2023 SCCC 봄 #4

BOJ 27920. 카드 뒤집기

어떤 칸에서 N칸 떨어진 카드는 존재하지 않기 때문에 N이 적혀있는 카드는 가장 마지막에 선택해야 합니다. 비슷한 이유로 $N-1$이 적힌 카드는 반드시 첫 번째나 N번째 칸에 위치해야 하고, 이 카드는 $N-1$번째로 선택해야 합니다. 따라서 $N-1$과 N이 적힌 카드는 각각 맨앞과 맨뒤에 위치해야 합니다.

이 과정을 계속 반복하면 큰 수가 적힌 카드일수록 바깥쪽에 위치해야 한다는 사실을 알아낼 수 있습니다. 따라서 1을 중심에 배치한 뒤, 다음 카드들을 왼쪽과 오른쪽에 번갈아가며 배치하면 정답을 구할 수 있습니다.

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

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int N; cin >> N;

    vector<int> V, P(N+1);
    for(int i=N-1; i>=1; i-=2) V.push_back(i);
    V.push_back(N);
    for(int i=2-N%2; i<N; i+=2) V.push_back(i);
    for(int i=0; i<V.size(); i++) P[V[i]%N] = i + 1;

    cout << "YES\n";
    for(auto i : V) cout << i << " "; cout << "\n";
    for(int i=N-1; i>=0; i--) cout << P[i] << " ";
}
```

BOJ 27896. 특별한 서빙

$i-1$번째 학생까지 파묻튀를 줬을 때 불만도가 M 미만이었고, i번째 학생에게 파묻튀를 줬을 때 처음으로 불만도가 M 이상이 된 상황을 생각해 봅시다. $1, 2, \cdots, i$번째 학생 중 몇 명의 학생에게 파묻튀 대신 가지를 줘서 불만도를 M 미만으로 내려야 하는 상황입니다.

이런 상황에서는 x_j가 가장 큰 학생에게 가지를 주는 것이 최적이라는 것을 직관적으로 알 수 있습니다. x_j가 가장 큰 학생에게 가지를 주면 불만도가 $2x_j$ 만큼 감소하게 되고, $x_j \le x_i$이므로 불만도가 M 미만으로 내려갑니다.

이 문제를 해결하기 위해서는 지금까지 파묻튀를 받은 학생 중 x_j가 가장 큰 학생을 빠르게 찾을 수 있어야 합니다. 원소가 추가되는 상황에서 가장 큰 원소를 찾아서 지우는 연산을 효율적으로 처리하는 자료구조로는 힙(우선순위 큐)가 있습니다. C++의 `std::priority_queue`를 사용하면 $O(N \log N)$에 문제를 해결할 수 있습니다.

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

ll N, M, A[202020], S, R;
priority_queue<ll> Q;
```

```
int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++){
        S += A[i]; Q.push(A[i]);
        while(S >= M) S -= Q.top() * 2, Q.pop(), R++;
    }
    cout << R;
}
```

BOJ 27893. 특별한 드롭킥

장애물을 최대 M개 만큼 추가로 설치해서 $x = N$에 도착하는 비용을 최소화해야 합니다. 장애물을 설치할 때마다 비용이 얼마나 감소하는지 생각해 봅시다.

d칸 떨어져 있는 두 장애물을 d개의 장애물을 추가해서 연결하는 경우를 먼저 보겠습니다. 원래는 $2 + d + 2$ 만큼의 시간이 걸렸지만 d개의 장애물을 추가해서 2로, 총 $d + 2$ 만큼 줄일 수 있습니다. 따라서 이 경우 장애물 하나당 시간이 $1 + 2/d$ 만큼 감소합니다.

떨어져 있는 두 장애물을 연결하는 것이 아닌, $x = 1$ 또는 $x = N$까지 장애물을 잇는 경우를 생각해 봅시다. 즉, d개의 장애물을 이용해 $x = d + 1$에 있는 장애물을 $x = 1$까지 연장하는 것, 또는 $x = N - d$에 있는 장애물을 $x = N$까지 연장하는 경우를 보겠습니다.

