2023 SCCC 봄 #11

BOJ 27965. N결수

어떤 수 x 뒤에 d개의 숫자로 이루어진 수 $10^{d-1} \le y < 10^d$를 붙이면 $x \times 10^d + y$가 됩니다. 따라서 10^d 꼴의 수를 K로 나눈 나머지를 모두 전처리해서 갖고 있으면, 각각의 수를 문자열로 나타내었을 때의 길이를 구해서 정답을 계산할 수 있습니다.

C++의 `std::to_string`을 이용하면 정수를 문자열로 변환할 수 있으므로 간단하게 수의 길이를 알 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll N, K, P[11], R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> K;
    P[0] = 1; for(int i=1; i<11; i++) P[i] = P[i-1] * 10 % K;
    for(int i=1; i<=N; i++) R = (R * P[to_string(i).size()] + i) % K;
    cout << R;
}
```

BOJ 28106. 나무 타기

A_i가 모두 서로 다르기 때문에 한 정점의 DP값을 계산할 때 조상 정점들의 DP값을 가져오는 것보다는, 한 정점의 DP값을 계산한 다음 깊이 차이가 A_i 이하인 자손 정점들에게 DP값을 뿌리는 방식으로 계산하는 것이 편합니다.

모든 정점의 서브 트리 크기의 합은 $O(N^2)$이므로 $O(N^2)$ 시간에 문제를 해결할 수 있습니다. 자세한 구현 방법은 아래 코드를 참고하세요.

```
#include <bits/stdc++.h>
using namespace std;
constexpr int MOD = 998244353;
inline void Add(int &a, const int b){ if((a += b) >= MOD) a -= MOD; }

int N, P[2020], A[2020], D[2020], S, R;
vector<int> G[2020];

void Add(int v, int d, int lim, int dp){
    if(d != 0) Add(D[v], dp);
    if(d < lim) for(auto i : G[v]) Add(i, d+1, lim, dp);
}

void DFS(int v, int d){
    if(G[v].empty()) Add(R, D[v]);
    Add(v, 0, A[v], D[v]);
    for(auto i : G[v]) DFS(i, d+1);
}
```

```
int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> P[i];
    for(int i=1; i<=N; i++) cin >> A[i];
    S = find(P+1, P+N+1, 0) - P;
    for(int i=1; i<=N; i++) if(i != S) G[P[i]].push_back(i);
    D[S] = 1; DFS(S, 0);
    cout << R;
}
```

BOJ 24241. インフルエンザ (Flu)

각 도시가 처음으로 전염병이 유행하기 시작하는 시간을 구하는 문제이므로 BFS를 이용해 해결할 수 있습니다. 거리가 D 이하인 도시를 간선으로 연결한 그래프는 $O(ND^2)$ 시간에 만들 수 있고, 문제의 조건에 의해 간선은 최대 10개 존재하므로 $O(N)$ 시간에 모든 정점까지의 최단 거리를 구할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, M, D, K, X[101010], Y[101010], C[101010], S[101010];
alignas(32) int A[1064][1064];
vector<pair<int,int>> V;
vector<int> G[101010];

int f(pair<int,int> p){ return p.first*p.first + p.second*p.second; }

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> D >> K;
    for(int i=1; i<=N; i++) cin >> X[i] >> Y[i], A[X[i]+=25][Y[i]+=25] = i;
    for(int i=-D; i<=D; i++) for(int j=-D; j<=D; j++) if(i*i + j*j <= D*D)
V.emplace_back(i, j);
    stable_sort(V.begin(), V.end(), [](auto a, auto b){ return f(a) < f(b); });
    V.erase(V.begin());
    for(int i=1; i<=N; i++){
        int x = X[i], y = Y[i];
        for(const auto &[dx,dy] : V){
            if(A[x+dx][y+dy]) G[i].push_back(A[x+dx][y+dy]);
            if(G[i].size() == 10) break;
        }
    }

    memset(C, -1, sizeof C);
    queue<int> Q; Q.push(1); C[1] = 0;
    while(!Q.empty()){
        int v = Q.front(); Q.pop();
        S[C[v]]++; S[C[v]+M]--;
        for(auto i : G[v]) if(C[i] == -1) Q.push(i), C[i] = C[v] + 1;
    }
    partial_sum(S, S+101010, S);
    cout << S[K];
}
```

BOJ 24111. 戦国時代 (Sengoku)

각 사람은 45도 대각선 방향에 있는 칸들을 감시할 수 있습니다. 따라서 $x-y$와 동일한 사람, $x+y$가 동일한 사람끼리 모아서 계산하는 것이 자연스러운 발상입니다.

구체적으로, $x-y=k$인 사람이 한 명이라도 있으면 $k \le i < L$를 만족하는 모든 정수 i에 대해 $(i, i-k)$를 감시할 수 있습니다. 마찬가지로 $x+y=k$인 사람이 한 명이라도 있으면 $k-L < i \le k$를 만족하는 모든 정수 i에 대해 $(i, k-i)$를 감시할 수 있습니다.

각도가 다른 두 대각선이 교차하는 지점은 2번 세기 때문에 교차점의 개수를 구해야 하는데, $x+y=a$인 대각선과 $x-y=s$인 대각선은 $((a+s)/2, (a-s)/2)$에서 교차합니다. 이때 교차점이 존재하기 위해서는 $a \equiv s \pmod 2, 0 \le a+s < 2L, 0 \le a-s < 2L$을 만족해야 합니다.

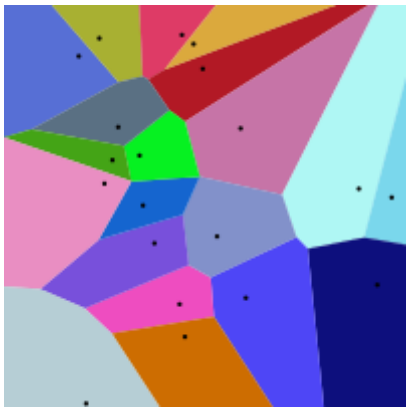
위 식을 다시 정리해 보면 $a \equiv s \pmod 2, |s| \le a < 2L - |s|$이고, s가 주어졌을 때 부등식을 만족하는 a의 개수는 이분 탐색을 이용해 $O(\log N)$ 시간에 구할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

void Compress(vector<ll> &v){
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
}

ll AddLen(ll len, ll x){
    // (i, x - i)
    // 0 <= i < len, x - len < i <= x
    ll st = max(0LL, x - len + 1), ed = min(len  - 1, x);
    return st <= ed ? ed - st + 1 : 0;
}

ll SubLen(ll len, ll x){
    // (i, i - x)
    // 0 <= i < len, x <= i < len + x
    ll st = max(0LL, x), ed = min(len - 1, len + x - 1);
    return st <= ed ? ed - st + 1 : 0;
}

ll L, N, R;
vector<ll> Add[2], Sub[2];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> L >> N;
    for(int i=0,x,y; i<N; i++) cin >> x >> y, Add[x+y&1].push_back(x+y), Sub[x-
y&1].push_back(x-y);
    for(int i=0; i<2; i++){
        Compress(Add[i]); Compress(Sub[i]);
        for(auto j : Add[i]) R += AddLen(L, j);
        for(auto j : Sub[i]) R += SubLen(L, j);
    }
    // add === sub (mod 2)
    // max(sub, -sub) <= add < min(L*2-sub, L*2+sub)
    for(int i=0; i<2; i++){
        for(auto sub : Sub[i]){
            auto le = lower_bound(Add[i].begin(), Add[i].end(), max(sub, -sub));
```

