2023 SCCC 봄 #10

BOJ 2194. 유닛 이동시키기

간단한 BFS 문제입니다. 유닛의 좌측 상단 위치가 (r, c)에 위치할 수 있으면 1, 그렇지 않으면 0을 저장하는 이차원 배열 $F(r, c)$ 같은 것을 관리하면 쉽게 구현할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
constexpr int di[] = {1, -1, 0, 0};
constexpr int dj[] = {0, 0, 1, -1};

int N, M, R, C, K, A[555][555], F[555][555], D[555][555];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> R >> C >> K;
    for(int i=1,r,c; i<=K; i++) cin >> r >> c, A[r][c] = 1;
    for(int i=1; i<=N; i++) for(int j=1; j<=M; j++) A[i][j] += A[i-1][j] + A[i]
[j-1] - A[i-1][j-1];
    for(int i=R; i<=N; i++) for(int j=C; j<=M; j++) if(A[i][j] - A[i-R][j] -
A[i][j-C] + A[i-R][j-C] == 0) F[i-R+1][j-C+1] = 1;
    int Si, Sj, Ti, Tj; cin >> Si >> Sj >> Ti >> Tj;
    if(F[Si][Sj] == 0 || F[Ti][Tj] == 0){ cout << -1; return 0; }

    memset(D, -1, sizeof D);
    queue<pair<int,int>> Q; Q.emplace(Si, Sj); D[Si][Sj] = 0;
    while(!Q.empty()){
        auto [i,j] = Q.front(); Q.pop();
        if(i == Ti && j == Tj){ cout << D[i][j]; return 0; }
        for(int k=0; k<4; k++){
            int r = i + di[k], c = j + dj[k];
            if(!F[r][c] || D[r][c] != -1) continue;
            D[r][c] = D[i][j] + 1; Q.emplace(r, c);
        }
    }
    cout << -1;
}
```

BOJ 12908. 텔레포트 3

시작점, 도착점, 텔레포트 좌표를 정점으로 하는 그래프를 만든 다음, 플로이드 와샬 알고리즘 등을 이용해 최단 거리를 구하면 됩니다.

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

ll X[8], Y[8], G[8][8];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> X[6] >> Y[6] >> X[7] >> Y[7];
```

```
    for(int i=0; i<6; i++) cin >> X[i] >> Y[i];
    memset(G, 0x3f, sizeof G);
    for(int i=0; i<8; i++) for(int j=0; j<8; j++) G[i][j] = abs(X[i] - X[j]) +
abs(Y[i] - Y[j]);
    for(int i=0; i<3; i++){
        int u = i * 2, v = u + 1;
        G[u][v] = min(G[u][v], 10LL);
        G[v][u] = min(G[v][u], 10LL);
    }
    for(int k=0; k<8; k++) for(int i=0; i<8; i++) for(int j=0; j<8; j++) G[i][j]
= min(G[i][j], G[i][k] + G[k][j]);
    cout << G[6][7];
}
```

BOJ 27988. 리본 (Hard)

절댓값을 다루는 것은 매우 귀찮으므로 $X_i < X_j$인 순서쌍 (i, j)만 고려합시다. $X_j - X_i \le L_i + L_j$인 순서쌍이 존재하는지 확인하면 되고, 식을 조금 변형하면 $X_j - L_j \le X_i + L_i$를 만족하는 순서쌍이 존재하는지 확인하면 됩니다.

리본을 순서대로 보면서, 각 색깔마다 $X_i + L_i$의 최댓값을 관리합시다. 만약 j번째 리본과 색깔이 다른 리본 중 $X_i + L_i$가 $X_j - L_j$보다 크거나 같은 리본이 하나라도 있으면 순서쌍을 찾은 것이기 때문에 최댓값만 들고 있어도 됩니다.

각 색깔마다 $X_i + L_i$의 최댓값과 최대가 되는 리본의 인덱스를 관리하면 $O(N)$ 시간에 문제를 해결할 수 있습니다.