원래는 $d + 2$ 만큼의 시간이 걸렸지만 d개의 장애물을 추가해서 2로, 총 d 만큼 감소합니다. 따라서 이 경우 장애물 하나당 시간이 1 만큼 감소합니다.

어떤 장애물 옆에 장애물을 하나 추가로 붙이는 경우를 생각해 봅시다. 원래는 $1 + 2$ 만큼의 시간이 걸렸지만 장애물 1개를 추가해서 2로, 시간이 총 1 만큼 감소합니다.

이제 장애물이 없는 공간에 새로 설치하는 경우만 남았습니다. 빈 공간에 장애물을 d개 설치하면 d 만큼 걸리던 것을 2로 바꿀 수 있고, 이때 장애물 하나당 효율은 $1 - 2/d$입니다. $d < 2$인 경우에는 효율이 음수이므로 설치를 하면 안 됩니다.

따라서 아래 순서대로 효율이 큰 구간부터 차례대로 장애물을 설치하면 시간을 최소화할 수 있습니다.

1. 두 장애물 사이에 있는 구간을 연결, 그런 구간이 여러 개라면 크기가 작은 구간부터 연결
2. 장애물을 연장
3. 빈 공간에 장애물 2개 이상 설치

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

int N, M, A[202020];
vector<pair<int,int>> V;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=N; i++){ char c; cin >> c; A[i] = c == 'X'; }

    int st = -1;
    for(int i=1; i<=N; i++){
        if(A[i] && st != -1) V.emplace_back(st, i-1), st = -1;
        if(!A[i] && st == -1) st = i;
    }
    if(st != -1) V.emplace_back(st, N);
```

```
    if(V.empty()){ cout << 2; return 0; }
    if(N == 1){ cout << 1; return 0; }

    sort(V.begin(), V.end(), [](auto a, auto b){
        if(a.first == 1 || a.second == N) return false;
        if(b.first == 1 || b.second == N) return true;
        return a.second - a.first < b.second - b.first;
    });

    for(auto [s,e] : V){
        if(s == 1){
            for(int i=e; i>=s && M>0; i--) A[i] = 1, M--;
        }
        else{
            for(int i=s; i<=e && M>0; i++) A[i] = 1, M--;
        }
    }

    int R = 0, F = 0;
    for(int i=1; i<=N; i++){
        if(!A[i]) R++, F = 0;
        else if(!F) F = 1, R += 2;
    }
    cout << R;
}
```

BOJ 4788. Double Dealing

i번째에 있는 카드가 한 번의 dealing 과정을 통해 이동하게 된 위치를 $f(i)$라고 합시다. 모든 $1 \leq i \leq N$에 대해 $f^x(i) = i$가 되는 가장 작은 양의 정수 x를 구하는 문제라고 생각할 수 있습니다.

i번 정점에서 $f(i)$번 정점으로 가는 방향 간선을 만든 그래프는 functional graph이고, $\{f(1), f(2), \cdots, f(n)\}$은 순열이기 때문에 그래프의 모든 컴포넌트는 사이클 형태입니다.

크기가 x인 사이클에 속한 정점은 x의 배수 번 이동할 때마다 자신의 자리로 돌아옵니다. 따라서 모든 사이클의 크기의 최소 공배수가 정답입니다.

$f(*)$는 $O(n+k)$ 시간에 처리할 수 있고, 각 사이클의 길이는 $O(n)$ 시간에 구할 수 있습니다. 유클리드 호제법을 이용해 최소 공배수를 구하면 매번 $O(\log X)$ 정도의 시간이 걸리므로 전체 시간 복잡도는 $O(k + n \log X)$입니다.

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

int N, K, C[888];

void Solve(){
    vector<vector<int>> V(K);
    for(int i=0; i<N; i++) V[i%K].push_back(i);
    for(int i=0; i<K; i++) reverse(V[i].begin(), V[i].end());
    vector<int> R;
    for(int i=0; i<K; i++) R.insert(R.end(), V[i].begin(), V[i].end());
    memset(C, 0, sizeof C);
    ll X = 1;
    for(int i=0; i<N; i++){
        if(C[i]) continue; ll cnt = 0;
```

```
        for(int j=i; !C[j]; j=R[j]) C[j] = 1, cnt++;
        X = lcm(X, cnt);
    }
    cout << X << "\n";
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    while(cin >> N >> K && N) Solve();
}
```