```
            auto ri = lower_bound(Add[i].begin(), Add[i].end(), min(L*2-sub,
L*2+sub));
            R -= distance(le, ri);
        }
    }
    cout << R;
}
```

BOJ 28098. 폭발 속에서 살아남기

2차원 평면 상의 모든 점들을 $(0,0)$ 또는 가장 가까운 폭발 지점에 따라 분류해 봅시다. 아래 그림과 같이 분할되고, 이런 그림을 보로노이 다이어그램이라고 부릅니다.



> Euclidean Voronoi diagram, Balu Ertl, CC-BY-SA 4.0

몇몇 영역은 무한한 넓이를 갖고 있음을 알 수 있습니다. $(0,0)$에서 출발해서 무한한 시간 동안 폭발에 휩쓸리지 않기 위해서는 $(0,0)$을 포함하는 영역의 넓이가 무한해야 합니다. 잘 생각해 보면 이 조건은 $(0,0)$이 폭발 지점들의 볼록 껍질 내부에 있지 않은 것과 동치라는 것을 알 수 있고, 따라서 $O(N \log N)$ 시간에 문제를 해결할 수 있습니다.

```
#include <bits/stdc++.h>
#define x first
#define y second
using namespace std;
using ll = long long;
using Point = pair<ll, ll>;

int CCW(const Point &p1, const Point &p2, const Point &p3){
    ll res = (p2.x - p1.x) * (p3.y - p1.y) - (p3.x - p1.x) * (p2.y - p1.y);
    return (res > 0) - (res < 0);
}

ll Dist(const Point &p1, const Point &p2){
    return (p2.x - p1.x) * (p2.x - p1.x) + (p2.y - p1.y) * (p2.y - p1.y);
}

int N;
vector<Point> V, H;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N; V.resize(N);
    for(auto &[a,b] : V) cin >> a >> b;
    swap(V[0], *min_element(V.begin(), V.end()));
    sort(V.begin()+1, V.end(), [](const auto &a, const auto &b){
```

```
        if(int dir=CCW(V[0], a, b); dir != 0) return dir > 0;
        return Dist(V[0], a) < Dist(V[0], b);
    });
    for(auto i : V){
        while(H.size() >= 2 && CCW(H[H.size()-2], H.back(), i) <= 0)
H.pop_back();
        H.push_back(i);
    }
    N = H.size();
    for(int i=0; i<N; i++){
        int j = (i + 1) % N;
        if(CCW(H[i], H[j], Point(0,0)) <= 0){ cout << "Yes"; return 0; }
    }
    cout << "No";
}
```

BOJ 28112. 나무 타기 (Hard)

A_i가 모두 같으면 HLD 같은 자료구조를 이용해 아무 생각 없이 밀어버릴 수 있지만, 아쉽게도 이 문제는 A_i가 서로 다를 수 있습니다. 이 말은 조상 정점들의 DP값을 가져오는 방식으로 계산하는 것보다는, 자신의 DP값을 자손 정점으로 뿌리는 방식으로 접근해야 함을 의미합니다.

v의 깊이를 L_v, 루트에서 시작해 v로 이동하는 경우의 수를 D_v라고 정의합시다. D_v는 $L_v < L_i \le L_v + A_v$를 만족하는 v의 자손 정점 i들에게 영향을 미칩니다.

트리에서 DFS를 수행하는 과정을 생각해 보면, DFS 함수 호출 스택에는 루트 정점과 현재 정점을 잇는 경로 상의 정점만 들어있습니다. 따라서 $DFS(v)$가 호출되었을 때 구간 $(L_v, L_v + A_v]$에 D_v를 더한 다음 $DFS(v)$가 종료될 때 D_v를 빼면, 각 정점의 DP값을 계산하는 것은 단순히 자료구조에서 현재 정점의 깊이에 있는 값을 가져오는 것으로 해결할 수 있습니다.

range add update point query는 펜윅 트리나 세그먼트 트리를 이용해 $O(\log N)$ 시간에 처리할 수 있습니다. 따라서 전체 문제를 $O(N \log N)$에 해결할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
constexpr int MOD = 998244353;
constexpr int SZ = 1 << 21;
void Add(int &a, const int b){ if((a += b) >= MOD) a -= MOD; }

int N, A[SZ], P[SZ], D[SZ], T[SZ<<1], S, R;
vector<int> G[SZ];

void Update(int l, int r, int v){
    for(l|=SZ, r|=SZ; l<=r; l>>=1, r>>=1){
        if(l & 1) Add(T[l++], v);
        if(~r & 1) Add(T[r--], v);
    }
}

int Query(int x){
    int res = 0;
    for(x|=SZ; x; x>>=1) Add(res, T[x]);
    return res;
}

void DFS(int v, int d){
    if(v != S) D[v] = Query(d);
```