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

ll N, X[1010101], L[1010101], C[1010101];
pair<ll,ll> V[3];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> X[i];
    for(int i=1; i<=N; i++) cin >> L[i];
    for(int i=1; i<=N; i++){ char c; cin >> c; C[i] = string("RYB").find(c); }
    fill(V, V+3, make_pair(-1e18, -1e18));
    for(int i=1; i<=N; i++){
        for(int j=0; j<3; j++){
            if(C[i] == j){ V[j] = max(V[j], make_pair(X[i]+L[i], (ll)i));
continue; }
            if(X[i] - L[i] <= V[j].first){ cout << "YES\n" << V[j].second << " "
<< i; return 0; }
        }
    }
    cout << "NO";
}
```

BOJ 2135. 문자열 압축하기

전형적인 구간 DP 문제입니다.

문자열의 구간 $[i, j]$를 나타낼 수 있는 가장 짧은 방법의 길이를 $D(i, j)$라고 정의합시다. 만약 $[i, j]$ 전체를 하나의 $k(S)$ 형태로 압축하지 않는다면 $D(i, j) = \min_{i \le k < j} \{D(i, k) + D(k+1, j)\}$입니다.

$[i, j]$를 압축하기 위해서는 $[i, j]$가 주기를 가져야 합니다. 만약 $[i, j]$ 구간이 길이가 $k | (j-i+1)$인 문자열이 반복되는 형태라면 $D(i, j) \leftarrow D(i, i+k-1) + 2 + len((j-i+1)/k)$를 시도할 수 있습니다.

모든 구간의 주기를 구하는 것은 naive하게 구현하거나 KMP를 이용하면 $O(N^3)$에 할 수 있습니다. N의 약수의 개수를 약 $O(N^{1/3})$ 정도로 생각한다면, DP를 계산하는 것은 $O(N^{10/3})$ 정도 걸립니다.

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

vector<int> GetFail(const string &s){
    int n = s.size();
    vector<int> fail(n);
    for(int i=1, j=0; i<n; i++){
        while(j > 0 && s[i] != s[j]) j = fail[j-1];
        if(s[i] == s[j]) fail[i] = ++j;
    }
    return fail;
}

bool Check(const string &s, int len){
    int n = s.size();
    if(n % len != 0) return false;
    for(int i=len; i<n; i+=len) if(s.substr(0, len) != s.substr(i, len)) return
false;
    return true;
}

vector<int> Period(const string &s){
    vector<int> res, fail = GetFail(s);
    int n = s.size();
    for(int i=fail[n-1]; i; i=fail[i-1]) if(Check(s, i)) res.push_back(i);
    return res;
}

int N, D[222][222];
string S;
vector<int> V[222][222];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> S; N = S.size(); S = "#" + S;
    for(int i=1; i<=N; i++) for(int j=i; j<=N; j++) V[i][j] = Period(S.substr(i,
j-i+1));
    memset(D, 0x3f, sizeof D);
    for(int i=1; i<=N; i++) D[i][i] = 1;
    for(int d=1; d<=N; d++){
        for(int i=1, j=d+1; j<=N; i++, j++){
            for(int k=i; k<j; k++) D[i][j] = min(D[i][j], D[i][k] + D[k+1][j]);
            for(auto k : V[i][j]) D[i][j] = min<int>(D[i][j], D[i][i+k-1] + 2 +
to_string((j-i+1)/k).size());
        }
```

```
    }
    cout << D[1][N];
}
```

BOJ 26434. Happy Subarrays

모든 prefix sum이 0 이상인 subarray의 개수를 세는 문제입니다. 투 포인터를 사용하거나, 각 시작점마다 이분 탐색을 이용해 끝점을 구하는 방식으로 해결할 수 있다는 것은 어렵지 않게 알 수 있습니다. 후자의 방법을 설명하겠습니다.

괄호 문자열 관련 문제를 푸는 것처럼 (현재까지의 합)과 (현재까지의 누적 합의 최솟값)을 함께 관리합니다. 두 구간의 정보를 연결하는 연산은 결합 법칙이 성립하기 때문에 세그먼트 트리와 같은 자료구조를 이용해 특정 구간의 모든 prefix sum이 0 이상인지 효율적으로 확인할 수 있습니다. 따라서 이분 탐색을 할 때마다 세그먼트 트리에 쿼리를 날리면 $O(N \log^2 N)$에 해결할 수 있습니다.

로그를 하나 떼는 방법도 있는데, 스파스 테이블을 이용해서 이분 탐색을 수행하면 됩니다. 자세한 방법은 아래 코드를 참고하세요.

