2024 Taiwan Online Programming Contest

> 난이도: AJ K BD IE CF LGH

A. Animal Farm

가장 영향력이 높은 돼지의 영향력 X와, 영향력이 X 미만인 모든 동물의 영향력을 더한 값을 출력하면 됩니다.

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

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int N; cin >> N;
    vector<pair<string,int>> V(N);
    for(auto &[a,b] : V) cin >> a >> b;
    int mx = 0;
    for(auto [a,b] : V) if(a == "pig") mx = max(mx, b);
    long long res = mx;
    for(auto [a,b] : V) if(a != "pig" && b < mx) res += b;
    cout << res;
}
```

B. Business Magic

$D(i,j)$를 다음과 같이 정의합시다.

- $D(i,0) := 1,2,\cdots,i$번 상점만 고려했을 때, blue magic을 사용하지 않고 얻을 수 있는 최댓값
- $D(i,1) := 1,2,\cdots,i$번 상점만 고려했을 때, i번 상점에 blue magic을 사용해서 얻을 수 있는 최댓값
- $D(i,2) := 1,2,\cdots,i$번 상점만 고려했을 때, blue magic을 i보다 앞에 있는 상점들에 사용해서 얻을 수 있는 최댓값

정답은 $\max\{D(N,0), D(N,1), D(N,2)\}$이고, $O(N)$ 시간에 구할 수 있습니다.

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

ll N, A[303030], D[303030][3];

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    memset(D, 0xc0, sizeof D); D[0][0] = 0;
    for(int i=1; i<=N; i++){
        D[i][0] = D[i-1][0] + abs(A[i]);
        D[i][1] = max(D[i-1][0], D[i-1][1]) + A[i] * 2;
        D[i][2] = max(D[i-1][1], D[i-1][2]) + abs(A[i]);
```

```
    }
    cout << max({D[N][0], D[N][1], D[N][2]});
}
```

이 밖에도 최대 연속 합을 이용한 풀이도 있습니다. blue magic을 사용하지 않았을 때의 답은 $\sum |r_i|$입니다. 일단 $r_i < 0$인 모든 상점에 green magic을 적용해서 $\sum |r_i|$를 만든 다음, 어떤 구간에 green magic을 적용한 것을 취소한 뒤 blue magic을 적용하는 것을 생각해 봅시다.

만약 $r_i \geq 0$이었다면 blue magic을 적용했을 때 가중치가 $r_i \to 2r_i$가 되므로 blue magic을 통해 얻는 이득은 r_i입니다. 반면, $r_i < 0$이었다면 blue magic을 적용했을 때 가중치가 $-r_i \to 2r_i$가 되므로 blue magic을 통해 얻는 이득은 $3r_i$입니다. 따라서 다음과 같은 수열 a를 정의하면, 정답은 a의 구간 합으로 가능한 최댓값에 $\sum |r_i|$를 더한 것과 같음을 알 수 있습니다.

$$a_i = \begin{cases} r_i & \text{if } r_i \geq 0 \\ 3r_i & \text{if } r_i < 0 \end{cases}$$

이 풀이를 사용해도 $O(N)$ 시간에 문제를 해결할 수 있습니다.

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

ll N, A[303030], S;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=1; i<=N; i++) cin >> A[i];
    for(int i=1; i<=N; i++) S += abs(A[i]);
    for(int i=1; i<=N; i++) A[i] = A[i] >= 0 ? A[i] : 3 * A[i];
    ll now = 0, mx = 0;
    for(int i=1; i<=N; i++) mx = max(mx, now=max(0LL,now)+A[i]);
    cout << S + mx;
}
```

C. Cards

순열에서 임의의 두 원소를 교환하면 항상 inversion 개수를 2로 나눈 나머지가 바뀝니다. 따라서 두 순열 a와 b의 inversion의 홀짝성이 다르다면 불가능합니다. 홀짝성이 같다면 항상 가능함을 보일 수 있습니다.

일단 카드를 a_i 오름차순으로 정렬한 뒤, b의 inversion의 개수를 X라고 합시다. 순열의 inversion 개수와 버블 정렬의 swap 횟수가 같다는 것을 생각해 보면, $b_{i-1} > b_i$인 두 카드를 교환하면 a의 inversion은 1 증가하고 b의 inversion은 1 감소한다는 것을 알 수 있습니다. 따라서 인접한 두 원소를 교환하는 것을 $X/2$번 반복하면 a, b의 inversion이 같아집니다. 하지만 이 과정을 직접 구현하면 $X \leq N(N-1)/2$이기 때문에 시간 복잡도가 $O(N^2)$이 되어 문제를 해결할 수 없습니다.

버블 정렬의 K번째 루프까지 돌린 결과를 $O(N \log N)$ 시간에 구하는 방법(BOJ 11920)은 잘 알려져 있으며, 파라메트릭 서지를 이용하면 swap 횟수가 $X/2$ 이하인 최대 루프 횟수를 구할 수 있습니다. 그 다음에는 $O(N)$짜리 루프를 한 번 돌려서 swap 횟수를 정확히 $X/2$로 맞추면 되므로, $O(N \log^2 N)$에 답을 구할 수 있습니다.

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