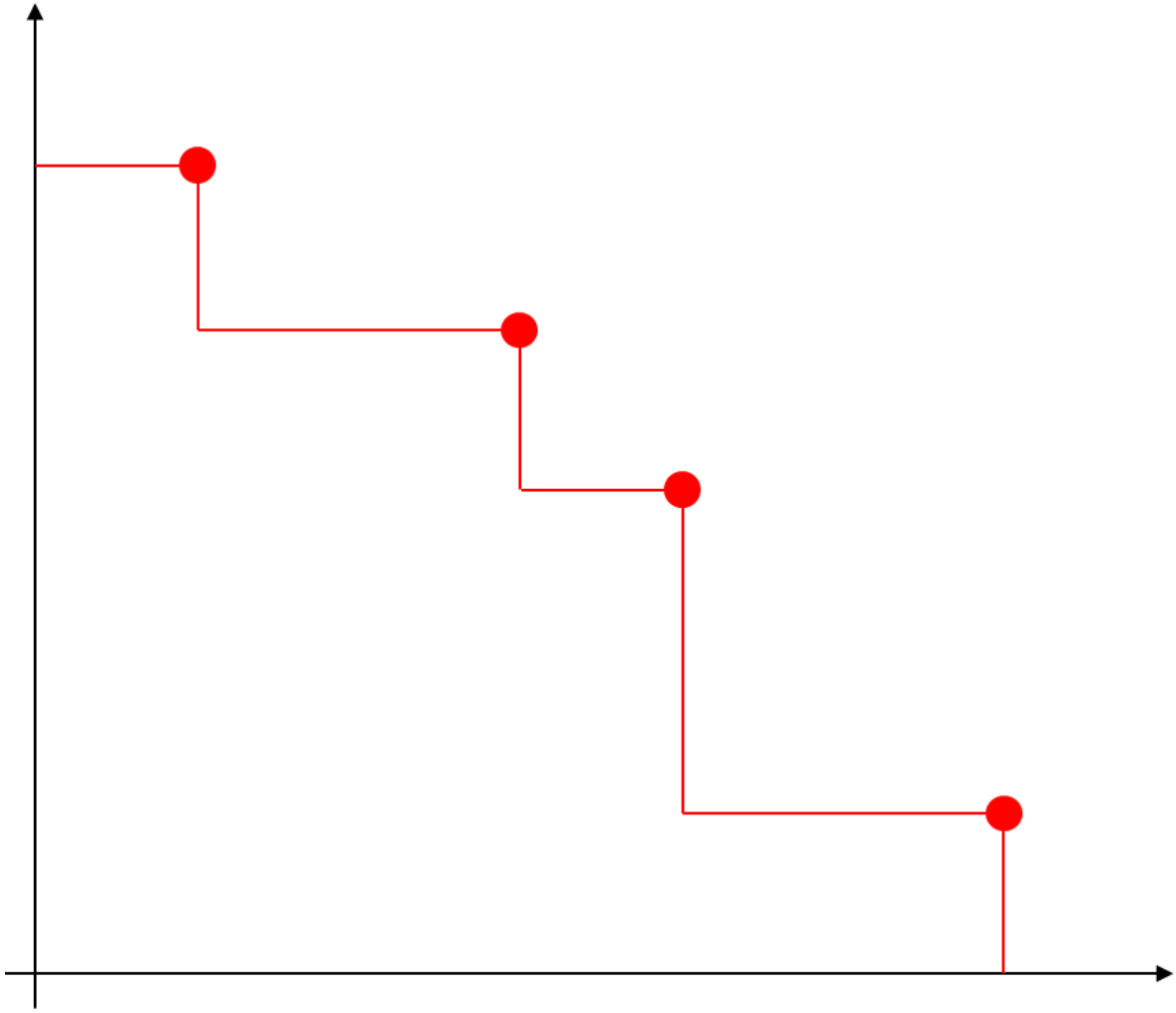
```
    if(G[v].empty()) Add(R, D[v]);
    Update(0, A[v] + d, D[v]);
    for(auto i : G[v]) DFS(i, d+1);
    Update(0, A[v] + d, D[v] ? MOD-D[v] : 0);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> P[i];
    for(int i=1; i<=N; i++) cin >> A[i];
    S = find(P+1, P+N+1, 0) - P;
    for(int i=1; i<=N; i++) if(i != S) G[P[i]].push_back(i);
    D[S] = 1; DFS(S, 0);
    cout << R;
}
```

BOJ 17763. Fish

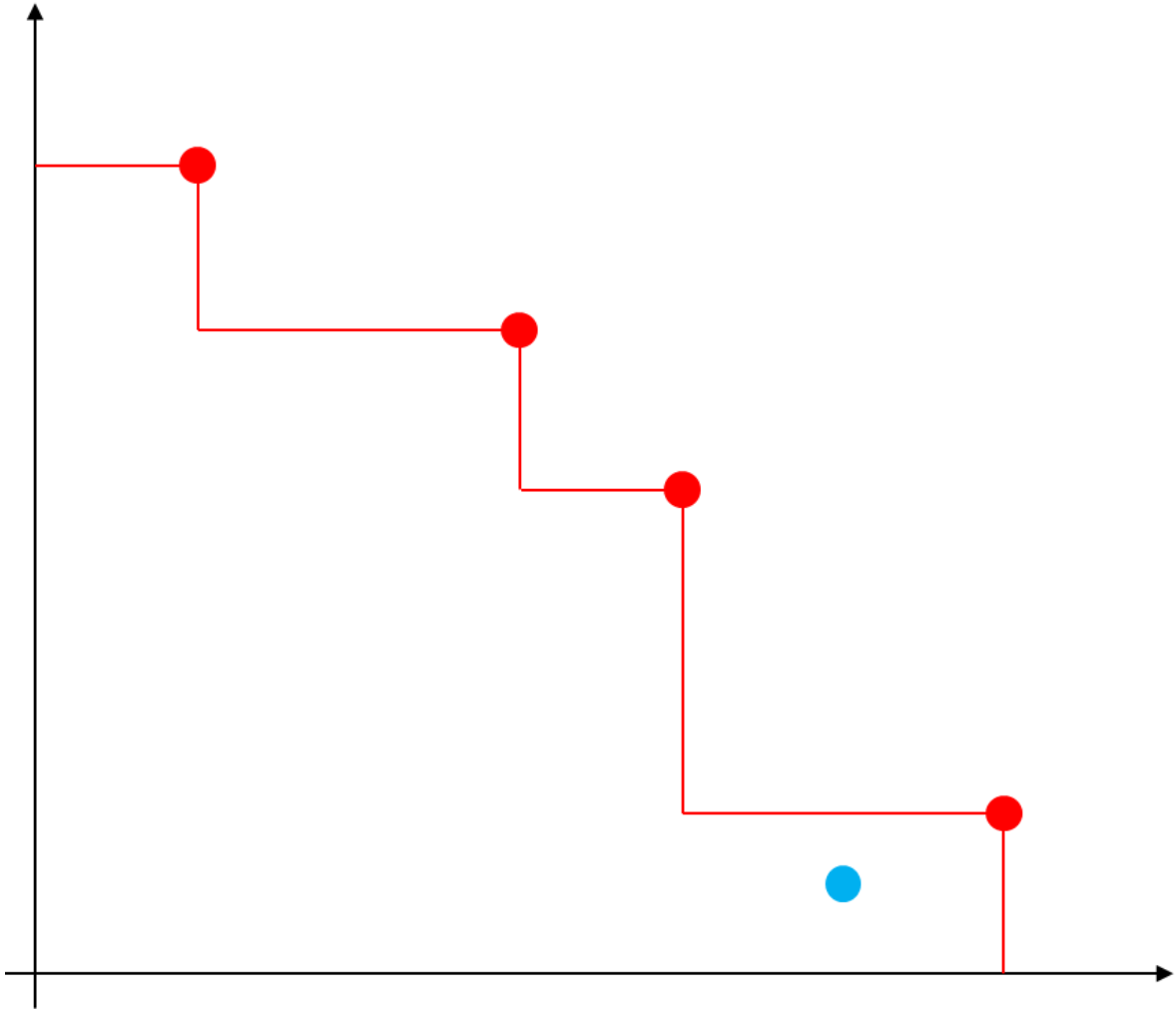
물고기를 크기 순으로 정렬하고 생각합시다. s번째 물고기부터 시작해서 연속한 몇 개의 물고기를 키운다고 할 때, 가능한 가장 큰 e는 투 포인터를 이용해 $O(N)$ 시간에 구할 수 있습니다. 이렇게 구간 $[s_i, e_i]$를 모두 모아서 보면, 각 구간은 빨간색이 최대 r_i개, 초록색이 최대 g_i개, 파란색이 최대 b_i개 들어갈 수 있다는 정보 (r_i, g_i, b_i)를 갖고 있습니다. 이 문제의 목표는 모든 구간의 정보에 대해, $[0, r_i] \times [0, g_i] \times [0, b_i]$ 꼴의 직육면체의 합집합에 포함되는 격자점의 개수를 세는 것입니다.

