2025 SYNU 5月周赛 Round III

很抱歉很抱歉，本场比赛由于中途服务器崩溃，导致后半场比赛直接skip，给您带来了不好的体验QAQ。

题解报告

A. Merge

操作本质上就是删掉一个数，因此原问题为是否能从 a 序列删掉一些数变成 b，实际上是在问 b 是否为 a 的子序列。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define IOS ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0)
// dont use umap!!!

void solve()
{
    int n, m;
    cin >> n >> m;
    if(m > n)
    {
        cout << "No\n";
        return;
    }
    vector<int> a(n + 1), b(m + 1);
    for(int i = 1; i <= n; i ++) cin >> a[i];
    for(int i = 1; i <= m; i ++) cin >> b[i];
    int p = 1;
    for(int i = 1; i <= n and p <= m; i ++) p += a[i] == b[p];
    // cout << p << '\n';
    cout << (p == m + 1 ? "Yes" : "No") << '\n';
}

signed main()
{
    IOS;
    int t = 1;
    // cin >> t;
    while(t --) solve();
    return 0;
}
```

B. Increase

本题本质上是求解长度为4的非降子序列的数量。

考虑dp：

定义 `dp[len][idx]` 为：以序列中第 `idx` 个元素 `a[idx]` **结尾**的，长度为 `len` 的非降子序列的数量。

对于长度为1的非降子序列：任何单个元素 `a[idx]` 本身都构成一个长度为1的非降子序列。
所以， `dp[1][idx] = 1` 对于所有的 $0 \leq idx < n$。

我们要计算 `dp[len][i]` （以 `a[i]` 结尾，长度为 `len` 的非降子序列）。
这个子序列的前 `len-1` 个元素构成了以某个 `a[j]` （其中$j < i$且$a[j] \leq a[i]$）结尾的、长度为 `len-1` 的非降子序列。
因此，状态转移方程为：

$$dp[len][i] = \sum_{i=0}^{len-1} [a_i \leq a_{len}] \cdot dp[i][len-1]$$

最终的答案就是 $\sum dp[4][i]$。

```cpp
#include <bits/stdc++.h>

using namespace std;
using ll = long long;

int n;
int a[5005];
ll dp[5][5005];

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    cin >> n;
    for (int i = 0; i < n; ++i)
        cin >> a[i];

    if (n < 4) {
        cout << 0 << "\n";
        return 0;
    }

    for (int i = 0; i < n; ++i)
        dp[1][i] = 1LL;

    for (int l = 2; l <= 4; ++l) {
        for (int i = 0; i < n; ++i) {
            dp[l][i] = 0LL;
            for (int j = 0; j < i; ++j)
                if (a[j] <= a[i])
                    dp[l][i] += dp[l-1][j];
        }
    }

    ll ans = 0LL;
    for (int i = 0; i < n; ++i)
        ans += dp[4][i];

    cout << ans << "\n";

    return 0;
}
```

C. LCA

换根LCA模板。

我们钦定 1 恒定为根，对于每次询问 R, u, v：

- 若 u, v 都在 R 所在的子树内，则$LCA(u, v)$就是原树的LCA。
- 若 u, v 只有一者在 R 子树内，那么很显然，当 R 为根时，u, v 之间的简单路径一定穿过 R，显然 $LCA(u, v) = R$。
- 若两者均不在 R 子树内，我们假设 $x = LCA(x, R), y = LCA(y, R)$，若 $x \neq y$，则可以发现，$LCA(u, v) =$ 深度较深的那个点，否则 $LCA(x, y) = x = y$。

综上所述，实际上在 R 为根时， $LCA(u, v)$ 就是原 $LCA(u, v)$，x 和 y 中深度最大的点。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define IOS ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
// #define int long long
#define endl '\n'
#define all(_x) _x.begin(), _x.end()
#define matrix(_x, _y, _z) vector<vector<int>>(_x, vector<int>(_y, _z))
#define debug(_x) cout << #_x << '=' << _x << endl
using i64 = long long;
using i128 = __int128;
using pii = pair<int, int>;
using mat = vector<vector<int>>;
using u64 = unsigned long long;
constexpr int N = 2e5 + 10;
// dont use umap!!!

struct LCA {
    std::vector<int> dep, parent, in;
    int cur, n;
    int logn;
    std::vector<std::vector<int>> e;
    vector<vector<int>>a;
    LCA(int _n) : n(_n),dep(_n), parent(_n, -1), e(_n), in(_n), cur(1) {
        logn = std::__lg(n);
        a.assign(logn + 1, std::vector<int>(n + 1));
    }
    //下标从0开始
    void addEdge(int u, int v) {
        e[u].push_back(v);
        e[v].push_back(u);
    }
    void dfs (int x) {
        in[x] =cur++;
        if (cur > 1) {
            a[0][cur - 2] = parent[x];
        }
        for (auto y : e[x]) {
            if (y == parent[x]) continue;
            parent[y] = x;
            dep[y] = dep[x] + 1;
            dfs(y);
        }
    }
    void init(int s) {
        dfs(s);
        for (int j = 0; j < logn; j++) {
            for (int i = 1; i + (2 << j) <= n; i++) {
                a[j + 1][i] = dep[a[j][i]] < dep[a[j][i + (1 << j)]] ? a[j][i] : a[j][i + (1 << j)];
            }
        }
    }
    int  lca(int x, int y) {
        if (x == y) {
            return x;
        }
        if (in[x] > in[y]) {
            std::swap(x, y);
        }
        int k = std::__lg(in[y] - in[x]);
        int u = a[k][in[x]];
        int v = a[k][in[y] - (1 << k)];
        return dep[u] < dep[v] ? u : v;
    }
};

signed main()
```

```
{
    IOS;
    int n, q;
    cin >> n >> q;
    LCA tr(n);
    for(int i = 1; i < n; i ++)
    {
        int x, y;
        cin >> x >> y;
        x --, y --;
        tr.addEdge(x, y);
    }
    tr.init(0);
    while(q --)
    {
        int r, u, v;
        cin >> r >> u >> v;
        r --, u --, v --;
        auto x = tr.lca(u, v), y = tr.lca(r, u), z = tr.lca(r, v);
        if(tr.dep[x] > tr.dep[y])
        {
            if(tr.dep[x] > tr.dep[z]) cout << x + 1 << '\n';
            else cout << z + 1 << '\n';
        }
        else
        {
            if(tr.dep[y] > tr.dep[z]) cout << y + 1 << '\n';
            else cout << z + 1 << '\n';
        }
    }
    return 0;
}
```