```cpp
int f(int x){ return x; }
int f(pair<int,int> x){ return x.first; }

template<typename Type>
ll InversionCount(const vector<Type> &v){
    vector<int> T(v.size()+3, 0); ll res = 0;
    for(int i=v.size()-1; i>=0; i--){
        for(int j=f(v[i]); j; j-=j&-j) res += T[j];
        for(int j=f(v[i]); j<T.size(); j+=j&-j) T[j]++;
    }
    return res;
}

// https://www.acmicpc.net/problem/11920
template<typename Type>
vector<Type> AfterLoop(const vector<Type> &v, int k){
    if(k == 0) return v;
    vector<Type> res;
    priority_queue<Type, vector<Type>, greater<>> pq;
    for(int i=0; i<v.size(); i++){
        pq.push(v[i]);
        if(pq.size() > k) res.push_back(pq.top()), pq.pop();
    }
    while(!pq.empty()) res.push_back(pq.top()), pq.pop();
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    int N; cin >> N;
    vector<int> A(N), B(N);
    for(auto &i : A) cin >> i;
    for(auto &i : B) cin >> i;
    if(InversionCount(A) % 2 != InversionCount(B) % 2){ cout << "No"; return 0;
}

    vector<pair<int,int>> V(N);
    for(int i=0; i<N; i++) V[i] = {A[i], B[i]};
    sort(V.begin(), V.end(), [](auto a, auto b){ return a.second < b.second; });

    ll total = InversionCount(V);
    int l = 0, r = N;
    while(l < r){
        int m = (l + r + 1) / 2;
        if(total - InversionCount(AfterLoop(V, m)) <= total/2) l = m;
        else r = m - 1;
    }

    V = AfterLoop(V, l);
    ll need = InversionCount(V) - total/2;
    for(int i=1; i<N; i++){
        if(need > 0 && V[i-1] > V[i]) swap(V[i-1], V[i]), need--;
    }