3차원을 생각하는 것은 어렵기 때문에 2차원 문제를 먼저 생각해 봅시다. 즉, $[0, r_i] \times [0, g_i]$ 꼴의 직사각형의 합집합에 포함되는 격자점의 개수를 빠르게 세는 방법을 알아봅시다.

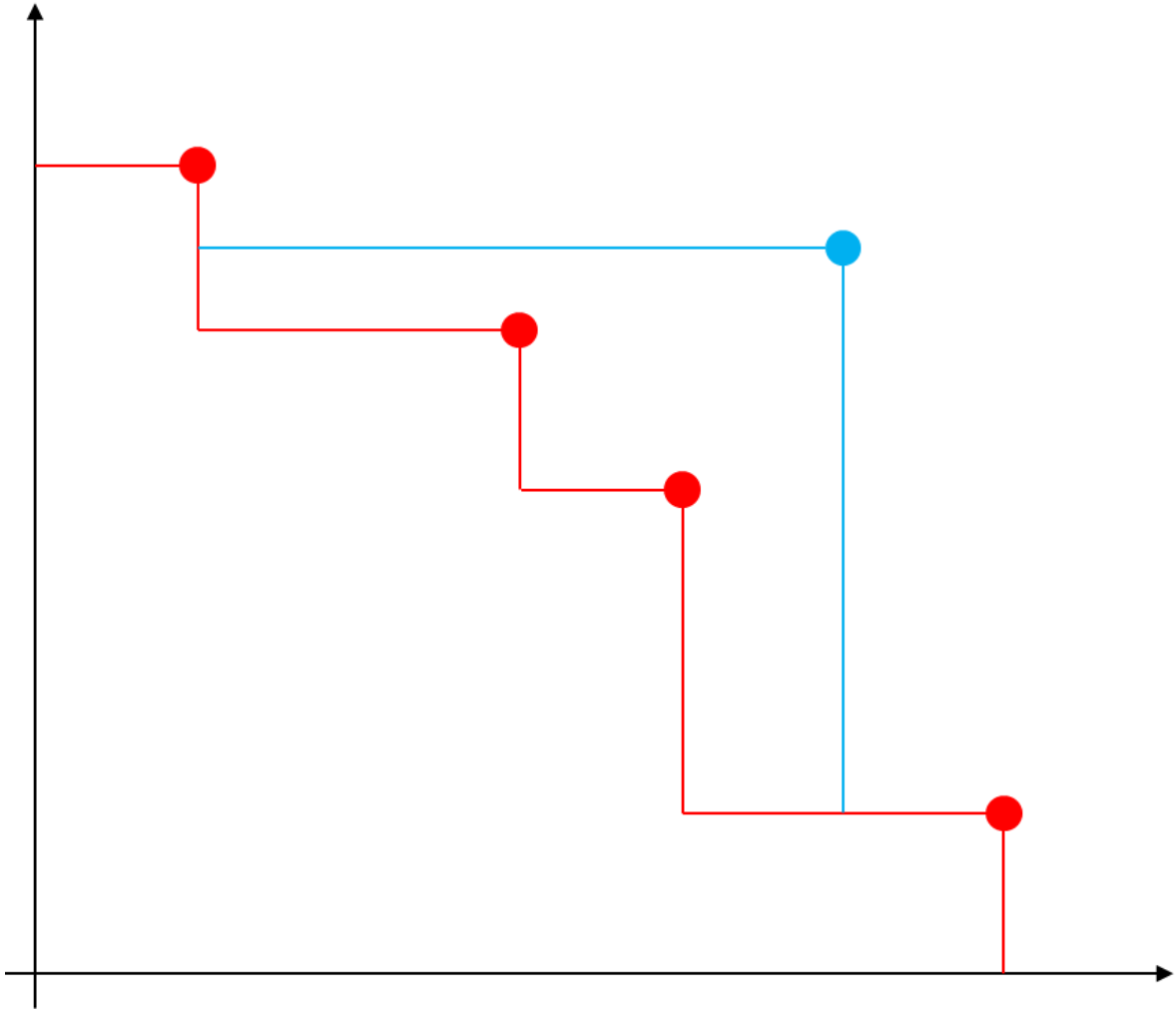
만약 $r_i \le r_j, g_i \le g_j$를 만족하는 i, j가 존재한다면 i번째 물고기는 넓이에 영향을 주지 못합니다. 따라서 넓이에 영향을 줄 수 있는 물고기를 r_i의 오름차순으로 정렬하면 g_i는 감소한다는 것을 알 수 있습니다. 그림으로 그려보면 오른쪽 아래로 내려가는 계단 형태로 생각할 수 있습니다.



새로운 점 (r_k, g_k)를 추가했을 때 넓이의 변화를 트래킹하는 방법을 생각해 봅시다. 만약 새로 추가된 점이 이미 계단 영역에 포함되어 있다면 점을 추가할 필요가 없습니다.



