day3NOIP参考题解

本文汇总 A、B、C、D 四题题解。

目录

1. A. 共鸣（resonance）
2. B. 刻度（gauge）
3. C. 晶片（wafer）
4. D. 序册（ledger）

共鸣（resonance）

题意概括

每段旋律用一个字符串表示。给定 n 个带编号的字符串。把每个字符串看成完全图中的一个顶点，边 (i,j) 的权值是 s_i 与 s_j 的最长公共前缀长度。要求把所有标号生成树的边权和全部加起来，并对 10^9+7 取模。

记 $S=\sum_i |s_i|$。字符串可能相同，但编号不同，因此相同内容仍然对应不同顶点。

先处理生成树的部分

固定一条边 (i,j)。在所有标号生成树中，包含这条边的生成树数量与具体的 i,j 无关。把这条边收缩成一个超级顶点，或者直接使用 Cayley 公式的带边版本，可以得到

$$|T:(i,j)\in T| = 2n^{n-3} \qquad (n\ge 3).$$

当 $n=2$ 时完全图本身只有一棵生成树，数量为 1，所以代码单独处理这个边界。

于是可以交换求和顺序：先统计每条完全图边的权值，再乘上它出现在多少棵生成树中。设

$$W=\sum_{1\le i<j\le n}\mathrm{LCP}(s_i,s_j),$$

答案就是 $W\cdot 2n^{n-3}$；$n=2$ 时乘数改为 1，$n=1$ 时答案为 0。

子任务 1：$n\le 8$，$S\le 50$

最直接的做法是枚举每一对字符串，逐字符比较它们的公共前缀。若最长公共前缀长度为 l，就把 l 加入 W。这个做法的时间复杂度为 $O(n^2\cdot L)$，其中 L 是字符串最大长度，在本子任务中完全足够。

这一范围内直接枚举每对字符串即可通过；下面统一采用 Trie 统计，避免按子任务维护不同实现。

子任务 2：$n\le 300$，$S\le 5000$

仍然可以使用两两比较，复杂度上界为 $O(Sn)$。更稳妥的做法是把所有字符串插入 Trie：从根走到某个节点，经过该节点的字符串数量记为 cnt，则该前缀对 W 的贡献是 $\binom{cnt}{2}$。Trie 的总节点数不超过 $S+1$，所以构建和统计都是 $O(S)$。

子任务 3：$n\le 2000$，$S\le 5000$

字符串总长度仍然只有 5000，Trie 的节点数仍然是 $O(S)$。因此不需要依赖 n 较小的两两枚举，直接按字符插入 Trie 即可。每个字符只被访问一次，之后对每个 Trie 节点做一次组合数统计。

该算法与满分算法相同，复杂度为 $O(S+\log n)$，其中快速幂的 $\log n$ 项很小。

子任务 4：所有字符串完全相同

若每个字符串长度为 L，则任意一对顶点的边权都是 L，所以也可以直接得到

$$W=\binom{n}{2}L.$$

不过通用 Trie 在这种数据上只产生一条长度为 L 的链，同样只需 $O(S)$ 时间，因此仍可使用统一算法。

子任务 5：满分算法

1. Trie 中的贡献

Trie 的根表示空前缀，不计入答案。对每个非根节点 v，设 count 为所有经过 v 的字符串数量。恰好这 count 个字符串两两共享由根到 v 的这一段前缀，因此前缀深度对应的贡献为

$$\binom{count}{2} = \frac{count(count-1)}{2}.$$

一个字符串对共享长度为 l，当且仅当它们在 Trie 中共同经过深度为 $1, 2, \ldots, l$ 的节点，所以这对字符串会在上式中被计算恰好 l 次。对所有非根节点求和便得到

$$W = \sum_{v \neq root} \binom{count_v}{2}.$$

2. 构建 Trie

依次插入每个字符串。当前节点为 u，读入字符 c 后沿着 child[c] 走；如果边不存在，就新建节点。每走到一个非根节点，就把它的 count 加一。因为每个输入字符只经过一次，构建复杂度为 $O(S)$。