    cout << "Yes\n";
    for(auto [a,b] : V) cout << a << " "; cout << "\n";
    for(auto [a,b] : V) cout << b << " "; cout << "\n";
}
```

D. Disbursement on Quarantine Policy

기댓값의 선형성에 의해 모든 사람의 격리 기간의 합의 기댓값은 각 사람의 격리 기간의 기댓값의 합과 같습니다. 즉, (i, j)에 위치한 사람의 격리 기간을 $R_{i,j}$라고 하면, $\mathbb{E}[\sum_i \sum_j R_{i,j}] = \sum_i \sum_j \mathbb{E}[R_{i,j}]$입니다.

(i, j)에 위치한 사람이 감염되었을 확률을 $A_{i,j}$라고 할 때, $d_0, d_1, d_2, 0$일 동안 격리되어야 할 확률을 각각 구해봅시다.

1. (i, j)에 위치한 사람 자기 자신이 감염되었을 확률: $p_0 = A_{i,j}$
2. 상하좌우로 인접한 사람이 감염되었을 확률:
   $p_1 = 1 - (1 - A_{i-1,j})(1 - A_{i+1,j})(1 - A_{i,j-1})(1 - A_{i,j+1})$
3. 대각선 방향으로 인접한 사람이 감염되었을 확률:
   $p_2 = 1 - (1 - A_{i-1,j-1})(1 - A_{i-1,j+1})(1 - A_{i+1,j-1})(1 - A_{i+1,j+1})$

d_0일 격리할 확률은 p_0, d_1일 격리할 확률은 $(1 - p_0)p_1$, d_2일 격리할 확률은 $(1 - p_0)(1 - p_1)p_2$이고, 따라서 $\mathbb{E}[R_{i,j}] = p_0 d_0 + (1 - p_0)p_1 d_1 + (1 - p_0)(1 - p_1)p_2 d_2$ 입니다.

2의 모듈러 역원을 미리 계산해 두면 $O(NM)$ 시간에 문제를 해결할 수 있습니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll MOD = 1e9+7;
constexpr ll INV = (MOD + 1) / 2;
constexpr int di[] = {1, -1, 0, 0, 1, 1, -1, -1};
constexpr int dj[] = {0, 0, 1, -1, 1, -1, 1, -1};

struct ModInt{
    ll v;
    ModInt() : ModInt(0) {}
    ModInt(ll v) : v(v) { norm(); }
    void norm(){ v = (v % MOD + MOD) % MOD; }
    friend istream& operator >> (istream &in, ModInt &t){ in >> t.v; t.norm();
return in; }
    friend ostream& operator << (ostream &out, ModInt &t){ return out << t.v; }
    ModInt& operator += (const ModInt &t) { if((v+=t.v) >= MOD) v -= MOD; return
*this; }
    ModInt& operator -= (const ModInt &t) { if((v-=t.v) < 0) v += MOD; return
*this; }
    ModInt& operator *= (const ModInt &t) { v = v * t.v % MOD; return *this; }
    ModInt operator + (const ModInt &t) const { return ModInt(*this) += t; }
    ModInt operator - (const ModInt &t) const { return ModInt(*this) -= t; }
    ModInt operator * (const ModInt &t) const { return ModInt(*this) *= t; }
    ModInt operator - () const { return -v; }
    friend ModInt operator + (ll a, const ModInt &b){ return ModInt(a) + b; }
    friend ModInt operator - (ll a, const ModInt &b){ return ModInt(a) - b; }
    friend ModInt operator * (ll a, const ModInt &b){ return ModInt(a) * b; }
};

ll N, M;
ModInt D[3], A[111][111], P[111][111][3], R;
bool Bound(int i, int j){ return 1 <= i && i <= N && 1 <= j && j <= M; }
ll Input(){
    char c; cin >> c;
    if(c == 'V') return 1;
```

```
    if(c == '.') return 0;
    return INV;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> D[0] >> D[1] >> D[2];
    for(int i=1; i<=N; i++) for(int j=1; j<=M; j++) A[i][j] = Input();
    for(int i=1; i<=N; i++){
        for(int j=1; j<=M; j++){
            P[i][j][0] = A[i][j];
            P[i][j][1] = P[i][j][2] = 1;
            for(int k=0; k<8; k++){
                int r = i + di[k], c = j + dj[k];
                if(Bound(r, c)) P[i][j][1+(k>=4)] *= 1 - A[r][c];
            }
            P[i][j][1] = 1 - P[i][j][1];
            P[i][j][2] = 1 - P[i][j][2];
        }
    }
    for(int i=1; i<=N; i++){
        for(int j=1; j<=M; j++){
            ModInt ban = 1;
            for(int k=0; k<3; k++){
                R += ban * P[i][j][k] * D[k];
                ban *= 1 - P[i][j][k];
            }
        }
    }
    cout << R;
}
```

E. Effcient Slabstones Rearrangement

새로운 막대를 인접한 위치에 있는 두 막대 사이 등에 배치한다고 하면, 배치할 수 있는 구역은 총 $N+1$개 있습니다. 각 구간에서의 최적의 위치를 $O(N)$ 또는 $O(N \log N)$ 시간에 찾을 수 있다면 문제를 $O(N^2)$ 또는 $O(N^2 \log N)$에 해결할 수 있습니다.

막대를 어떤 위치에 배치하면, 새로 배치한 막대의 왼쪽에 있던 막대는 더 왼쪽으로, 오른쪽에 있던 막대는 더 오른쪽으로 이동해야 합니다. 이때, 한 구간 안에서는 왼쪽의 이동량과 오른쪽의 이동량이 최대한 비슷하도록 배치하는 것이 최적입니다. 만약 둘의 차이가 크다면 막대 전체를 이동량이 더 적은 방향으로 아주 조금 옮겨서 총 이동 거리를 줄일 수 있기 때문입니다. 여기에서 조금 더 생각하면, 한 구간 안에서는 새로 배치하는 막대의 위치에 대한 기존 막대들의 이동 거리 함수가 unimodal하다는 것도 관찰할 수 있습니다.