BOJ 27490. Moving Dots

점의 이동 방향이 중간에 바뀌지 않는다는 것을 먼저 관찰해야 합니다. 그림을 몇 번 그려보면 어렵지 않게 증명할 수 있습니다.

따라서 두 점 X_i와 Y_j가 만나서 하나의 점을 이룰 조건은 X_i와 X_j가 서로 가장 가까운 두 점이 되어서 두 점의 중점에서 만나는 것임을 알 수 있습니다.

더블 카운팅을 합시다. 즉, 모든 부분 집합에 대해 점의 개수를 구하는 것 대신, 각 점이 등장하는 부분 집합의 개수를 셀 것입니다.

u에 있는 점과 v에 있는 점($u < v$)이 만나게 되는 부분 집합의 수를 구합시다. 이는 $[2u - v, 2v - u)$ 구간에 포함되지 않는 점들의 집합의 부분 집합의 개수와 동일합니다. 따라서 구간에 포함되지 않은 점의 개수를 이분 탐색으로 구한 뒤, 2의 거듭제곱 꼴을 계산해서 정답에 더하면 됩니다.

더블 카운팅이라는 키워드만 떠올릴 수 있다면 어렵지 않게 풀 수 있는 문제입니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll MOD = 1e9+7;

ll N, A[3030], P[3030], R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    P[0] = 1; for(int i=1; i<3030; i++) P[i] = P[i-1] * 2 % MOD;
    cin >> N; for(int i=0; i<N; i++) cin >> A[i];
    for(int i=0; i<N; i++){
        for(int j=i+1; j<N; j++){
            int cnt = upper_bound(A, A+N, 2*A[j]-A[i]-1) - lower_bound(A, A+N,
2*A[i]-A[j]);
            R += P[N-cnt]; if(R >= MOD) R -= MOD;
        }
    }
    cout << R;
}
```

BOJ 2601. 도서실카펫

직사각형 $[x_1, x_2] \times [y_1, y_2]$가 정사각형 $[x - k, x] \times [y - k, y]$으로 가려질 조건은 다음과 같습니다.

- $x - k \le x_1 \le x_2 \le x$
- $y - k \le y_1 \le y_2 \le y$

이런 상황에서 문제를 풀 때는 최대한 변수를 적게 사용한 식으로 조건을 표현하는 것이 좋습니다. 그러므로 두 부등식을 각각 x, y의 구간 형태로 나타내어 봅시다.

- $x_2 \le x \le x_1 + k$
- $y_2 \le y \le y_1 + k$

카펫을 적당히 배치했을 때 가릴 수 있는 직사각형의 최대 개수를 구해야 합니다. 이 문제는 직사각형마다 2차원 배열의 $[x_2, x_1 + k] \times [y_2, y_1 + k]$ 구간에 1을 더한 다음, 배열의 최댓값을 구해서 해결할 수 있습니다. 하지만 2차원 쿼리는 귀찮기 때문에 플레인 스위핑을 사용할 것입니다.

x좌표를 기준으로 스위핑합시다. 구체적으로, x_2를 만나면 $[y_2, y_1 + k]$ 구간을 1 증가시키고, $x_1 + k$를 지나면 $[y_2, y_1 + k]$를 구간을 1 감소시킵니다. 구간에 어떤 값을 더하는 쿼리와 수열의 전체 원소의 최댓값을 구하는 쿼리를 처리해야 하고, 이는 세그먼트 트리로 처리할 수 있습니다.

각 x좌표에서의 이벤트를 모두 처리한 다음 세그먼트 트리의 루트 정점의 값을 참조하는 방식으로 구현하면 전체 문제를 $O(N \log N)$ 시간에 해결할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
constexpr int SZ = 1 << 21;

int T[SZ<<1], L[SZ<<1];

void Push(int node, int s, int e){
    T[node] += L[node];
    if(s != e) L[node<<1] += L[node], L[node<<1|1] += L[node];
