最小质因子

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 1e6+5;

bool visit[MAXN];//用于埃氏筛统计是否是质数

int prime[MAXN];//欧拉筛中用于收集质数的数组

int num[MAXN];//记录每个数字的最小质因子

// 埃氏筛统计0 ~ n范围内的质数个数

// 时间复杂度O(n \* log(logn))

int ehrlich(int n) {

    memset(visit,0,sizeof(visit));

    // 初始时认为0~n所有数都是质数，但0和1不是质数

    for(int i=2;i\*i<=n;i++){

        if(!visit[i]){

            for(int j=i\*i;j<=n;j+=i){

                visit[j]=true;

            }

        }

    }

    int cnt=0;

    for(int i=2;i<=n;i++){

        if(!visit[i]){

            cnt++;

        }

    }

    return cnt;

}

// 欧拉筛统计0 ~ n范围内的质数个数

// 时间复杂度O(n)

//欧拉筛还可以设置每个数字的最小质因子

int euler(int n){

    int cnt=0;

    memset(num,0,sizeof(num));

    for(int i=2;i<=n;i++){

        if(num[i]==0){

            prime[++cnt]=i;

        }

        //无论是不是质数  都要进行下面的过程

        for(int j=1;j<=cnt&&i\*prime[j]<=n;j++){

            num[i\*prime[j]]=prime[j];

            if(i%prime[j]==0){

                //如果i可以整除prime[j]说明一定含有这个质因子

                //那么如果继续的话  就是将接下来的数字的按照下一个质数作为他的最小质因子排除的

                //而不是被最小质因子排除的  所以不能继续  要立即跳出

                break; // 每个合数只被其最小的质因数筛去一次

            }

        }

    }

    return cnt;

}

int main()

{

    int n;

    while(cin>>n){

        cout<<n<<": "<<ehrlich(n)<<' '<<euler(n)<<' '<<ehrlich2(n)<<endl;

    }

    return 0;

}

逆元

//这个文件中以下三种逆元：

//单个除数的逆元

//连续数字的逆元

//连续数字阶乘逆元

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 3e5+5;

int n,m,mod;

int inv[MAXN];//连续数字的逆元

int fac[MAXN];//连续数字的阶乘

int facinv[MAXN];//连续数字阶乘的逆元

//乘法快速幂  求a的b次方对mod取余

long long power(long long a,long long b,long long mod){

    long long ans=1;

    while(b){

        if(b&1){

            ans\*=a;

            ans%=mod;

        }

        b>>=1;

        a\*=a;

        a%=mod;

    }

    return ans;

}

//单个除数的逆元

//费马小定理求(a/b)%mod的值

long long compute(long long a,long long b,long long mod){

    long long inv=power(b,mod-2,mod);//求b在%mod的情况下的逆元

    return (a\*inv)%mod;

}

//连续数字的逆元

// 预计算模逆元

void build(int n) {

    inv[1] = 1;

    for (int i = 2; i <= n; i++) {

        inv[i] = (mod - (long long)inv[mod % i] \* (mod / i) % mod + mod) % mod;

    }

}

//连续数字阶乘的逆元

// 初始化阶乘表和逆元表

void build() {

    //先求出阶乘表

    fac[0] = 1; // 0! = 1

    for (int i = 1; i <= n; i++) {

        fac[i] = (fac[i - 1] \* i) % mod;

    }

    // 利用线性递推优化计算逆元

    facinv[n] = power(fac[n], mod - 2,mod);//求出最后一个数字的阶乘逆元

    for (int i = n - 1; i >= 0; i--) {

        facinv[i] = (facinv[i + 1] \* (i + 1)) % mod;//线性递推

    }

}

int main()

{

    cin>>n>>m>>mod;

    compute(n,m,mod);

    build(n);

    build();

    return 0;

}

差分（二维、等差数列）

// 一开始1~n范围上的数字都是0，一共有m个操作，每次操作为(l,r,s,e,d)

// 表示在l~r范围上依次加上首项为s、末项为e、公差为d的数列

// m个操作做完之后，统计1~n范围上所有数字的最大值和异或和

//等差数列差分需要两遍前缀和

#include<bits/stdc++.h>

using namespace std;

int n,m;

long long arr[10000005];

long long l,s,r,e,maxans,eor;

void Set(long long l,long long r,long long s,long long e,long long d){

    //以上参数分别表示  左起点  右终点  初始增加值  末尾增加值  公差

    arr[l]+=s;

    arr[l+1]+=(d-s);

    arr[r+1]-=(d+e);

    arr[r+2]+=e;

}

void build(){

    //两遍前缀和完成

    for(int i=1;i<=n;i++){

        arr[i]+=arr[i-1];

    }

    for(int i=1;i<=n;i++){

        arr[i]+=arr[i-1];

    }

}

int main(){

    cin>>n>>m;

    while(m--){

        l=read(),r=read(),s=read(),e=read();

        Set(l,r,s,e,(e-s)/(r-l));

    }

    build();

    eor=0;

    for(int i=1;i<=n;i++){

        maxans=max(maxans,arr[i]);

        eor^=arr[i];

    }

    cout<<eor<<' '<<maxans<<endl;

    return 0;

}

// 二维差分模版(洛谷)

#include<bits/stdc++.h>

using namespace std;

int n,m;

int nums[1005][1005];

void Set(int a,int b,int c,int d,int v){

    nums[a][b]+=v;

    nums[a][d+1]-=v;

    nums[c+1][b]-=v;

    nums[c+1][d+1]+=v;

}

void build(){

    //来一遍二维前缀和即可

    for(int i=1;i<=n;i++){

        for(int j=1;j<=n;j++){

            nums[i][j]+=nums[i-1][j]+nums[i][j-1]-nums[i-1][j-1];