3. 生成树乘数

快速幂计算 $n^{n-3} \bmod MOD$，再乘以 2。代码在 $n = 2$ 时输出乘数 1，在 $n = 1$ 时直接输出 0。组合数先在 long long 中计算再取模，避免 int 乘法溢出。

正确性说明

对任意字符串对 (i, j)，它们的公共前缀的每一层在 Trie 中对应唯一节点；这对字符串恰好同时经过这些节点，因此在所有 $\binom{count_v}{2}$ 中被计算的次数等于 $\mathrm{LCP}(s_i, s_j)$。所以 Trie 求出的 W 正确。

另一方面，每条完全图边都出现在相同数量的标号生成树中。把每条边的权值乘以这个数量并求和，正好等于所有生成树的总边权和。因此最终乘积就是题目要求的答案。

复杂度

Trie 节点数为 $O(S)$。时间复杂度为 $O(S + \log n)$，空间复杂度为 $O(S)$；其中每个节点保存 26 条字符边和一个计数。

实现要点

trie 保存 Trie，count 保存经过节点的字符串数，prefixSum 就是 W，treeCount 是固定边出现在多少棵生成树中的数量。

完整代码

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

using namespace std;

const long long MOD = 1000000007LL;

long long modPow(long long a, long long e) {
    long long r = 1;
    while (e > 0) {
        if (e & 1) r = r * a % MOD;
        a = a * a % MOD;
        e >>= 1;
    }
    return r;
}

struct Node {
    int child[26];
    long long count;

    Node() : count(0) {
        fill(child, child + 26, -1);
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    freopen("resonance.in", "r", stdin);
    freopen("resonance.out", "w", stdout);
```

```
    int n;
    if (!(cin >> n)) return 0;

    vector<string> s(n);
    for (string &x : s) cin >> x;

    if (n < 2) {
        cout << 0 << '\n';
        return 0;
    }

    vector<Node> trie(1);
    for (const string &x : s) {
        int u = 0;
        for (char ch : x) {
            int c = ch - 'a';
            if (trie[u].child[c] == -1) {
                trie[u].child[c] = static_cast<int>(trie.size());
                trie.emplace_back();
            }
            u = trie[u].child[c];
            ++trie[u].count;
        }
    }

    long long prefixSum = 0;
    for (size_t i = 1; i < trie.size(); ++i) {
        prefixSum = (prefixSum + trie[i].count * (trie[i].count - 1) / 2) % MOD;
    }

    long long treeCount;
    if (n == 2) {
        treeCount = 1;
    } else {
        treeCount = 2 * modPow(n, n - 3) % MOD;
    }