따라서 각 구간에서 삼분 탐색을 이용해 $O(N \log N)$ 시간에 최적의 위치를 찾는 것을 $N+1$개의 구간에 대해 반복하면, 전체 문제를 $O(N^2 \log N)$ 시간에 해결할 수 있습니다.

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

ll N, M, D, x, L[2020], R[2020], LE[2020], RI[2020];
ll Len(int i){ return R[i] - L[i] + 1; }

ll f(int idx, ll pos){
```

```
    ll res = 0, le = pos, ri = pos+x-1;
    for(int i=idx; i>=1; i--){
        ll nxt = min(L[i], le - D - Len(i));
        res += L[i] - nxt; le = nxt;
    }
    for(int i=idx+1; i<=N; i++){
        ll nxt = max(R[i], ri + D + Len(i));
        res += nxt - R[i]; ri = nxt;
    }
    return res;
}

ll Solve(int idx){
    ll l = idx > 0 ? LE[idx]+D+1 : 1;
    ll r = idx < N ? RI[idx+1]-D-x : M-x+1;
    if(l > r) return 1e18;

    ll res = 1e18;
    while(l + 3 < r){
        ll m1 = (l + l + r) / 3, m2 = (l + r + r) / 3;
        if(f(idx, m1) > f(idx, m2)) l = m1; else r = m2;
    }
    for(ll i=l; i<=r; i++) res = min(res, f(idx, i));
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> D >> x;
    for(int i=1; i<=N; i++) cin >> L[i] >> R[i];

    LE[1] = Len(1);
    for(int i=2; i<=N; i++) LE[i] = LE[i-1] + D + Len(i);
    RI[N] = M - Len(N) + 1;
    for(int i=N-1; i>=1; i--) RI[i] = RI[i+1] - D - Len(i);

    ll res = 1e18;
    for(int i=0; i<=N; i++) res = min(res, Solve(i));
    cout << (res < 1e18 ? res : -1);
}
```

F. Fibonacci Lucky Numbers

피보나치 수열의 7^{7^n} 번째 항을 10^{10}으로 나눈 나머지를 구해야 합니다. 피보나치 수열을 10^n으로 나눈 나머지는 $C = 1.5 \times 10^n$을 주기로 순환한다는 사실이 알려져 있습니다. 모르더라도 Floyd's Cycle Detection Algorithm을 사용하면 순환 주기를 적당히 빠르게 찾을 수 있습니다.

아무튼, 문제를 피보나치 수열의 $7^{7^n} \mod C$번째 항을 찾는 문제로 바꿨습니다. 피보나치 수열의 N번째 항을 찾는 건 행렬을 이용해 $O(\log N)$에 할 수 있다는 것을 알고 있을 것이라고 믿고, $7^{7^n} \mod C$를 잘 계산하는 방법에 대해 이야기해 보겠습니다.

7을 아무리 많이 곱하더라도 $C = 2^{10} \times 3 \times 5^{11}$과는 서로소입니다. 따라서 오일러 정리에 의해 임의의 양의 정수 x에 대해 $(7^x)^{\phi(C)} \equiv 1 \pmod C$입니다. 따라서 $7^{7^n} \equiv 7^{(7^n \mod \phi(C))} \equiv 7^{(7^{(7^n \mod \phi(\phi(C)))} \mod \phi(C))} \pmod C$입니다. 이를 이용하면 구하고자 하는 항의 위치를 빠르게 알 수 있습니다.

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll MOD = 10'000'000'000;
constexpr ll PERIOD = MOD * 1.5;
ll MulMod(ll a, ll b, ll c){ return (__int128_t)a * b % c; }
ll PowMod(ll a, ll b, ll c){
    ll res = 1;
    for(a%=c; b; b>>=1, a=MulMod(a, a, c)) if(b & 1) res = MulMod(res, a, c);
    return res;
}

namespace fibonacci{
    struct matrix{
        int n; vector<vector<ll>> a;
        matrix(int n, int id=0) : n(n), a(n, vector<ll>(n)) {
            for(int i=0; i<n; i++) a[i][i] = id;
        }
        vector<ll>& operator [] (int i) { return a[i]; }
        const vector<ll>& operator [] (int i) const { return a[i]; }
        matrix operator * (const matrix &b) const {
            matrix c(n);
            for(int i=0; i<n; i++) for(int j=0; j<n; j++) for(int k=0; k<n; k++)
                        c[i][j] = (c[i][j] + MulMod(a[i][k], b[k][j], MOD)) %
MOD;
            return c;
        }
    };
    matrix pow(matrix a, ll b){
        matrix res(a.n, 1);
        for(; b; b>>=1, a=a*a) if(b & 1) res = res * a;
        return res;
    }
    ll run(ll n){
        matrix M(2);
        M[0][0] = 1; M[0][1] = 1;
        M[1][0] = 1; M[1][1] = 0;
        M = pow(M, n-1);
        return M[0][0];
    }
}

ll Phi(ll x){
    ll res = x;
    for(ll p=2; p*p<=x; p++){
        if(x % p) continue;
        res = res / p * (p - 1);
        while(x % p == 0) x /= p;
    }
    if(x != 1) res = res / x * (x - 1);
    return res;
}

ll PHI[3];
void Solve(){
    ll n; cin >> n;
    for(int i=2; i>=0; i--) n = PowMod(7, n, PHI[i]);
    cout << setw(10) << setfill('0') << fibonacci::run(n) << "\n";
```

```
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    PHI[0] = PERIOD;
    PHI[1] = Phi(PHI[0]);
    PHI[2] = Phi(PHI[1]);
    int TC; cin >> TC;
    for(int tc=1; tc<=TC; tc++) Solve();
}
```