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

struct Info{
    int mn, sum;
    Info() : Info(0) {};
    Info(int v) : Info(min(0,v), v) {}
    Info(int mn, int sum) : mn(mn), sum(sum) {}
    Info operator + (const Info &t) const {
        return { min(mn, sum + t.mn), sum + t.sum };
    }
};

int N, A[404040];
ll S[404040], X[404040];
Info P[19][404040];

ll Query(int l, int r){
    // r -> N-r+1 -> 1
    return X[r] - X[l-1] - (S[r] - S[l-1]) * (N - r);
}

void Solve(){
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++) S[i] = S[i-1] + A[i];
    for(int i=1; i<=N; i++) X[i] = X[i-1] + 1LL * (N - i + 1) * A[i];
    for(int i=1; i<=N; i++) P[0][i] = A[i];
    for(int i=1; i<19; i++) for(int j=1; j<=N; j++) if(j + (1<<i) - 1 <= N) P[i]
[j] = P[i-1][j] + P[i-1][j+(1<<(i-1))];

    ll R = 0;
    for(int i=1; i<=N; i++){
        if(A[i] < 0) continue;
        int x = i; Info now;
        for(int i=18; i>=0; i--) if(x + (1<<i) - 1 <= N && (now + P[i][x]).mn >=
0) now = now + P[i][x], x += 1 << i;
        R += Query(i, x-1);
```

```
    }
    cout << R << "\n";
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int TC; cin >> TC;
    for(int tc=1; tc<=TC; tc++) cout << "Case #" << tc << ": ", Solve();
}
```

BOJ 24165. 直線 (Lines)

오일러 지표에 의해 평면의 개수는 (선분 개수) - (교차점 개수) + 1입니다. 정확한 공식을 모르더라도 오일러 지표와 관련있다는 점만 알아차리면 그림을 몇 번 그려보는 것으로 어렵지 않게 알 수 있습니다.

직선이 최대 1000개밖에 주어지지 않으므로 $O(N^2)$ 시간에 모든 교차점을 확인해도 됩니다.

```
#include <bits/stdc++.h>
using namespace std;
using ld = long double;
constexpr ld EPS = 1e-8;

struct Line{
    ld a, b; int flag;
    Line() : Line(0, 0) {}
    Line(ld x) : a(0), b(x), flag(1) {}
    Line(ld a, ld b) : a(a), b(b), flag(0) {}
    bool operator < (const Line &l) const {
        if(flag != l.flag) return flag < l.flag;
        if(abs(a - l.a) > EPS) return a < l.a;
        return b + EPS < l.b;
    }
    bool operator == (const Line &l) const {
        if(flag != l.flag) return false;
        return abs(a - l.a) < EPS && abs(b - l.b) < EPS;
    }
    ld f(ld x){ return a * x + b; }
};

tuple<bool, ld, ld> Check(Line u, Line v){
    if(u.flag) swap(u, v);
    if(u.flag && v.flag) return {false, 0, 0};
    if(v.flag) return {true, v.b, u.f(v.b)};
    if(abs(u.a - v.a) < EPS) return {false, 0, 0};
    ld x = (u.b - v.b) / (v.a - u.a);
    assert(abs(u.f(x) - v.f(x)) < EPS);
    return {true, x, u.f(x)};
}

bool Less(pair<ld, ld> a, pair<ld, ld> b){
    if(abs(a.first - b.first) > EPS) return a.first < b.first;
    return a.second + EPS < b.second;
}

bool Equal(pair<ld, ld> a, pair<ld, ld> b){
    return abs(a.first - b.first) < EPS && abs(a.second - b.second) < EPS;
}
```

```
int UniquePoints(vector<pair<ld,ld>> &v){
    sort(v.begin(), v.end(), Less);
    v.erase(unique(v.begin(), v.end(), Equal), v.end());
    return v.size();
}

int Solve(int N, vector<tuple<int,int,int,int>> lines){
    vector<Line> A; A.reserve(N);
    assert(lines.size() == N);
    for(const auto &[x1,y1,x2,y2] : lines){
        if(x1 == x2) A.emplace_back(x1);
        else{
            ld m = 1.L * (y2 - y1) / (x2 - x1);
            A.emplace_back(m, y1 - m * x1);
        }
    }
    sort(A.begin(), A.end());
    A.erase(unique(A.begin(), A.end()), A.end());
    N = A.size();

    int E = N;
    vector<pair<ld, ld>> V;