    L[node] = 0;
}

void Update(int l, int r, int v, int node=1, int s=0, int e=SZ-1){
    Push(node, s, e);
    if(r < s || e < l) return;
    if(l <= s && e <= r){ L[node] += v; Push(node, s, e); return; }
    int m = (s + e) / 2;
    Update(l, r, v, node<<1, s, m);
    Update(l, r, v, node<<1|1, m+1, e);
    T[node] = max(T[node<<1], T[node<<1|1]);
}

int Query(int l, int r, int node=1, int s=0, int e=SZ-1){
    Push(node, s, e);
    if(r < s || e < l) return 0;
    if(l <= s && e <= r) return T[node];
    int m = (s + e) / 2;
    return max(Query(l, r, node<<1, s, m), Query(l, r, node<<1|1, m+1, e));
}

struct Event{
    int t, l, r, v;
    Event() = default;
    Event(int t, int l, int r, int v) : t(t), l(l), r(r), v(v) {}
    bool operator < (const Event &e) const { return t < e.t; }
};

int K, N, R, X1, Y1, X2, Y2;
vector<Event> E;
```

```
int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> X1 >> Y2 >> X2 >> Y1 >> K >> N; X1 += K; Y1 += K; E.reserve(N+N);
    for(int i=0; i<N; i++){
        int r2, c2, r1, c1; cin >> r2 >> c1 >> r1 >> c2;
        r2 += K; c2 += K;
        if(r1 <= r2 && c1 <= c2) E.emplace_back(r1, c1, c2, +1),
E.emplace_back(r2+1, c1, c2, -1);
    }
    sort(E.begin(), E.end());
    for(int i=0, j=0; i<E.size(); i=j){
        while(j < E.size() && E[i].t == E[j].t) Update(E[j].l, E[j].r, E[j].v),
j++;
        Push(1, 0, SZ-1); R = max(R, T[1]);
    }
    cout << R;
}
```

BOJ 17704. Matryoshka

N개의 점 (X_i, Y_i)가 주어지고 쿼리로 (A, B)가 주어지면, $X_i \geq A \wedge Y_i \leq B$인 점들만 고려했을 때 오른쪽 위로 strict하게 올라가는 minimum path cover의 크기를 구하는 문제입니다.

poset에서 minimum path cover의 크기는 maximum anti chain의 크기와 같습니다. 이 문제에서 anti chain은 $X_i \leq X_j \wedge Y_i \geq Y_j$를 만족하는 경로입니다. 이런 꼴의 경로 중 가능한 길이의 최댓값을 구하면 됩니다.

부등호 방향이 다르면 보기 싫으니까 y좌표를 뒤집어서 통일시킵니다. 쿼리로 (A, B)가 주어졌을 때 (A, B) 기준 1사분면 또는 축에 있는 점들 중 오른쪽 위로 monotone하게 올라가는 LIS를 구하는 문제가 됩니다.

오른쪽 위로 올라가는 경로 대신 왼쪽 아래로 내려가는 경로를 생각해 봅시다. 이런 형식의 DP는 세그먼트 트리를 들고 (y, x) 내림차순으로 스위핑하면서 계산할 수 있습니다.