G. Game of Rounding

문제는 구간의 시작점 l이 고정되어 있을 때 (1) 평균값으로 가능한 최댓값을 구하는 것, (2) 그 최댓값을 달성할 수 있는 구간의 최소 길이를 구하는 것까지 총 두 단계로 구성되어 있습니다.

누적 합 배열 S를 생각해 보면, 첫 번째 단계는 $\frac{S_r-S_l}{r-l}+0.5$의 최댓값을 찾는 문제가 됩니다. 결정 문제로 바꾸면 $\frac{S_r-S_l}{r-l} \geq x-0.5$를 판별하는 문제가 되고, 식을 정리하면 $S_r-(x-0.5)r \geq S_l-(x-0.5)l$인지 판별하는 문제라고 생각할 수 있습니다. 양변에 2를 곱하면 $2S_r-(2x-1)r \geq 2S_l-(2x-1)l$이 되어 정수만 이용해 계산할 수 있습니다.

여기에서 다시 양변을 x에 대한 일차 함수로 나타내어 보면 $-2rx+(2S_r+r) \geq -2lx+(2S_l+l)$이 되어 Convex Hull Trick을 적용할 수 있다는 생각을 할 수 있습니다.

l이 고정되어 있을 때 x가 주어지면 우변은 상수 시간에 계산할 수 있고, 좌변은 Convex Hull Trick을 이용해 $O(\log N)$ 시간에 계산할 수 있습니다. 따라서 각각의 l마다 가능한 x의 최댓값, 즉 $\frac{S_r-S_l}{r-l}$의 최댓값은 $O(N \log^2 N)$(Li Chao Tree를 사용하면 $O(N \log N \log A)$)에 계산할 수 있습니다.

구간의 시작점 l에서 만들 수 있는 최댓값 x_l을 알고 있을 때, 실제로 이 최댓값을 달성하기 위한 구간 끝 점의 최솟값 R_l은 세그먼트 트리의 각 정점에 CHT를 저장한 뒤, 세그먼트 트리 위에서 이분 탐색을 하면 $O(N \log^2 N)$ 또는 $O(N \log N \log A)$에 모두 구할 수 있습니다.

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

struct Line {
    mutable ll k, m, p;
    bool operator<(const Line& o) const { return k < o.k; }
    bool operator<(ll x) const { return p < x; }
};
struct LineContainer : multiset<Line, less<>> {
    static const ll inf = LLONG_MAX;
    ll div(ll a, ll b) { return a / b - ((a ^ b) < 0 && a % b); } // floor
    bool isect(iterator x, iterator y) {
        if (y == end()) return x->p = inf, 0;
        if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
        else x->p = div(y->m - x->m, x->k - y->k);
        return x->p >= y->p;
    }
    void add(ll k, ll m) {
        auto z = insert({k, m, 0}), y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
        while ((y = x) != begin() && (--x)->p >= y->p) isect(x, erase(y));
    }
```

```
    ll query(ll x) {
        if(empty()) return -1e18;
        auto l = *lower_bound(x);
        return l.k * x + l.m;
    }
};