    for(auto i : A){
        vector<pair<ld, ld>> points;
        for(auto j : A){
            if(i == j) continue;
            auto [flag,x,y] = Check(i, j);
            if(flag) V.emplace_back(x, y), points.emplace_back(x, y);
        }
        E += UniquePoints(points);
    }
    UniquePoints(V);
    return E - V.size() + 1;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int N; cin >> N;
    vector<tuple<int,int,int,int>> V(N);
    for(auto &[a,b,c,d] : V) cin >> a >> b >> c >> d;
    cout << Solve(N, V);
}
```

BOJ 12771. Oil

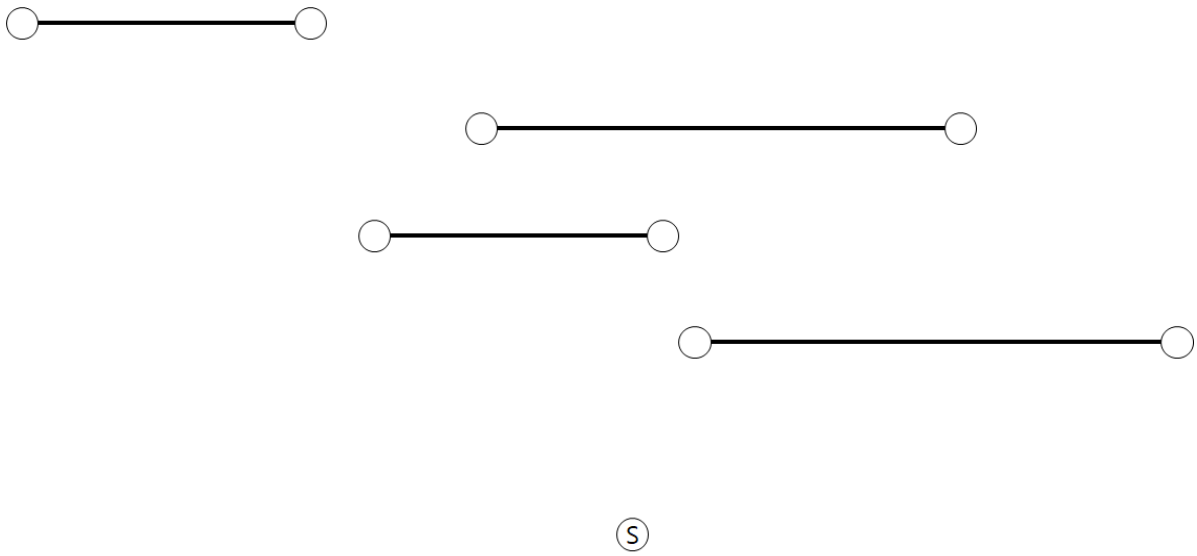
지표면에서부터 직선을 긋는 것이 아닌, 땅속 어느 점에서 시작해 기울기가 0이 아닌 직선을 긋는다고 생각해 봅시다.

(x1, x2, y)선분 위의 점에서 시작해 직선을 그을 때는 (x1, y), (x2, y)처럼 선분의 끝부분에서 직선을 긋는 것이 이득이라는 것을 관찰할 수 있습니다. 이 사실을 이용하면 직선의 시작점으로 2N개만 보면 된다는 사실을 알 수 있습니다 (선분 N개의 양 끝점).

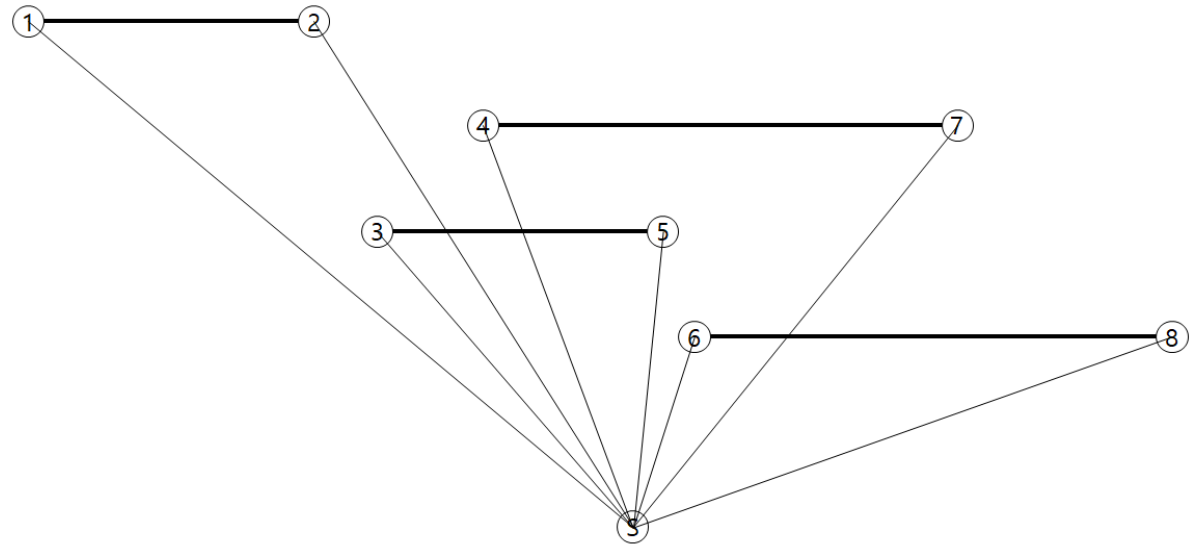
각 시작점에서의 최댓값을 찾는 방법을 생각해 봅시다.

직선을 그을 것이기 때문에, 시작점보다 아래쪽에 있는 점은 시작점을 기준으로 점대칭 이동을 해도 상관 없습니다. 또한 기울기가 0이 아닌 직선을 그을 것이기 때문에 시작점과 y좌표가 같은 선분은 볼 필요가 없습니다.

모든 점들을 시작점보다 위에 오도록 세팅을 하고 문제를 풀어봅시다.



모든 점들을 시작점(S)보다 위로 올렸으니, 대충 위 그림과 같은 상황일 것입니다. 이제 시작점을 기준으로 직선을 여러 개 그어서 그중 최댓값을 찾을 것입니다. 구체적으로, 직접 긋는 것이 아니라 **스위핑** 하면서 최댓값만 구할 것입니다.

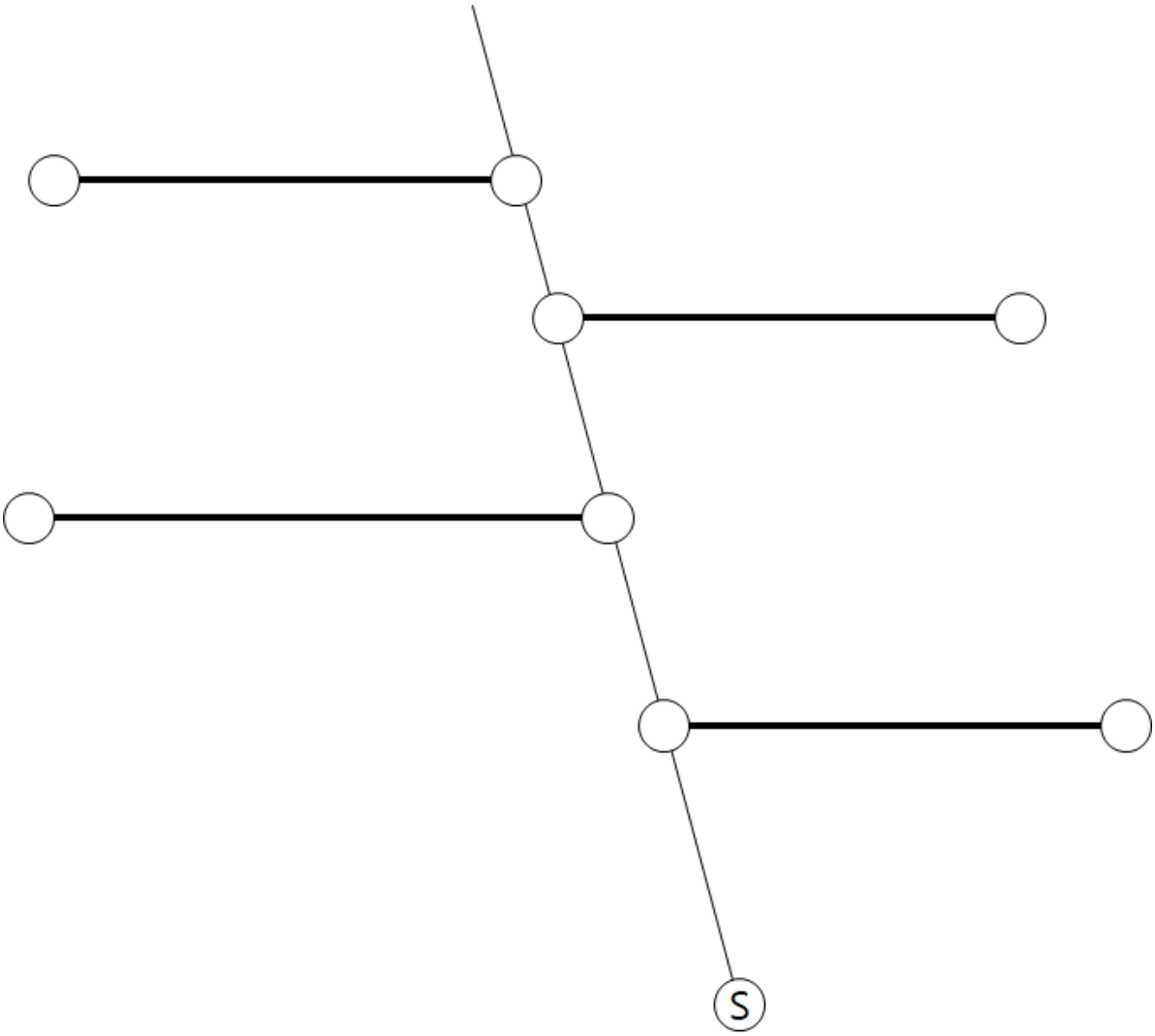


스위핑을 하기 위해 위 그림처럼 각도 순으로 점들을 정렬합시다. 각도 순으로 점들의 번호를 붙여주면 각 선분을 (1, 2), (3, 5), (4, 7), (6, 8)로 표현할 수 있습니다.