    cout << treeCount * prefixSum % MOD << '\n';
    return 0;
}
```

刻度（gauge）题解

将标尺按放入顺序记为 $b_1, b_2, \ldots, b_n$，并令 $g_i = \gcd(b_1, \ldots, b_i)$。我们需要最大化所有前缀最大公约数之和。

对于一个正整数 d，记 c_d 为输入中能被 d 整除的元素个数。若当前前缀的最大公约数为 d，那么后续只能把它变成 d 的约数。令 F_d 表示从一个“已经选完所有能被 d 整除的元素”的状态继续安排时，可以得到的最大贡献。若下一段的公约数为 j，其中 $d \mid j$ 且 $j > d$，则这段中被 j 统计的元素在降到 d 时需要扣除 dc_j，因此有转移

$$F_d = dc_d + \max\left(0, \max_{j>d,\, d\mid j} (F_j - dc_j)\right).$$

按 d 从大到小计算即可。朴素地枚举每个数的所有倍数需要接近 $M \log M$ 的数组扫描，其中 $M = \max a_i$。本实现改为只枚举输入值和状态值的因数：先用最小质因子筛分解每个不同的输入值，把它的所有因数加入 c_d；计算 F_j 后，再枚举 j 的因数 d 更新

$$H_d = \max(H_d, F_j - dc_j), \qquad F_d = dc_d + H_d.$$

这样只处理实际出现的约数，答案仍与倍数递推完全相同。数组大小为 $O(M)$，因数枚举的总量为所有不同状态值的约数个数之和；在本题范围内远小于完整的倍数双循环。

子任务 1：$n \le 16$

用集合状态记录已经使用的元素和当前 gcd，枚举下一个未使用元素，复杂度为 $O(n2^n)$。

子任务 2：$\max a_i \le 5000$

使用按倍数统计 c_d，再按 d 从大到小枚举所有倍数完成转移，复杂度为 $O(M \log M)$。

子任务 3：$\max a_i \le 10^6$

算法与子任务 2 相同，只需把计数数组扩大到 10^6。

子任务 4：完整约束

使用最小质因子筛和实际因数枚举，避免对 1 到 M 的每一个位置都扫描所有倍数。答案使用 `long long` 保存，完整实现见下方代码。

正确性依据：任何排列都可以按 gcd 不变的连续段划分；段 gcd 严格下降时，新段的元素必然是前一个 gcd 的倍数，而转移恰好扣除了这些元素在较大 gcd 段中已经贡献的部分。反之，每条合法的 gcd 链都能按段内元素构造出一个排列。因此递推覆盖且只覆盖所有排列，F_1 即为最优答案。

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

int main() {
    freopen("gauge.in", "r", stdin);
    freopen("gauge.out", "w", stdout);
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    if (!(cin >> n)) return 0;
    vector<int> values(n);
    for (int &x : values) {
        cin >> x;
    }
    sort(values.begin(), values.end());
    int mx = values.back();

    vector<int> spf(mx + 1);
    for (int i = 2; i <= mx; ++i) {
        if (spf[i]) continue;
        spf[i] = i;
        if (i <= mx / i) {
            for (int j = i * i; j <= mx; j += i) {
                if (!spf[j]) {
                    spf[j] = i;
                }
            }
        }
    }

    vector<int> count(mx + 1);
    for (int i = 0; i < n;) {
        int j = i + 1;
        while (j < n && values[j] == values[i]) {
            ++j;
        }
        int x = values[i], multiplicity = j - i;
```

```
        vector<pair<int, int>> factors;
        while (x > 1) {
            int p = spf[x], e = 0;
            do {
                x /= p;
                ++e;
            } while (x > 1 && spf[x] == p);
            factors.push_back({p, e});
        }
        vector<int> divisors{1};
        for (auto [p, e] : factors) {
            int old_size = (int)divisors.size(), power = 1;
            for (int k = 1; k <= e; ++k) {
                power *= p;
                for (int z = 0; z < old_size; ++z)
                    divisors.push_back(divisors[z] * power);
            }
        }
        for (int d : divisors) {
            count[d] += multiplicity;
        }
        i = j;
    }

    vector<long long> best(mx + 1);
    long long answer = 0;
    for (int j = mx; j >= 1; --j) {
        if (!count[j]) continue;
        long long current = 1LL * j * count[j] + best[j];
        if (j == 1) {
            answer = current;
            break;
        }
        int x = j;
        vector<pair<int, int>> factors;
        while (x > 1) {
            int p = spf[x], e = 0;
            do {
                x /= p;
                ++e;
            } while (x > 1 && spf[x] == p);
            factors.push_back({p, e});
        }
        vector<int> divisors{1};
        for (auto [p, e] : factors) {
            int old_size = (int)divisors.size(), power = 1;
            for (int k = 1; k <= e; ++k) {
                power *= p;
                for (int z = 0; z < old_size; ++z)
                    divisors.push_back(divisors[z] * power);
            }
        }
        for (int d : divisors) {
            if (d == j) continue;
            best[d] = max(best[d], current - 1LL * d * count[j]);
        }
    }
    cout << answer << '\n';
    return 0;
}
```