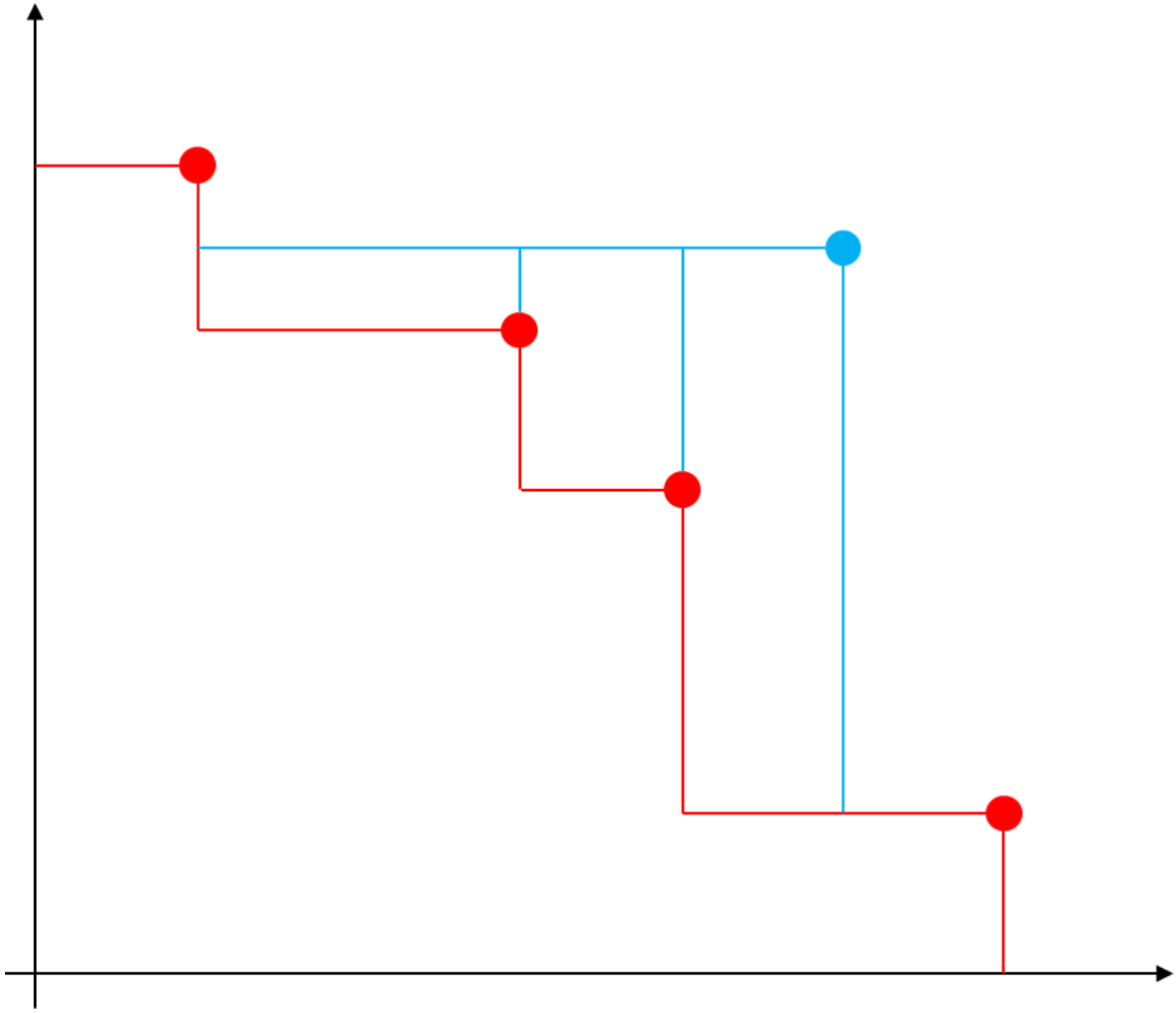
반대로, 새로 추가되는 점이 계단 영역 밖에 위치해 있다면 아래 그림처럼 기존에 있던 점 몇 개가 제거되고 넓이가 늘어나게 됩니다.



새로운 점이 추가될 때마다 추가되는 점에게 지배되는 점들을 빠르게 구해서 제거하고 넓이를 갱신할 수 있어야 합니다.

이를 효율적으로 처리하기 위해서 다른 점에게 지배되지 않는 점들의 목록을 `std::set`으로 관리할 것입니다. 어떤 점 (x, y)가 이미 계단 영역에 포함되어 있는지 확인하는 것은 `st.lower_bound(Point(x, -INF))->y >= y`인지 확인하면 됩니다.

추가되는 점 (x, y)에 의해 제거되는 점 목록은 x좌표가 x 미만인 가장 오른쪽에 있는 점을 보면서, 그 점의 y좌표가 y 이하인지 확인하면 됩니다. 각 점은 최대 1번 삽입되고 최대 1번 삭제되므로 N의 점 목록을 관리하는 것은 $O(N \log N)$ 만큼의 시간이 걸립니다. 점을 제거할 때마다 넓이를 갱신하는 것은 점이 제거될 때마다 아래 그림과 같이 직사각형 영역의 넓이를 더하면 됩니다. 이때 가장 왼쪽 영역도 계산해야 한다는 것에 주의해야 합니다.



3차원 문제를 직접 해결하는 것은 어렵습니다. 하지만 잘 생각해 보면 좌표가 클 때의 2차원 영역이 항상 좌표가 작을 때의 2차원 영역에 포함된다는 것을 알 수 있습니다. 따라서 점들을 세 번째 축의 좌표 내림차순으로 보면서, 점을 추가할 때마다 현재 단면의 넓이를 관리하면 문제를 해결할 수 있습니다.

종만북 2권 702페이지에 있는 "너드인가, 너드가 아닌가? 2" 문제의 풀이를 보면 이해하는 데 도움이 될 것입니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr int INF = 0x3f3f3f3f;

struct Point{
    ll x, y, z;
    Point() = default;
    Point(ll x, ll y, ll z) : x(x), y(y), z(z) {}
    bool operator < (const Point &p) const { return make_tuple(-x,y,z) <
make_tuple(-p.x,p.y,p.z); }
};

int N, C[505050][3];
pair<int,int> A[505050];
vector<Point> V;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N; V.reserve(N);
    for(int i=1; i<=N; i++){
```