여기에서 알 수 있는 사실은, S와 1번 점을 잇는 각도와 2번 점을 잇는 각도 사이에서 직선을 그으면 선분 (1, 2)를 지나는 직선이 됩니다. 마찬가지로 4번 점을 잇는 각도와 7번 점을 잇는 각도 사이에서 직선을 그으면 선분 (4, 7)을 지나는 직선이 됩니다.

그러므로 점들을 각도 순으로 보면서 각 선분마다 점을 처음 만나면 현재 값에 선분의 길이를 더해주고, 두 번째로 만다면 선분의 길이를 빼줄 수 있습니다. 당연히 최댓값은 점들을 하나씩 볼 때마다 갱신하면서 구할 수 있습니다.

이렇게 하면 잘 해결될 것 같았지만, 아래 그림과 같은 반례가 있습니다.



이렇게 여러 개의 점이 같은 각도에 존재하는 경우에는 더하는 이벤트를 모두 처리한 다음에 최댓값을 갱신하고, 빼는 이벤트는 나중에 처리해야 합니다. 각 점을 만날 때마다 더해야 하는 값들을 저장한 뒤, 각도가 같은 경우에는 더해야 하는 값이 큰 점을 먼저 보도록 정렬하면 됩니다.

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

struct Point{
    ll x, y, i, d;
    constexpr Point() : Point(0, 0, 0, 0) {}
    constexpr Point(ll x, ll y) : x(x), y(y), i(0), d(0) {}
    constexpr Point(ll x, ll y, ll i, ll d) : x(x), y(y), i(i), d(d) {}
};
constexpr Point operator + (const Point &p1, const Point &p2){ return {p1.x +
p2.x, p1.y + p2.y}; }
constexpr Point operator - (const Point &p1, const Point &p2){ return {p1.x -
p2.x, p1.y - p2.y}; }
constexpr Point operator * (const Point &pt, const ll v){ return {pt.x * v, pt.y
* v}; }
constexpr ll operator * (const Point &p1, const Point &p2){ return p1.x * p2.x +
p1.y * p2.y; }
constexpr ll operator / (const Point &p1, const Point &p2){ return p1.x * p2.y -
p1.y * p2.x; }
```

```cpp
constexpr int Sign(ll v){ return (v > 0) - (v < 0); }
constexpr int CCW(const Point &p1, const Point &p2, const Point &p3){ return
Sign((p2 - p1) / (p3 - p1)); }