D. Damaged Chessboard

首先，考虑完美的棋盘，我们从 $(1,1)$ 走到 (n,m)，一定是向下走 $n-1$ 步，然后向右走 $m-1$ 步，因此我们实际上一共走了 $n+m-2$ 步，所以从 $(1,1)$ 走到 (n,m) 的方案数为 $\binom{n+m-2}{n-1}$。

现在考虑有格子不能走的情况，可以发现并不存在像上述一句话总结的数学结论。此时再来考虑 dp，传统走棋子的dp是 $O(nm)$ 的，通过 $dp[x][y]=dp[x-1][y]+dp[x][y-1]$ 来转移，这显然不可取。此时又想到容斥原理，可以考虑对所有损坏的格子做容斥，但显然也不可取。

结合上述两种做法，我们可以考虑一种更优秀的dp方式。钦定 $f[i]$ 表示从原点走到第 i 个棋(x_i,y_i)，且不经过其它不能走的点的方案数。则 $f[i]$ 就是走到该点的总方案数 $\binom{x_i+y_i+1}{x_i-1}$ 减去前面经过非法位置的方案数总和(对于当前点前方的一个非法点 p，应该减去从原点到 p 之前不经过任何一个黑色格子的总数乘以从 p 到当前点的路线总数)。状态转移方程为：

$$f[i]=\binom{x_i+y_i+1}{x_i-1}\cdot\sum_{j=1}^{n}([x_j\le x_i \text{ and } y_j\le y_i]\cdot f[j]\cdot\binom{x_i-x_j}{x_i+x_j-x_j-y_j})$$

为了方便统计答案，我们假设终点也是一个不能走的点$(n+1)$，则答案为 $f[n+1]$。

注意！dp转移是有先后顺序之分的，因此要先对所有点进行排序，这样才能保证计算当前点时，其左上方的所有点的答案已经被计算出来。

```cpp
#include <bits/stdc++.h>
using namespace std;
#define IOS ios_base::sync_with_stdio(0);cin.tie(0);cout.tie(0);
// #define int long long
#define endl '\n'
#define all(_x) _x.begin(), _x.end()
#define matrix(_x, _y, _z) vector<vector<int>>(_x, vector<int>(_y, _z))
#define debug(_x) cout << #_x << '=' << _x << endl
using i64 = long long;
using i128 = __int128;
using pii = pair<int, int>;
using mat = vector<vector<int>>;
using u64 = unsigned long long;
constexpr int N = 2e5 + 10, mod = 1e9 + 7;
// dont use umap!!!

class Comb
{
    vector<int> fact, invfac;
    int mod;
    public:
    Comb(int limit, int mod = 1e9 + 7) : mod(mod)
    {
        fact.assign(limit, 0);
        invfac.assign(limit, 0);
        fact[0] = invfac[0] = 1;
        for(int i = 1; i < limit; i ++)
        {
            fact[i] = (i64)fact[i - 1] * i % mod;
            invfac[i] = qmi(fact[i], mod - 2);
        }
    }
    Comb() {}
    i64 qmi(i64 a,int k)
    {
        i64 res = 1;
        while(k)
        {
            if(k & 1) res = res * a % mod;
            a = a * a % mod;
            k >>= 1;
        }
        return res;
    }
    i64 get(int n,int m)
    {
        if(m > n) return 0;
        return (i64)fact[n] * invfac[m] % mod * invfac[n - m] % mod;
    }
};

signed main()
{
    IOS;
    int n, m, k;
    cin >> n >> m >> k;
    Comb comb(n + m);
    vector<pii> a(k + 2);
    a[k + 1] = {n, m};
    for(int i = 1; i <= k; i ++)
        cin >> a[i].first >> a[i].second;
    sort(a.begin() + 1, a.end());
    vector<i64> f(k + 2);
    for(int i = 1; i <= k + 1; i ++)
    {
        auto[x1, y1] = a[i];
```

```
            f[i] = comb.get(x1 + y1 - 2, x1 - 1);
            for(int j = 1; j < i; j ++)
            {
                auto[x2, y2] = a[j];
                if(x2 > x1 or y2 > y1) continue;
                f[i] -= f[j] * comb.get(x1 - x2 + y1 - y2, x1 - x2) % mod;
                if(f[i] < 0) f[i] += mod;
            }
        }
        cout << f[k + 1] << '\n';
        return 0;
    }
```