```
        int a; char b; cin >> a >> b;
        A[i] = {a, string("RGB").find(b)};
    }
    sort(A+1, A+N+1);
    for(int i=1; i<=N; i++) C[i][A[i].second] = 1;
    for(int i=1; i<=N; i++) for(int j=0; j<3; j++) C[i][j] += C[i-1][j];
    for(int i=1; i<=N; i++){
        int j = lower_bound(A+1, A+N+1, make_pair(A[i].first*2, -1)) - A;
        int now[3];
        for(int k=0; k<3; k++) now[k] = C[j-1][k] - C[i-1][k];
        V.emplace_back(now[0] + 1, now[1] + 1, now[2] + 1); // 0 based -> 1
based
    }
    sort(V.begin(), V.end());

    ll plane = 0, res = 0;
    map<ll, ll> H; H[0] = INF; H[INF] = 0;
    for(int i=0, j=0; i<V.size(); i=j){
        while(j < V.size() && V[i].x == V[j].x) j++;
        for(int k=i; k<j; k++){
            auto it = H.lower_bound(V[k].y);
            if(it->second >= V[k].z) continue;
            ll base = it->second; --it;
            while(it != H.begin() && it->second <= V[k].z){
                plane += (V[k].y - it->first) * (it->second - base);
                base = it->second;
                auto t = prev(it); H.erase(it); it = t;
            }
            plane += (V[k].y - it->first) * (V[k].z - base);
            H[V[k].y] = V[k].z;
        }
        if(j < V.size()) res += plane * (V[i].x - V[j].x);
        else res += plane * V[i].x;
    }
    cout << res - 1;
}
```

BOJ 28111. 평범한 그래프와 이상한 쿼리

a에서 b로 가는 아무 경로를 하나 선택하고 그 경로의 길이를 d라고 합시다. 경로의 길이를 k의 배수로 만들기 위해서는 길이가 d인 경로에 사이클을 적당히 추가해야 합니다. 즉, 그래프 상에 존재하는 모든 사이클의 길이를 $c_1, c_2, \cdots, c_t$라고 하면, $d + \sum_{i=1}^{t} x_i c_i \equiv 0 \pmod{k}$가 되도록 하는 $x_1, x_2, \cdots, x_t$가 존재해야 합니다. 이 합동식은 $d + \sum_{i=1}^{t} x_i c_i = yk$로 쓸 수 있고, 이 방정식은 $\gcd(k, c_1, c_2, \cdots, c_t) | d$일 때만 정수해 $x_1, x_2, \cdots, x_t, y$가 존재합니다. 따라서 모든 사이클의 길이를 구할 수 있으면 문제를 해결할 수 있습니다.

그래프에서 임의의 스패닝 트리를 하나 만들어 봅시다. 스패닝 트리에 포함되지 않은 간선을 하나 추가해서 만들어지는 사이클들을 **fundamental cycles**라고 부릅니다. 이때 fundamental cycles는 무향 그래프의 사이클을 모두 모아놓은 cycle space의 basis이기 때문에, 모든 사이클의 길이의 선형 결합은 fundamental cycles의 길이의 선형 결합만으로 표현할 수 있습니다.

따라서 그래프를 컴포넌트마다 나눈 다음 스패닝 트리를 만들고, d를 구하는 것은 a에서 루트까지의 거리와 b에서 루트까지의 거리의 합, 모든 사이클의 길이의 최대공약수는 fundamental cycles의 길이의 최대공약수를 계산하는 것으로 문제를 해결할 수 있습니다.

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int N, M, Q, P[101010];
ll D[101010], W[101010];
vector<pair<int,ll>> G[101010];
vector<tuple<int,int,ll>> E, F;

int Find(int v){ return v == P[v] ? v : P[v] = Find(P[v]); }
bool Merge(int u, int v){ return Find(u) != Find(v) && (P[P[u]]=P[v], true); }
void DFS(int v, int b=-1, ll d=0){
    D[v] = d;
    for(auto [i,w] : G[v]) if(i != b) DFS(i, v, d+w);
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> Q; E.resize(M); iota(P+1, P+N+1, 1);
    for(auto &[u,v,w] : E) cin >> u >> v >> w;
    for(const auto &[u,v,w] : E){
        if(Merge(u, v)) G[u].emplace_back(v, w), G[v].emplace_back(u, w);
        else F.emplace_back(u, v, w);
    }
    for(int i=1; i<=N; i++) if(i == Find(i)) DFS(i);
    for(const auto &[u,v,w] : E) W[Find(u)] = __gcd(W[Find(u)], w * 2);
    for(const auto &[u,v,w] : F) W[Find(u)] = __gcd(W[Find(u)], abs(D[u] - D[v])
+ w);
    for(int q=1; q<=Q; q++){
        ll a, b, k; cin >> a >> b >> k;
        if(Find(a) != Find(b)){ cout << "No\n"; continue; }
        ll id = Find(a), len = D[a] + D[b];
        if(len % __gcd(W[id], k) == 0) cout << "Yes\n";
        else cout << "No\n";
    }
}
```

BOJ 3311. Traffic

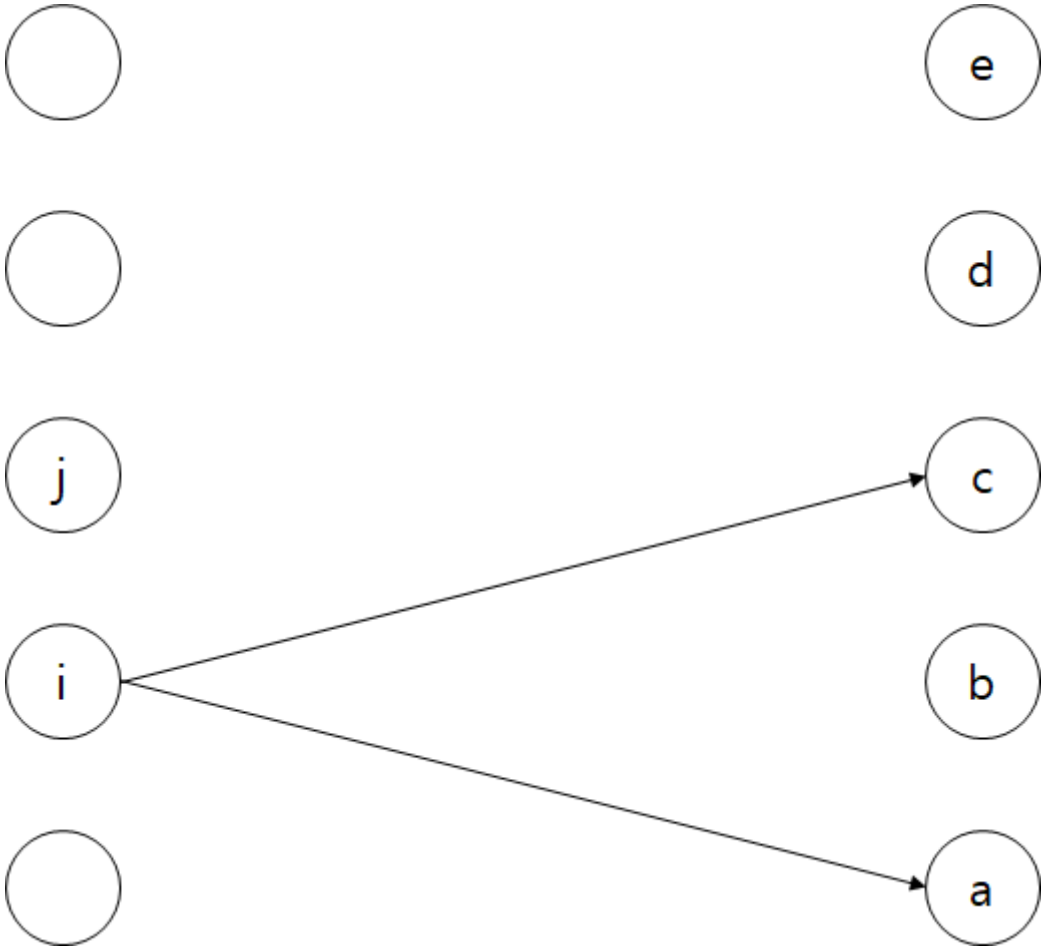
주어지는 그래프가 평면 그래프라는 것에서 풀이가 시작합니다.