LineContainer T[1<<19]; int sz;
void Init(int n){
    sz = 1;
    while(sz <= n) sz *= 2;
    for(int i=1; i<sz*2; i++) T[i].clear();
}
void Insert(int x, ll k, ll m){
    for(x|=sz; x; x>>=1) T[x].add(k, m);
}
int Find(ll x, ll mn){ // T[node].find(x) >= mn
    int node = 1;
    while(node < sz) node = node << 1 | (T[node<<1].query(x) < mn);
    return node ^ sz;
}

ll N, A[202020], S[202020], X[202020], R[202020];

void Solve(){
    cin >> N; Init(N);
    for(int i=1; i<=N; i++) cin >> A[i];
    partial_sum(A+1, A+N+1, S+1);

    LineContainer H;
    auto get_a = [](int i){ return -2*i; };
    auto get_b = [](int i){ return 2*S[i] + i; };
    auto get_limit = [&](int i, ll x){ return get_a(i-1)*x + get_b(i-1); };
    for(int i=N; i>=1; i--){
        H.add(get_a(i), get_b(i));
        ll l = 0, r = 2e9;
        while(l < r){
            ll m = (l + r + 1) / 2;
            if(H.query(m) >= get_limit(i, m)) l = m;
            else r = m - 1;
        }
        X[i] = l;
    }

    for(int i=N; i>=1; i--){
        Insert(i, get_a(i), get_b(i));
        R[i] = Find(X[i], get_limit(i, X[i]));
    }
    for(int i=1; i<=N; i++) cout << R[i] - i + 1 << " \n"[i==N];
}

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

H. Harmonious Passage of Magicians

점프할 수 있는 경우에는 무조건 점프해야 합니다. 점프하지 않고 한 칸 전진시키는 경우, 예를 들어 `LR.R` 일 때 `.RLR`로 만드는 대신 `LRR.`로 만든다면 앞에 있는 `L`은 영원히 이동하지 못합니다. 따라서 점프할 수 있을 때는 항상 점프해야 하고, 점프 가능한 칸이 2개 이상 있을 수 없으므로 tie breaker는 필요 없습니다.

점프를 할 수 없다면 어떤 원소를 한 칸 전진시켜야 합니다. 만약 `L`과 `R` 중 하나만 전진시킬 수 있다면 그것을 전진시키면 됩니다. 따라서 `L`과 `R` 모두 전진 가능한 상황, 즉 `L.R`의 형태가 나타나는 상황을 생각해 봅시다. 사전 순으로 최소인 결과를 만들고 싶으므로 웬만하면 `L`을 전진시키는 것이 좋을 것입니다. 따라서 `L`을 옮기면 안 될 조건을 생각하는 것이 좋아 보입니다.

둘 모두 전진시킬 수 있는 상황에서 항상 `L`을 이동할 때 발생할 수 있는 상황을 `LL.RL`, `LL.RR`, `RL.RL`, `RL.RR`로 나눠서 생각해 봅시다.

1. `LL.RL` - `L.LRL` - `LRL.L` - `LR.LL` - `.RLLL` - `R.LLL`
2. `LL.RR` - `L.LRR` - `LRL.R` - 3/4번 케이스로 이동
3. `RL.RL` - `R.LRL` - `RRL.L` - `R`들은 더 이상 이동할 수 없음
4. `RL.RR` - `R.LRR` - `RR.LR` - `R`들은 더 이상 이동할 수 없음

위에서 보면 알 수 있듯, 현재 상태가 `RL.R`를 포함하고 있다면 `RL.R`이 현재 상태의 prefix가 아닌 이상 `L`을 이동하면 안 됩니다. 그렇지 않은 경우에는 `L`을 이동하더라도 막히는 것 없이 잘 이동할 수 있습니다.

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

void Solve(){
    int N, M; cin >> N >> M;
    vector<int> V; V.reserve(N+M+1);
    for(int i=1; i<=N; i++) V.push_back(i);
    V.push_back(0);
    for(int i=1; i<=M; i++) V.push_back(N+i);

    auto L = [&](int x){ return 1 <= x && x <= N; };
    auto R = [&](int x){ return N+1 <= x && x <= N+M; };

    int pos = N;
    vector<int> res;

    while(true){
        assert(V[pos] == 0);