晶片（wafer）

题意概括

给定一个 $N \times M$ 网格，`1` 是可用格，`0` 是障碍格。截取区域必须是边与网格平行的正方形；边长为 d 的截取区域覆盖连续的 d 行和连续的 d 列。对每个询问格 (x, y)，求一个完全由 `1` 组成且包含该格的正方形的最大面积 d^2。

如果询问格本身是 `0`，答案自然为 0。下面把“某个正方形全部可用”统一称为有效。

子任务 1：$N, M \le 50$，$Q \le 1000$

可以对每个询问枚举边长 d，再枚举所有可能的左上角 (a, b)。用二维前缀和在 $O(1)$ 时间判断一个 $d \times d$ 方块是否含障碍，找到最大的有效边长后输出平方。二分边长也可以把单个询问的枚举次数降到 $O(\log \min(N, M))$。

满分算法在这个小范围内同样适用。

子任务 2：$N, M \le 800$，$Q \le 10^5$

查询数已经很大，不能对每个询问重新扫描方块。需要把“每个左上角能放多大的有效正方形”预处理出来，再用二维区间最大值回答询问，使用单调队列加二维 Sparse Table 即可完成。

子任务 3：$N \le 10$，$M \le 2000$，$Q \le 500$

网格较窄，可以对每一列维护从当前位置向右的连续 `1` 长度，并枚举可能的正方形边长；也可以直接使用下面的预处理。满分算法的复杂度按一般的 N, M 估计，因而覆盖本子任务。

子任务 4：$N, M \le 2000$，$Q \le 10^5$

使用完整算法。障碍数量 K 只影响数据分布，不改变算法；稀疏表的最大内存按最坏的 $N = M = 2000$ 估算。

满分算法

1. 固定左上角的最大正方形

定义

$$right[i][j] = \text{从 } (i, j) \text{ 开始向右连续的可用格数量.}$$

若 (i, j) 是障碍格，则 $right[i][j] = 0$；否则

$$right[i][j] = right[i][j+1] + 1.$$

设 $best[i][j]$ 为左上角固定为 (i, j) 时，有效正方形的最大边长。一个边长为 t 的正方形有效，当且仅当它覆盖的每一行在区间 $[j, j+t-1]$ 内都有至少 t 个连续可用格。因此

$$best[i][j] = \max t : \min_{r=i}^{i+t-1} right[r][j] \ge t.$$

朴素地对每个 (i, j) 检查所有 t 会很慢。按列从下到上扫描，并维护一个单调队列：

- 队列中的行按 $right[r][j]$ 非降序排列，队首给出当前候选行的最小水平连续长度；
- 若队首行的水平长度小于它到当前行之间需要覆盖的高度，就把它弹出，并把它记为当前不能再扩展的最下方障碍边界；
- 当前答案是“最小水平长度”和“距离最近失效边界”中的较小值。

这个过程的关键是不再为每个边长重复检查所有行。每行只进出队列一次，所以所有列的总复杂度是 $O(NM)$。将结果保存在 `best` 数组中。

从定义也可以看出，`best[i][j]` 至少为 t，当且仅当以 (i, j) 为左上角的 $t \times t$ 方块有效。

2. 把一个询问转成矩形最大值

询问格为 (x, y)，固定一个候选边长 t。正方形左上角 (a, b) 必须满足

$$x - t + 1 \le a \le x, \qquad y - t + 1 \le b \le y.$$

因此，只要矩形

$$R_t = [x - t + 1, x] \times [y - t + 1, y]$$

中存在一个位置 (a, b) 使 `best[a][b]` 至少为 t，就存在一个包含询问格的有效正方形。

这还自动处理了边界：如果左上角使正方形越出网格，则对应的 `best` 不会达到该边长。给数组增加平移量 `ad=min(N,M)`，让所有查询下标都落在稀疏表的安全区域。