먼저 왼쪽 정점에서 도달할 수 없는 오른쪽 정점을 제거하고, 마찬가지로 도달할 수 있는 오른쪽 정점이 없는 왼쪽 정점도 제거한 상태에서 시작합시다.
제거된 정점들은 0을 출력하면 됩니다. 정점을 제거하는 작업은 주어진 그래프에서 DFS/BFS를 돌리고, 간선을 모두 뒤집은 다음에 다시 한 번 DFS/BFS를 돌리는 것으로 쉽게 처리할 수 있습니다.

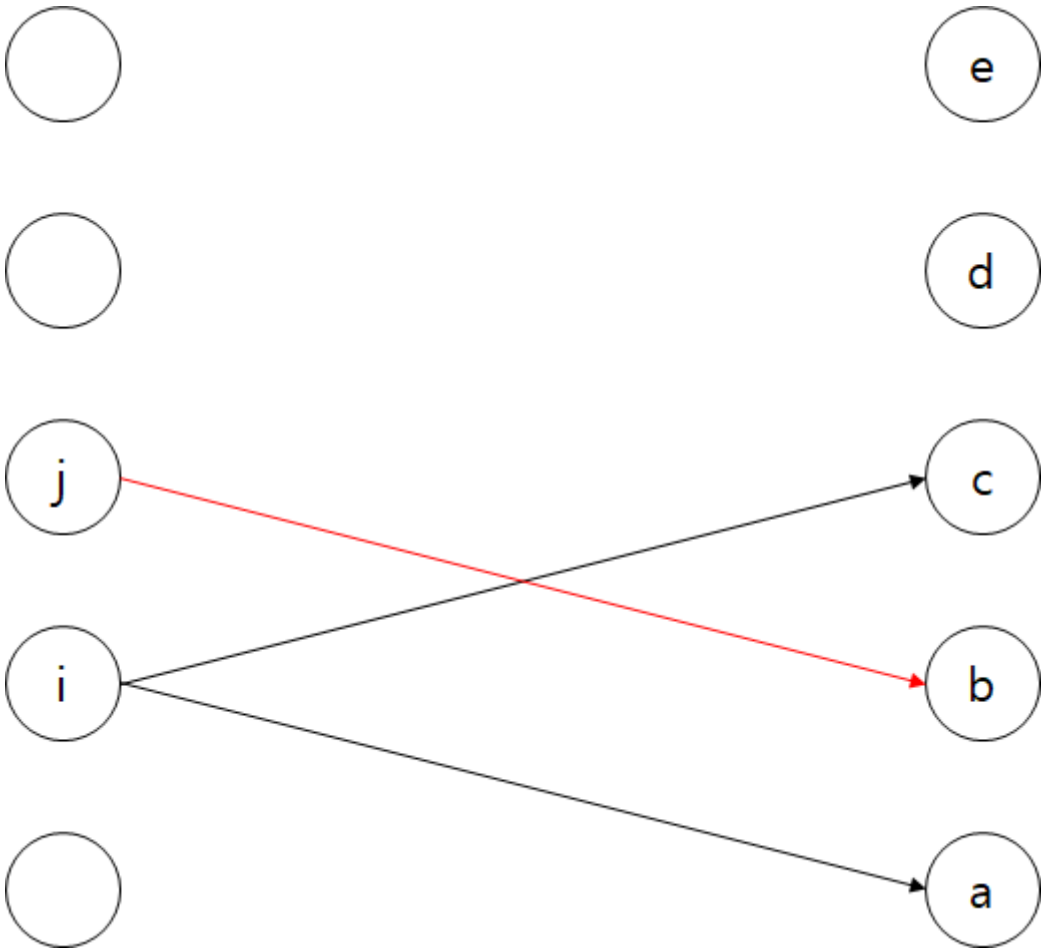
필요 없는 정점들을 제거한 다음, 왼쪽 정점과 오른쪽 정점을 각각 y좌표 기준으로 정렬해서 보면 아래 3가지 성질을 알 수 있습니다.

가장 먼저, 왼쪽 정점에서 도달 가능한 오른쪽 정점은 연속적입니다.
오른쪽 정점을 y좌표 기준으로 정렬합시다. 어떤 왼쪽 정점 L에서 $i \le j$인 i번째 오른쪽 정점과 j번째 오른쪽 정점을 모두 갈 수 있으면, L에서 $i \le x \le j$인 x번째 오른쪽 정점에 모두 도달할 수 있습니다. 이것은 평면 그래프를 직접 그려보면 알 수 있습니다.