        // 1. 점프할 수 있으면 무조건 해야 함 / 둘 다 점프 가능한 경우는 발생하지 않음
        if(pos-2 >= 0 && L(V[pos-2]) && R(V[pos-1])){
            res.push_back(V[pos-2]);
            swap(V[pos-2], V[pos]); pos -= 2;
            continue;
        }
        if(pos+2 < V.size() && L(V[pos+1]) && R(V[pos+2])){
            res.push_back(V[pos+2]);
            swap(V[pos], V[pos+2]); pos += 2;
            continue;
        }

        bool le = pos-1 >= 0 && L(V[pos-1]);
```

```
        bool ri = pos+1 < V.size() && R(V[pos+1]);

        // 2. 둘 다 전진 가능하면 -> L.R 상태
        if(le && ri){
            // 2-1. ?RL.R 인 상태에서 L 전진하면 안 됨 (?RL.R -> ?R.LR -> ?RRL. 에서 R 이동 불가능)
            if(pos-3 >= 0 && R(V[pos-2])) le = false;
            else ri = false;
        }

        // 3. 전진 가능한 거 이동
        if(le){
            res.push_back(V[pos-1]);
            swap(V[pos-1], V[pos]); pos--;
        }
        else if(ri){
            res.push_back(V[pos+1]);
            swap(V[pos], V[pos+1]); pos++;
        }
        else break;
    }
    for(int i=1; i<=M; i++) assert(V[i-1] == N + i);
    for(int i=1; i<=N; i++) assert(V[M+i] == i);
    for(auto i : res) cout << i << " ";
    cout << "\n";
}

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

I. In Search of the Lost Array

$D(lst, bit) :=$ 집합 B에서 사용한 원소의 번호가 bit이고, 지금까지 재구성한 수열 A의 마지막 원소가 lst 일 때 남은 원소를 모두 복원할 수 있는가? 를 나타내는 점화식을 정의합시다. A_1의 값이 고정되어 있다면 Bit DP를 이용해 $O(N \times 2^N)$ 시간에 점화식을 계산한 뒤 실제 수열 A를 역추적할 수 있습니다.
$1 \le A_1 \le 100$이므로 $O(100N2^N)$ 시간에 전체 문제를 해결할 수 있습니다.

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

int N, B[17], D[111][1<<17], P[111][1<<17];

int f(int lst, int bit){
    if(lst < 1 || lst > 100) return 0;
    if(bit == 0) return 1;
    int &res = D[lst][bit];
    if(res != -1) return res;
    for(int i=0; i+1<N; i++){
        if(~bit >> i & 1) continue;
        if(B[i] % lst) continue;
        if(f(B[i]/lst, bit ^ (1 << i))){
            P[lst][bit] = i;
            return res = 1;
```

```
        }
    }
    return res = 0;
}

bool Test(int st){
    memset(D, -1, sizeof D);
    if(!f(st, (1<<(N-1))-1)) return false;

    cout << "Yes\n";
    int lst = st, bit = (1<<(N-1))-1;
    while(bit != 0){
        cout << lst << " ";
        int nxt = P[lst][bit];
        lst = B[nxt] / lst;
        bit ^= 1 << nxt;
    }
    cout << lst << "\n";
    return true;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N;
    for(int i=0; i+1<N; i++) cin >> B[i];
    if(N == 1){ cout << "Yes\n" << 1; return 0; }
    for(int i=1; i<=100; i++) if(Test(i)) return 0;
    cout << "No";
}
```

J. Just Round Down

다양한 방법이 있지만, 가장 간단한 방법은 double로 입력 받은 다음 int로 형 변환하여 출력하는 것, 또는 문자열로 입력 받은 다음 소수점이 나오기 전까지 출력하는 것이라고 생각합니다.

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

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    double x; cin >> x;
    cout << int(x);
}
```

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

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    string s; cin >> s;
    cout << s.substr(0, s.find('.'));
}
```

K. Kingdom's Development Plan

in-degree가 0인 정점들을 우선순위 큐로 관리하면서 위상 정렬을 수행하면 됩니다.

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

int N, M, C[101010];
vector<int> G[101010], R;

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M;
    for(int i=1,u,v; i<=M; i++) cin >> u >> v, G[u].push_back(v), C[v]++;
    priority_queue<int, vector<int>, greater<>> Q;
    for(int i=1; i<=N; i++) if(C[i] == 0) Q.push(i);
    while(!Q.empty()){
        int v = Q.top(); Q.pop();
        R.push_back(v);
        for(auto i : G[v]) if(!--C[i]) Q.push(i);
    }
    if(R.size() != N) cout << "IMPOSSIBLE";
    else for(auto i : R) cout << i << " ";
}
```