constexpr int QuadrantID(const Point& p){
    constexpr int arr[9] = { 5, 4, 3, 6, -1, 2, 7, 0, 1 };
    return arr[Sign(p.x)*3+Sign(p.y)+4];
}

int N, C[2020];
vector<Point> Inp;

long long Solve(const Point o){
    vector<Point> V;
    for(auto i : Inp) if(o.y != i.y) V.push_back(i);
    for(auto &p : V){
        if(p.y < o.y){
            auto t = o * 2 - p;
            p.x = t.x; p.y = t.y;
        }
    }
    sort(V.begin(), V.end(), [&](const auto &p1, const auto &p2){
        return QuadrantID(p1-o) != QuadrantID(p2-o) ? QuadrantID(p1-o) <
QuadrantID(p2-o) : CCW(o, p1, p2) > 0;
    });
    memset(C, 0, sizeof C);
    for(auto &i : V) if(!C[i.i]) C[i.i] = 1; else i.d *= -1;
    sort(V.begin(), V.end(), [&](const auto &p1, const auto &p2){
        if(QuadrantID(p1-o) != QuadrantID(p2-o)) return QuadrantID(p1-o) <
QuadrantID(p2-o);
        if(int cw=CCW(o,p1,p2); cw) return cw > 0;
        return p1.d > p2.d;
    });
    long long now = o.d, mx = o.d;
    for(auto i : V) mx = max(mx, now += i.d);
    return mx;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++){
        int x1, x2, y; cin >> x1 >> x2 >> y;
        if(x1 > x2) swap(x1, x2);
        Inp.emplace_back(x1, y, i, x2 - x1);
        Inp.emplace_back(x2, y, i, x2 - x1);
    }
    ll R = 0;
    for(int i=0; i<Inp.size(); i++) R = max(R, Solve(Inp[i]));
    cout << R;
}
```