3. 二维 Sparse Table

对 `best` 建二维幂次表。令 `step[i][j][k]` 表示以 (i, j) 为左上角、边长为 2^k 的方块内最大 `best`。转移由四个边长为 2^{k-1} 的子方块取最大值：

$$step(i,j,k) = \max\{step(i,j,k-1),\ step(i+2^{k-1},j,k-1),\ step(i,j+2^{k-1},k-1),\ step(i+2^{k-1},j+2^{k-1},k-1).$$

要查询任意 $t \times t$ 矩形，令 $p = \lfloor \log_2 t \rfloor$，$l = 2^p$。四个边长为 l 的方块覆盖整个矩形，因此只需取四次 `step` 的最大值。`check(x,y,t)` 就是在做这件事，并判断最大值是否至少为 t。

4. 二分边长

如果存在一个包含 (x, y) 的有效 $t \times t$ 正方形，那么把它缩小成边长 $t - 1$ 的正方形仍可以包含 (x, y)。所以可行性关于 t 单调，答案可以在 $[0, \min(N, M)]$ 上二分。每次检查只需要常数次 Sparse Table 查询，单个询问耗时 $O(\log \min(N, M))$。

正确性说明

首先，`right[i][j]` 的递推直接描述了从该格开始向右的连续可用长度，因此它准确记录了每一行对某个正方形的水平支持范围。

对固定左上角 (i, j)，边长为 t 的方块有效，当且仅当前 t 行的 `right` 都至少为 t。单调队列始终保留这些行中可能成为最小值的行，队首是当前最小水平长度；被弹出的第一行恰好限制了向上扩展的最大高度。因此 `best` 等于最大可行 t。

随后，询问边长为 t 的可行性等价于矩形 R_t 中是否存在 `best` 至少为 t 的左上角。四块 Sparse Table 覆盖 R_t，所以 `check` 返回真当且仅当该边长可行。最后利用可行性的单调性二分，得到的就是最大边长，输出其平方即为题目答案。

复杂度与实现细节

单调队列预处理为 $O(NM)$。二维 Sparse Table 需要 $O((N + M)^2 \log \min(N, M))$ 的存储上界；实现按平移后的数组分配，实际常数由 `MAXN` 和 `MAXLOG` 固定。建表时间同阶。每个询问为 $O(\log \min(N, M))$，总时间为

$$O\big(NM + (N + \min(N, M))^2 \log \min(N, M) + Q \log \min(N, M)\big).$$

需要特别处理边长 1：此时 $p = 0$，四块查询都退化为单点。询问格是 `0` 时所有检查都会失败，二分结果为 0，无需额外分支。

各个子任务均可使用同一份满分算法；差别只在所面对的约束范围。

完整代码

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

using namespace std;

const int MAXN = 4000 + 10;
const int MAXLOG = 12;

int n, m, q;
int x, y;
char grid[MAXN][MAXN];
int horizontal[MAXN][MAXN];
int best[MAXN][MAXN];
int sparse[MAXN][MAXN][MAXLOG];

bool checkSquare(int queryX, int queryY, int side) {
    int power = static_cast<int>(floor(log(side) / log(2)));
    int block = 1 << power;

    int x1 = queryX - side + 1;
    int y1 = queryY - side + 1;
    int x2 = queryX - block + 1;
    int y2 = queryY - block + 1;

    int maximum = max({
        sparse[x1][y1][power],
        sparse[x2][y1][power],
        sparse[x1][y2][power],
```