쿼리는 $[A, \infty] \times [B, \infty]$ 구간에서의 2차원 최댓값 쿼리라고 생각할 수 있지만 2차원 쿼리를 처리하는 것은 귀찮습니다. 따라서 그냥 쿼리를 오프라인으로 입력받은 다음, DP 계산할 때 스위핑하면서 같이 정답을 구하는 것이 편합니다.

```
#include <bits/stdc++.h>
using namespace std;
constexpr int SZ = 1 << 19;

struct Event{
    int x, y, i;
    Event() = default;
    Event(int x, int y, int i) : x(x), y(y), i(i) {}
    bool operator < (const Event &e) const {
        return make_tuple(y, x, -i) > make_tuple(e.y, e.x, -e.i);
    }
};

int N, Q, R[SZ], T[SZ<<1];
vector<int> X, Y;
vector<Event> E;
```

```
void Update(int x, int v){
    for(x|=SZ; x; x>>=1) T[x] = max(T[x], v);
}

int Query(int l, int r){
    int res = 0;
    for(l|=SZ, r|=SZ; l<=r; l/=2, r/=2){
        if(l & 1) res = max(res, T[l++]);
        if(~r & 1) res = max(res, T[r--]);
    }
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> Q; E.reserve(N+Q); X.reserve(N+Q); Y.reserve(N+Q);
    for(int i=1,x,y; i<=N; i++) cin >> x >> y, E.emplace_back(x,-y,0);
    for(int i=1,x,y; i<=Q; i++) cin >> x >> y, E.emplace_back(x,-y,i);
    for(const auto &[x,y,i] : E) X.push_back(x), Y.push_back(y);
    sort(X.begin(), X.end()); X.erase(unique(X.begin(), X.end()), X.end());
    sort(Y.begin(), Y.end()); Y.erase(unique(Y.begin(), Y.end()), Y.end());
    for(auto &[x,y,i] : E){
        x = lower_bound(X.begin(), X.end(), x) - X.begin() + 1;
        y = lower_bound(Y.begin(), Y.end(), y) - Y.begin() + 1;
    }
    sort(E.begin(), E.end());
    for(const auto &[x,y,i] : E){
        if(i == 0) Update(x, Query(x, SZ-1) + 1);
        else R[i] = Query(x, SZ-1);
    }
    for(int i=1; i<=Q; i++) cout << R[i] << "\n";
}
```

BOJ 24261. Same Sum Subsequences

연속한 부분 수열만 고려해도 항상 정답을 찾을 수 있습니다. A의 누적 합 배열을 a, B의 누적 합 배열을 b라고 하겠습니다.

일반성을 잃지 않고, $a_n \le b_m$을 만족한다고 가정합시다. 각 a_i에 대해 $b_j \le a_i$를 만족하는 j를 찾으면, $0 \le a_i - b_j < n$을 만족한다는 것을 알 수 있습니다. $b_j \le a_i < b_{j+1}$이기 때문입니다.

$a_0 - b_0$까지 포함해서 $0 \le a_i - b_j < n$인 순서쌍 (i, j)를 $n+1$개 찾을 수 있습니다. 이때 $a_i - b_j$로 가능한 값의 개수는 n가지이므로 비둘기집 원리에 의해 $i' < i, j' < j, a_i - b_j = a_{i'} - b_{j'}$을 찾을 수 있습니다. 따라서 항상 연속한 부분 수열 중에 정답이 존재합니다.