        }

    }

}

int main()

{

    cin>>n>>m;

    int a,b,c,d;

    while(m--){

        cin>>a>>b>>c>>d;

        Set(a,b,c,d,1);

    }

    build();

    for(int i=1;i<=n;i++){

        for(int j=1;j<=n;j++){

            printf("%d ",nums[i][j]);

        }

        printf("\n");

    }

    return 0;

}

二维前缀和

#include<bits/stdc++.h>

using namespace std;

const int MAXN=130;

int n;

int nums[MAXN][MAXN],sum[MAXN][MAXN];

int fun(int X1,int Y1,int X2,int Y2){

    return sum[X2][Y2]+sum[X1-1][Y1-1]-sum[X2][Y1-1]-sum[X1-1][Y2];

}

int main()

{

    cin>>n;

    for(int i=1;i<=n;i++){

        for(int j=1;j<=n;j++){

            cin>>nums[i][j];

            sum[i][j]=sum[i-1][j]+sum[i][j-1]-sum[i-1][j-1]+nums[i][j];

        }

    }

    int ans=-1e9;

    for(int X1=1;X1<=n;X1++){

        for(int Y1=1;Y1<=n;Y1++){

            for(int X2=X1;X2<=n;X2++){

                for(int Y2=Y1;Y2<=n;Y2++){

                    ans=max(ans,fun(X1,Y1,X2,Y2));

                }

            }

        }

    }

    cout<<ans;

    return 0;

}

博弈论（sg函数）

int sg[MAXN]; // 每个位置的 SG 值

bool appear[MAXV]; // 用于计算 SG 值的辅助数组

// 预处理 SG 值

void build() {

    for (int i = 1; i < MAXN; i++) {

        memset(appear, false, sizeof(appear)); // 重置 appear 数组

        for (int j = i - 1; j >= 0; j--) {

            for (int k = j; k >= 0; k--) {

                appear[sg[j] ^ sg[k]] = true; // 标记 SG 值的组合

            }

        }

        for (int s = 0; s < MAXV; s++) {

            if (!appear[s]) {

                sg[i] = s; // 找到最小的未出现的 SG 值

                break;

            }

        }

    }

}

倍增ST表

// ST表查询最大值和最小值

// 给定一个长度为n的数组arr，一共有m次查询

// 每次查询arr[l~r]上的最大值和最小值

#include<bits/stdc++.h>

using namespace std;

const int MAXN = 70005;

const int LIMIT = 16;

int stmax[MAXN][LIMIT],stmin[MAXN][LIMIT];

int arr[MAXN],Log2[MAXN],n,m;

void build(int n){

    Log2[0]=-1;

    for(int i=1;i<=n;i++){

        Log2[i]=Log2[i>>1]+1;

        stmax[i][0]=arr[i];

        stmin[i][0]=arr[i];

    }

    for(int i=1;i<=Log2[n];i++){

        for(int j=1;j<=n;j++){

            stmax[j][i]=max(stmax[j][i-1],stmax[j+(1<<(i-1))][i-1]);

            stmin[j][i]=min(stmin[j][i-1],stmin[j+(1<<(i-1))][i-1]);

        }

    }

}

int query(int l,int r){

    int p=Log2[r-l+1];

    int maxans=max(stmax[l][p],stmax[r-(1<<p)+1][p]);

    int minans=min(stmin[l][p],stmin[r-(1<<p)+1][p]);

    return maxans-minans;

}

int main(){

    cin>>n>>m;

    for(int i=1;i<=n;i++)

        cin>>arr[i];

    build(n);

    while(m--){

        int l,r;

        cin>>l>>r;

        cout<<query(l,r)<<endl;

    }

    return 0;

}

带权并查集

//寻找i节点的父亲 扁平化处理  并在该过程中更新距离

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] += dist[tmp];//将距离更新正确

    }

    return fa[i];

}

void un(int l, int r, long long v) {

int lf = find(l), rf = find(r);

if (lf != rf) {

fa[lf] = rf;//强制要求后面的节点作为父亲

dist[lf] = v + dist[r] - dist[l];

}

}

//加减关系：

//寻找i节点的父亲 扁平化处理  并在该过程中更新距离

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] += dist[tmp];//将距离更新正确

    }

    return fa[i];

}

void un(int l, int r) {

    int lf = find(l), rf = find(r);

    if (lf != rf) {

        fa[lf] = rf;//强制要求后面的节点作为父亲

        dist[lf] += sz[rf];

        sz[rf] += sz[lf];

    }

}

//种类关系

int find(int i) {

    if (i != fa[i]) {

        int tmp = fa[i];

        fa[i] = find(tmp);

        dist[i] = (dist[i] + dist[tmp]) % 3;

        //更新关系比较特殊

    }

    return fa[i];

}

// op == 1, 1 l r，l和r是同类

// op == 2, 2 l r，l吃r

void un(int op, int l, int r) {

    int lf = find(l), rf = find(r), v = op == 1 ? 0 : 1;//判断二者的关系

    if (lf != rf) {

        fa[lf] = rf;

        dist[lf] = (dist[r] - dist[l] + v + 3) % 3;

    }

}

//异或关系

int find(int i){

    if(i!=fa[i]){

        int tmp=fa[i];

        fa[i]=find(tmp);

        dis[i]^=dis[tmp];

    }

    return fa[i];

}

可撤销并查集

//利用数组完成栈的功能

int rollback[MAXN][2];

int opsize = 0;//表示栈的大小

void undo() {

    //撤销一步操作

    int fx = rollback[opsize][0];

    int fy = rollback[opsize--][1];

    father[fy] = fy;

    siz[fx] -= siz[fy];

    edgeCnt[fx] -= edgeCnt[fy] + 1;

}