L. Lexicopolis

간선을 정확히 K개 이용해서 S에서 T로 가는 방법이 있는지 확인하는 것은 인접 행렬을 K제곱하는 것으로 확인할 수 있습니다. 이 문제도 행렬 제곱과 비슷한 방식으로 접근해 봅시다. 구체적으로, 간선을 a개 사용해서 i에서 k로 가는 사전 순 최소 경로 $path[a][i][k]$, 간선을 b개 사용해서 k에서 j로 가는 사전 순 최소 경로 $path[b][k][j]$를 알고 있을 때, $path[a+b][i][j]$를 구하는 방법을 찾을 것입니다.

실제로 경로를 구성하는 간선을 모두 저장하기에는 시간과 메모리 모두 부족하기 때문에 해시값($\sum w_{e_i} x^{k-i}$)를 저장하는 것이 좋아 보입니다. 하지만 해시값만 저장하면 서로 다른 두 경로의 사전 순 비교가 불가능하기 때문에, 사전 순 비교를 수행하기 위한 다른 추가적인 정보를 함께 갖고 있어야 합니다. $path[l][i][j]$가 $path[l][*][*]$ 중에서 사전 순으로 몇 번째인지 저장하는 배열 $rank[l][i][j]$를 함께 들고 있으면 간선을 a개 사용하는 보행의 정보 N^2개와 b개 사용하는 보행의 정보 N^2개를 총 $O(N^3)$ 시간에 합칠 수 있습니다.

따라서 전체 문제를 $O(N^3 \log K)$에 해결할 수 있습니다. 두 정보를 합치는 구체적인 방법은 아래 코드를 참고하시길 바랍니다.

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
constexpr ll MOD = 1e9+7;
ll Pow(ll a, ll b){
    ll res = 1;
    for(; b; b>>=1, a=a*a%MOD) if(b & 1) res = res * a % MOD;
    return res;
}

int X;
struct matrix{
    int len;
    vector<vector<int>> rank;
```

```
    vector<vector<ll>> hash;
    matrix() = default;
    matrix(int n, int len) : len(len), rank(n, vector<int>(n, -1)), hash(n,
vector<ll>(n, -1)) {
        for(int i=0; i<n; i++) rank[i][i] = 0;
        for(int i=0; i<n; i++) hash[i][i] = 0;
    }
};

matrix operator + (const matrix &a, const matrix &b){
    int n = a.rank.size();
    matrix res(n, a.len + b.len);

    vector<vector<int>> mid(n, vector<int>(n));
    vector<pair<int,int>> comp;

    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            int pos = -1, le = 1e9+7, ri = 1e9+7;
            for(int k=0; k<n; k++){
                if(a.rank[i][k] == -1 || b.rank[k][j] == -1) continue;
                if(tie(a.rank[i][k], b.rank[k][j]) < tie(le, ri)) pos = k, le =
a.rank[i][k], ri = b.rank[k][j];
            }
            mid[i][j] = pos;
            comp.emplace_back(le, ri);
        }
    }

    sort(comp.begin(), comp.end());
    comp.erase(unique(comp.begin(), comp.end()), comp.end());

    for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
            int k = mid[i][j];
            if(k == -1){ res.rank[i][j] = res.hash[i][j] = -1; continue; }
            res.rank[i][j] = lower_bound(comp.begin(), comp.end(),
make_pair(a.rank[i][k], b.rank[k][j])) - comp.begin();
            res.hash[i][j] = (a.hash[i][k] * Pow(X, b.len) + b.hash[k][j]) %
MOD;
        }
    }
    return res;
}

int N, M, S, T, K, A[55][55];

matrix Pow(matrix a, ll b){
    matrix res(a.rank.size(), 0);
    for(; b; b>>=1, a=a+a) if(b & 1) res = res + a;
    return res;
}

int main(){
    ios_base::sync_with_stdio(false); cin.tie(nullptr);
    cin >> N >> M >> S >> T >> X >> K;
    memset(A, -1, sizeof A);
    for(int i=0,s,e,w; i<M; i++) cin >> s >> e >> w, A[s-1][e-1] = w;
```

```
    matrix G(N, 1);
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            G.rank[i][j] = G.hash[i][j] = A[i][j];
        }
    }
    auto R = Pow(G, K);
    cout << R.hash[S-1][T-1];
}
```