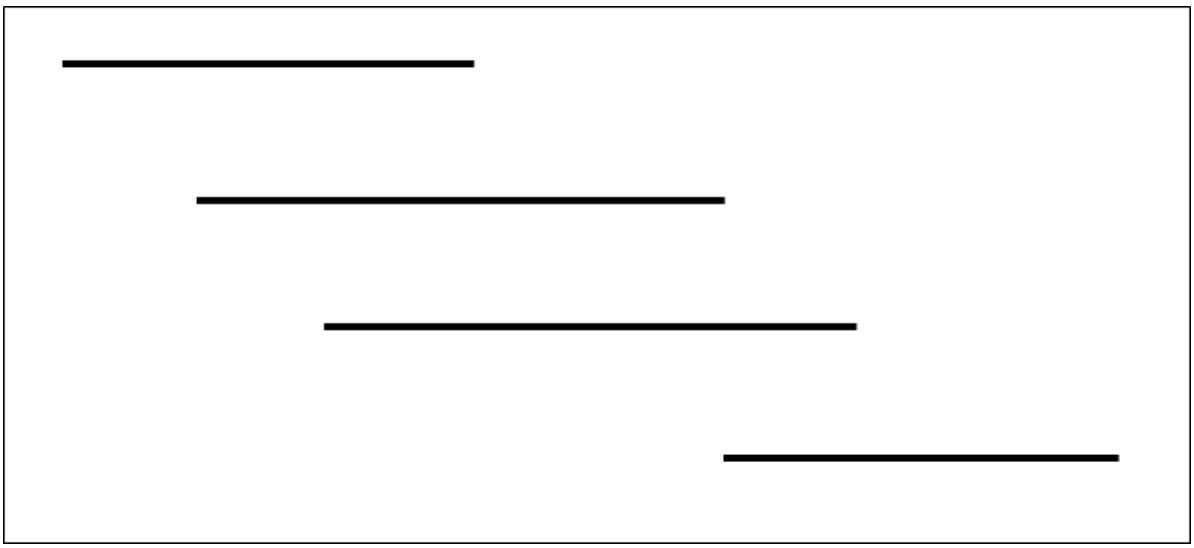
```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

ll N, M, A[1010101], B[1010101], F;
pair<int,int> Pos[1010101];

void Print(ll *arr, int s, int e){
    cout << e - s + 1 << "\n";
    for(int i=s; i<=e; i++) cout << i - 1 << " ";
    cout << "\n";
```

```
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    cin >> M;
    for(int i=1; i<=M; i++) cin >> B[i];
    ll SA = accumulate(A+1, A+N+1, 0LL);
    ll SB = accumulate(B+1, B+M+1, 0LL);
    F = SA > SB;
    if(F){
        for(int i=1; i<=max(N,M); i++) swap(A[i], B[i]);
        swap(N, M);
    }

    ll a = 0, b = 0;
    fill(Pos, Pos+1010101, make_pair(-1, 01));
    Pos[0] = {0, 0};
    for(int i=1, j=0; i<=N; i++){
        a += A[i];
        while(j + 1 <= M && b + B[j+1] <= a) b += B[++j];
        int diff = a - b;
        auto pos = Pos[diff];
        if(pos.first == -1){ Pos[diff] = {i, j}; continue; }
        ll s1 = pos.first + 1, e1 = i, s2 = pos.second + 1, e2 = j;
        if(F == 0) Print(A, s1, e1), Print(B, s2, e2);
        else Print(B, s2, e2), Print(A, s1, e1);
        break;
    }
}
```