BOJ 8131. Ploughing

열심히 직사각형을 깎아내다 보면, 마지막에는 가로 혹은 세로 길이가 1인 직사각형을 없애게 됩니다. 마지막에 없애는 직사각형의 크기를 $1 \times K$ 또는 $K \times 1$이라고 하면, 직사각형을 총 $(N-1)+(M-K)$ 또는 $(N-K)+(M-1)$번 없애게 됩니다. $(N+M)$은 일정하므로 K를 최대화시키는 것이 이 문제의 목적입니다.

마지막에 가로로 길쭉한 직사각형을 하나 남긴다고 생각합시다. 세로로 길쭉한 직사각형을 남기는 것이 최적인 경우에는 배열을 90도 회전한 다음에 똑같은 과정을 거치면 됩니다.

마지막까지 남겨놓을 가로로 길쭉한 직사각형의 양쪽 끝 지점을 고정하면, 2차원 누적 합 배열을 이용해 그런 직사각형을 남겨놓을 수 있는지 $O(N+M)$에 판별할 수 있습니다. 총 $O(M^2)$가지의 경우가 있으므로 $O(M^2(N+M))$정도에 문제를 풀 수 있지만 N과 M이 너무 큽니다.

하지만 투포인터 느낌으로 끝 지점을 관리하면 총 $O(M)$개의 경우만 봐도 되고, $O(M(N+M))$에 해결할 수 있습니다.

배열을 회전시키고 이 작업을 한 번 더 수행하면 $O(N^2+M^2)$ 시간에 문제를 해결할 수 있습니다.

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

ll N, M, K, A[2020][2020], B[2020][2020], S[2020][2020];

void Rotate(){
    memcpy(B, A, sizeof B);
    for(int i=1; i<=M; i++) for(int j=1; j<=N; j++) A[i][j] = B[N-j+1][i];
    swap(N, M);
}

void Build(){
    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] + A[i][j];
}

ll Sum(int r1, int r2, int c1, int c2){
    return S[r2][c2] - S[r1-1][c2] - S[r2][c1-1] + S[r1-1][c1-1];
}

bool Check(int l, int r){
    int len = r - l + 1, cut = N + M - len - 1;
    int r1 = 1, r2 = N, c1 = 1, c2 = M;
    for(int iter=0; iter<cut; iter++){
        if(r1 < r2 && Sum(r1, r1, c1, c2) <= K) r1++;
        else if(r1 < r2 && Sum(r2, r2, c1, c2) <= K) r2--;
        else if(c1 < l && Sum(r1, r2, c1, c1) <= K) c1++;
        else if(c2 > r && Sum(r1, r2, c2, c2) <= K) c2--;
        else return false;
    }
    return Sum(r1, r2, c1, c2) <= K;
}