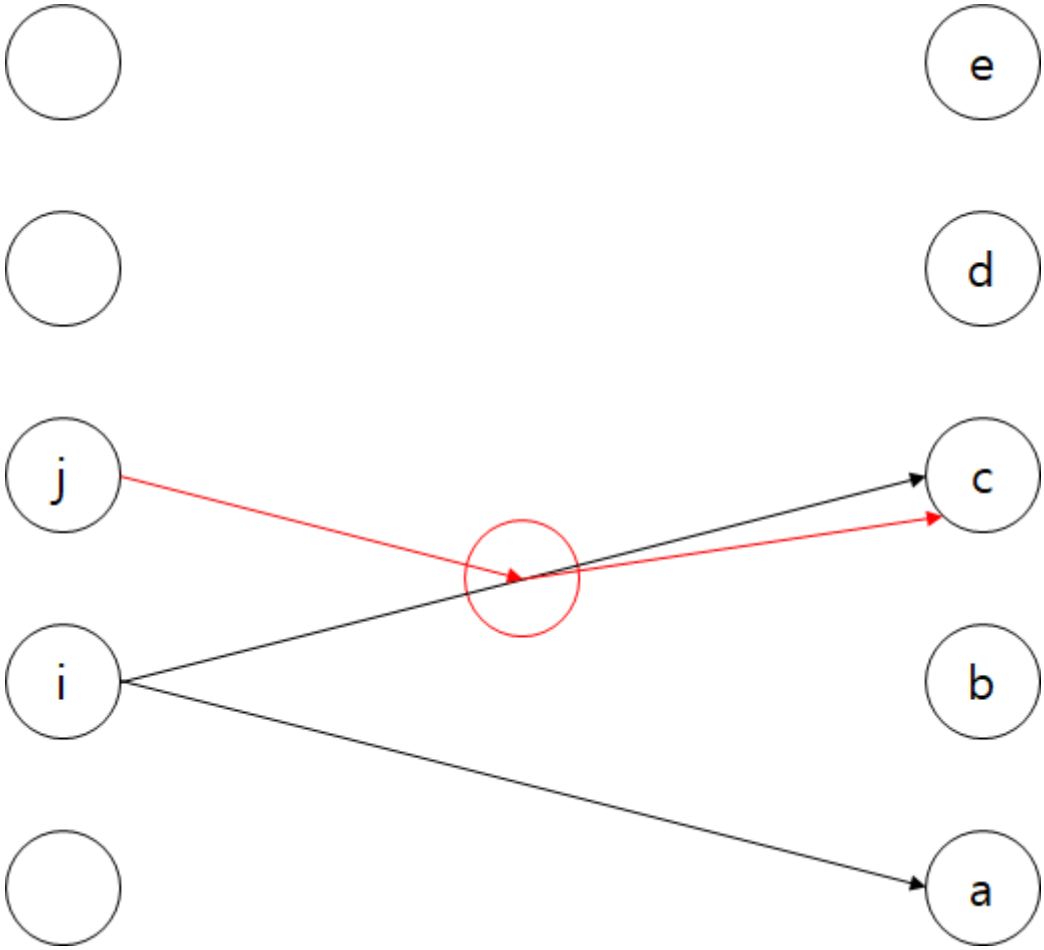
두 번째로, 각 왼쪽 정점에서 도달 가능한 오른쪽 정점의 y좌표 최댓값은 단조증가합니다.



위 그림에서 i번째 정점이 a번째 정점부터 c번째 정점까지 갈 수 있으면, j는 c와 같거나 더 위에 있는 정점까지 도달 가능합니다. 귀류법을 이용해서 y좌표의 최댓값이 단조 증가하지 않는다고 가정해 봅시다. 즉, j는 c, 그리고 c보다 위에 잇는 정점에 도달할 수 없다고 가정해보겠습니다.



j는 c에 갈 수 없고 c보다 아래에 있는 정점 a, b만 갈 수 있다고 가정하면, $i \to c$ 경로와 $j \to b$ 경로 사이에에 교점이 생기게 됩니다. 문제에서 주어지는 그래프는 평면 그래프이기 때문에 교점이 발생하는 곳에 정점이 있어야 하며, 그 지점에 정점을 만들면 j에서 c에 도달할 수 있게 됩니다.



마지막으로, 각 왼쪽 정점에서 도달 가능한 오른쪽 정점의 y좌표 최솟값은 단조 감소합니다. 동일한 방법으로 증명할 수 있습니다.

따라서 왼쪽 정점마다 도달할 수 있는 오른쪽 정점의 y좌표 최솟값과 최댓값만 구하면 문제를 해결할 수 있습니다.

왼쪽 정점마다 DFS를 돌려서 upper_bound와 lower_bound를 구할 것입니다. 그냥 돌리면 $O(V(V+E))$이기 때문에 시간 초과가 발생합니다.

우리는 위에서 y좌표의 최댓값과 최솟값이 단조성을 갖는다는 것을 관찰했기 때문에, DFS를 돌면서 방문한 정점을 굳이 다시 방문할 필요가 없습니다. 이를 이용하면 모든 왼쪽 정점에서 도달 가능한 y좌표 최댓값과 최솟값을 $O(V+E)$ 시간에 구할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;

int N, M, XV, YV, A[303030], X[303030], Y[303030];
vector<int> L, R, G1[303030], G2[303030];
void AddEdge(int s, int e){ G1[s].push_back(e); G2[e].push_back(s); }

int C[303030], Del[303030], Up[303030], Dw[303030], ID[303030];

void Remove(const vector<int> &st, const vector<int> &ed, vector<int> *gph){
    queue<int> Q;
    memset(C, 0, sizeof C);
    for(auto i : st) Q.push(i), C[i] = 1;
```

```
    while(!Q.empty()){
        int v = Q.front(); Q.pop();
        for(auto i : gph[v]) if(!C[i]) C[i] = 1, Q.push(i);
    }
    for(auto i : ed) if(!C[i]) Del[i] = 1;
}

int DFS(int v, bool up){
    int res = A[v] == 2 ? v : -1; C[v] = 1;
    for(auto i : G1[v]){
        if(C[i]) continue;
        int nxt = DFS(i, up);
        if(nxt == -1) continue;
        if(res == -1 || (Y[res] < Y[nxt]) == up) res = nxt;
    }
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> XV >> YV;
    for(int i=1; i<=N; i++) cin >> X[i] >> Y[i];
    for(int i=1; i<=N; i++) if(X[i] == 0) L.push_back(i), A[i] = 1;
    for(int i=1; i<=N; i++) if(X[i] == XV) R.push_back(i), A[i] = 2;
    for(int i=1; i<=M; i++){
        int u, v, w; cin >> u >> v >> w;
        AddEdge(u, v); if(w == 2) AddEdge(v, u);
    }

    Remove(L, R, G1); Remove(R, L, G2); L.clear(); R.clear();
    for(int i=1; i<=N; i++) if(!Del[i] && X[i] == 0) L.push_back(i);
    for(int i=1; i<=N; i++) if(!Del[i] && X[i] == XV) R.push_back(i);
    sort(L.begin(), L.end(), [](int a, int b){ return Y[a] < Y[b]; });
    sort(R.begin(), R.end(), [](int a, int b){ return Y[a] < Y[b]; });
    for(int i=0; i<R.size(); i++) ID[R[i]] = i + 1;

    memset(C, 0, sizeof C);
    for(int i=0; i<L.size(); i++){
        int res = DFS(L[i], true);
        Up[L[i]] = res != -1 ? ID[res] : Up[L[i-1]];
    }

    memset(C, 0, sizeof C);
    for(int i=(int)L.size()-1; i>=0; i--){
        int res = DFS(L[i], false);
        Dw[L[i]] = res != -1 ? ID[res] : Dw[L[i+1]];
    }

    L.clear();
    for(int i=1; i<=N; i++) if(X[i] == 0) L.push_back(i);
    sort(L.begin(), L.end(), [](int a, int b){ return Y[a] > Y[b]; });
    for(auto i : L) cout << (Del[i] ? 0 : Up[i] - Dw[i] + 1) << "\n";
}
```