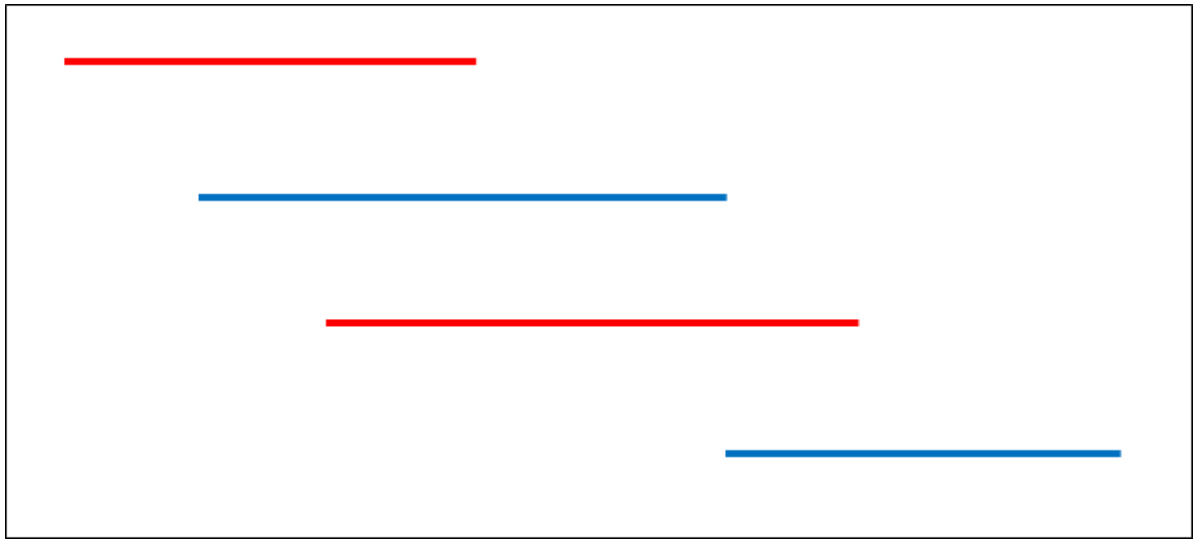
BOJ 15864. Alternating Current

어떤 선분 l_1이 l_2에 완전히 포함된다면, l_1은 l_2와 다른 색을 배정해주면 되기 때문에 l_1을 없애고 생각해도 됩니다. KOI 2014 버스노선처럼 Sliding Window를 이용해 다른 선분에 완전히 포함되는 선분을 제거하고, 어떤 선분에 포함되는지만 기록합시다. (`Elimination` 함수 참고)

각 구간은 최소 한 개 이상의 선분으로 덮여있기 때문에, 어떠한 선분에도 포함되지 않는 선분들만 남겨놓은 다음 정렬하면 아래 그림처럼 나오게 됩니다.



만약 정답이 **존재한다면** 아래 그림처럼 색을 교차해서 놓았을 때 항상 정답이 나온다는 것을 알 수 있습니다.



어떠한 선분에도 포함되지 않은 선분의 개수를 K라고 합시다.

K가 짝수라면 위 그림처럼 색을 교차하게 놓아서 확인하면 끝날텐데, 홀수라면 약간 까다롭습니다.

K = 5라면 아래 5가지 경우를 모두 확인해야 합니다.

```
01010 (1번 선분부터 시작)
11010 (2번 선분부터 시작)
10010 (3번 선분부터 시작)
10110 (4번 선분부터 시작)
10100 (5번 선분부터 시작)
```

K가지 경우를 Naive하게 확인해주면 서브태스크 1, 2, 3을 해결해 55점을 받을 수 있습니다.

위에서 작성한 5가지 경우를 잘 보면, 어떤 i번째 비트열과 i-1번째 비트열은 비트 1개 밖에 차이가 나지 않는다는 것을 알 수 있습니다. K가지 경우를 모두 확인할 때 각 선분의 색깔을 1번만 바꿔도 된다는 것을 알 수 있습니다.

Segment Tree + Lazy Propagation을 사용해 구간의 최솟값을 관리하면 선분이 잘 덮여있는지 빠르게 확인해줄 수 있습니다.