int Solve(){
    Build();
    int l = 1, r = 1, res = 1e9;
    while(true){
```

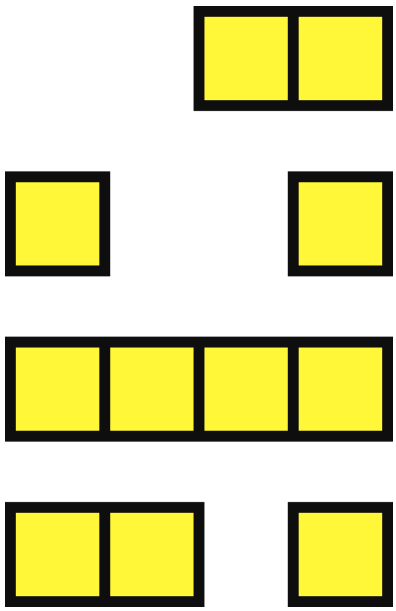
```
        bool flag = Check(l, r);
        if(flag) res = min<ll>(res, N + M - (r - l + 1));
        if(l == M) break;
        else if(r == M) l++;
        else if(l == r) r++;
        else if(flag) r++;
        else l++;
    }
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> K >> M >> N;
    for(int i=1; i<=N; i++) for(int j=1; j<=M; j++) cin >> A[i][j];
    int R = Solve();
    Rotate();
    R = min(R, Solve());
    cout << R;
}
```

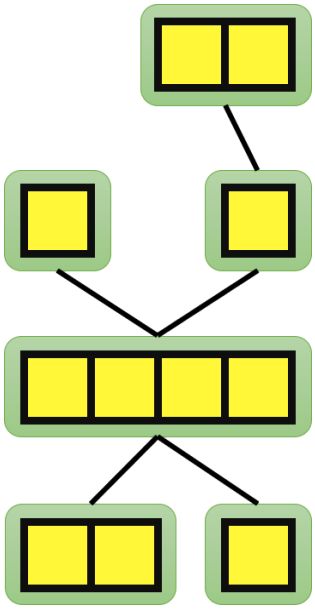
BOJ 5813. 이상적인 도시

문제에 주어진 조건을 보면 블록이 있는 칸끼리 서로 연결되어 있고, 그렇지 않은 칸끼리도 서로 연결되어 있습니다. 따라서 블록이 있는 칸 사이에는 경로가 항상 존재하고, 블록이 없는 칸 사이에도 항상 경로가 존재합니다.

블록이 있는 칸들을 행 단위로 분할해 봅시다.



연결되어 있는 칸들을 하나의 정점으로 생각하고, 인접한 위/아래로 인접한 칸끼리 간선으로 연결하면 아래와 같은 그래프를 만들 수 있습니다.



이렇게 해서 만들어진 그래프는 문제 조건에 의해 사이클이 없고 모든 정점이 연결된 트리가 됩니다. y좌표의 변화량을 계산하는 것은 위 트리에서 모든 칸 사이의 거리를 구하면 되고, 아래 코드와 같은 DFS를 이용해 $O(N)$에 계산할 수 있습니다.

```
int DFS(int v, int b=-1){
    int res = 0;
    for(auto i : G[v]) if(i != b) res += DFS(i, v), S[v] += S[i];
    res += S[v] * (N - S[v]);
    return res;
}
```

x, y좌표를 바꾼 뒤 똑같이 트리를 만들면 x좌표의 변화량도 계산할 수 있습니다.

트리를 구성하는 방법에 따라 전체 문제를 $O(N)$ 또는 $O(N \log N)$에 해결할 수 있습니다.

```
#include <bits/stdc++.h>
#define x first
#define y second
using namespace std;
using Point = pair<int, int>;
constexpr int MOD = 1'000'000'000;
inline void Add(int &a, const int b){ if((a += b) >= MOD) a -= MOD; }

int N, S[101010], R;
Point A[101010];
vector<int> G[101010];

void MakeTree(){
    sort(A+1, A+N+1);
    memset(S, 0, sizeof S);
    for(int i=1; i<=N; i++) G[i].clear();

    map<Point, int> ID;
    int P = 0; ID[A[1]] = ++P; S[P] = 1;
    for(int i=2; i<=N; i++){
        if(A[i-1].x == A[i].x && A[i-1].y + 1 == A[i].y) ID[A[i]] = P;
        else ID[A[i]] = ++P;
        S[P]++;
    }
```

```
    for(int i=1; i<=N; i++){
        auto [x,y] = A[i]; int id = ID[A[i]];
        for(auto dx : {-1, +1}){
            auto it = ID.find(make_pair(x+dx, y));
            if(it != ID.end() && id != it->second) G[id].push_back(it->second);
        }
    }
    for(int i=1; i<=N; i++){
        sort(G[i].begin(), G[i].end());
        G[i].erase(unique(G[i].begin(), G[i].end()), G[i].end());
    }
}

int GetDist(int v=1, int b=-1){
    int res = 0;
    for(auto i : G[v]) if(i != b) Add(res, GetDist(i, v)), S[v] += S[i];
    res = (res + 1LL * S[v] * (N - S[v])) % MOD;
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i].x >> A[i].y;
    MakeTree();
    Add(R, GetDist());
    for(int i=1; i<=N; i++) swap(A[i].x, A[i].y);
    MakeTree();
    Add(R, GetDist());
    cout << R;
}
```