선분을 정렬하고 필요 없는 선분을 제거하는데 $O(M \log M)$만큼 걸리고, 정답이 존재하는지 확인하는 데 $O(M \log N)$이 걸리므로 문제를 해결할 수 있습니다.

```cpp
#include <bits/stdc++.h>
using namespace std;
constexpr int SZ = 1 << 17;
constexpr int INF = 0x3f3f3f3f;

struct union_find{
    vector<int> p;
    union_find(int n) : p(n+1) { iota(p.begin(), p.end(), 0); }
    int find(int v){ return v == p[v] ? v : p[v] = find(p[v]); }
    void merge(int pa, int ch){ pa = find(pa); ch = find(ch); p[ch] = pa; }
};

struct segment_tree{
```

```
    int T[SZ<<1], L[SZ<<1];
    segment_tree(){ clear(); }
    void clear(){ memset(T, 0, sizeof T); memset(L, 0, sizeof L); }
    void push(int node, int s, int e){
        T[node] += L[node];
        if(s != e) L[node<<1] += L[node], L[node<<1|1] += L[node];
        L[node] = 0;
    }
    void range_add(int l, int r, int v, int node=1, int s=0, int e=SZ-1){
        push(node, s, e);
        if(r < s || e < l) return;
        if(l <= s && e <= r){ L[node] += v; push(node, s, e); return; }
        int m = (s + e) / 2;
        range_add(l, r, v, node<<1, s, m);
        range_add(l, r, v, node<<1|1, m+1, e);
        T[node] = min(T[node<<1], T[node<<1|1]);
    }
    int range_min(int l, int r, int node=1, int s=0, int e=SZ-1){
        push(node, s, e);
        if(r < s || e < l) return INF;
        if(l <= s && e <= r) return T[node];
        int m = (s + e) / 2;
        return min(range_min(l, r, node<<1, s, m), range_min(l, r, node<<1|1,
m+1, e));
    }
    void circular_add(int n, int l, int r, int v){
        if(r >= n) r -= n;
        if(l <= r) range_add(l, r, v);
        else range_add(0, r, v), range_add(l, n-1, v);
    }
};

struct segment_t{
    int s, e, idx, par, color;
    segment_t() : segment_t(0, 0, 0) {}
    segment_t(int s, int e, int idx) : s(s), e(e), idx(idx), par(-1), color(-1)
{}
    bool operator < (const segment_t &t) const { return s != t.s ? s < t.s : e >
t.e; }
};

int N, M;
vector<int> G[101010];
segment_t A[101010];
vector<int> V;

void Elimination(){
    int L = 0, P = -1;
    for(int i=1; i<=M; i++){
        if(A[i].s <= A[i].e) continue;
        if(L < A[i].e) L = A[i].e, P = i;
        A[i].e += N;
    }
    sort(A+1, A+M+1);

    deque<segment_t> Q;
    union_find U(M);
    for(int i=1; i<=M; i++){
```

```
        if(Q.empty() || Q.back().e < A[i].e) Q.push_back(A[i]);
        else U.merge(Q.back().idx, A[i].idx);
    }
    while(!Q.empty() && Q.front().e <= L) U.merge(P, Q.front().idx),
Q.pop_front();

    sort(A+1, A+M+1, [](auto a, auto b){ return a.idx < b.idx; });
    for(int i=1; i<=M; i++) A[i].par = U.find(i);
    for(int i=1; i<=N; i++) G[A[i].par].push_back(i);

    for(const auto &i : Q) V.push_back(i.idx);
}

segment_tree T[2];

void Print(){
    for(int i=1; i<=M; i++) cout << A[i].color;
}

void Even(){
    T[0].clear(); T[1].clear();
    for(int i=0; i<V.size(); i++) A[V[i]].color = i % 2;
    for(int i=1; i<=M; i++) if(A[i].color == -1) A[i].color =
!A[A[i].par].color;

    for(int i=1; i<=M; i++) T[A[i].color].circular_add(N, A[i].s, A[i].e, 1);
    if(T[0].range_min(0, N-1) > 0 && T[1].range_min(0, N-1) > 0){ Print();
return; }
    else cout << "impossible";
}

void Odd(){
    T[0].clear(); T[1].clear();
    for(int i=0; i<V.size(); i++) A[V[i]].color = i % 2;
    for(int i=1; i<=M; i++) if(A[i].color == -1) A[i].color =
!A[A[i].par].color;

    for(int i=1; i<=M; i++) T[A[i].color].circular_add(N, A[i].s, A[i].e, 1);
    if(T[0].range_min(0, N-1) > 0 && T[1].range_min(0, N-1) > 0){ Print();
return; }

    for(int i=0; i<V.size(); i++){
        for(auto j : G[V[i]]){
            T[A[j].color].circular_add(N, A[j].s, A[j].e, -1);
            A[j].color ^= 1;
            T[A[j].color].circular_add(N, A[j].s, A[j].e, +1);
        }
        if(T[0].range_min(0, N-1) > 0 && T[1].range_min(0, N-1) > 0){ Print();
return; }
    }
    cout << "impossible";
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1; i<=M; i++) cin >> A[i].s >> A[i].e, A[i].idx = i;
    Elimination();
```

```
    if(v.size() % 2 == 0) Even();
    else Odd();
}
```