```
        sparse[x2][y2][power]
    });
    return maximum >= side;
}

int main() {
    freopen("wafer.in", "r", stdin);
    freopen("wafer.out", "w", stdout);

    scanf("%d%d", &n, &m);
    int offset = min(n, m);

    for (int i = 1; i <= n; ++i) {
        scanf("%s", grid[i] + 1);
        for (int j = m; j >= 1; --j) {
            horizontal[i][j] = (grid[i][j] == '1')
                               ? horizontal[i][j + 1] + 1
                               : 0;
        }
    }

    deque<int> candidates;
    for (int column = 1; column <= m; ++column) {
        candidates.clear();
        int lastInvalid = n + 1;

        for (int row = n; row >= 1; --row) {
            while (!candidates.empty()
                   && horizontal[candidates.back()][column] > horizontal[row][column]) {
                candidates.pop_back();
            }
            while (!candidates.empty()
                   && horizontal[candidates.front()][column] < candidates.front() - row + 1) {
                lastInvalid = candidates.front();
                candidates.pop_front();
            }

            candidates.push_back(row);
            if (candidates.empty()) {
                best[row][column] = (grid[row][column] == '1');
            } else {
                best[row][column] = min(horizontal[candidates.front()][column],
                                        lastInvalid - row);
            }
        }
    }

    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            sparse[i + offset][j + offset][0] = best[i][j];
        }
    }

    for (int level = 1; level < MAXLOG; ++level) {
        // Index 1 is only padding; every query starts at an index at least 2.
        for (int i = 2; i <= n + offset; ++i) {
            for (int j = 2; j <= m + offset; ++j) {
                int shift = 1 << (level - 1);
                sparse[i][j][level] = max({
                    sparse[i][j][level - 1],
                    sparse[i][min(m + offset, j + shift)][level - 1],
                    sparse[min(n + offset, i + shift)][j][level - 1],
                    sparse[min(n + offset, i + shift)][min(m + offset, j + shift)][level - 1]
                });
            }
        }
    }

    scanf("%d", &q);
    while (q--) {
        scanf("%d%d", &x, &y);

        int low = 0;
        int high = offset;
        while (low < high) {
            int middle = (low + high + 1) >> 1;
            if (checkSquare(x + offset, y + offset, middle)) {
                low = middle;
            } else {
                high = middle - 1;
            }
        }
        printf("%d\n", low * low);
    }

    return 0;
}
```

序册（ledger）

比较两份副本

每个位置可填 1 到 26；实现中使用 0 到 25 表示这些值，大小关系不变。

考虑限制 $T_a \le T_b$。若 $a < b$，两个数列在前面一段完全相同；接下来依次比较相邻元素对 $(S_a, S_{a+1}), (S_{a+1}, S_{a+2}), \dots, (S_{b-1}, S_b)$。因此在边区间 $[a, b-1]$ 内，找到第一条两端元素不同的边后，它必须满足 $S_i > S_{i+1}$；如果区间内所有相邻元素都相等，则两份副本相等，限制也成立。若 $a > b$，方向相反，区间内第一条变化边必须满足 $S_i < S_{i+1}$。

把每条相邻边标成上升、下降或相等。每个限制只约束其区间中第一条非相等边的方向。

subtask 1（$N \le 10$）

枚举 $N-1$ 条边的三种关系，共 3^{N-1} 种关系串。对每个关系串，逐条检查每个限制：在相应边区间找到第一条非零关系，确认方向符合要求。朴素检查每条限制最多扫描 $N-1$ 条边，因此每个关系串的检查时间为 $O(MN)$。若全部满足，再统计有多少个长度为 N 的数列实现该关系串。固定关系串后，用最后一个数值做 26 状态 DP：相等边保持数值，上升边只从更小的旧数值转移，下降边只从更大的旧数值转移，时间为 $O(26N)$。总时间复杂度为 $O(3^N(MN + 26N))$，空间复杂度为 $O(N + M)$。

subtask 2（$N \le 200$）

对每个边位置 e 和每种方向，找出覆盖 e 的限制中最大的左端点 $L[e]$。用排序加优先队列维护活跃区间，可在 $O(M \log M)$ 时间完成。设 `dp[c][p]` 为：前缀最后一个数值是 c，最近一次相邻元素变化的位置对应的前缀端点是 p 时的方案数。若当前边发生上升，那么它不能是任何“要求第一条变化边下降”的活跃限制区间中的第一条变化边；因此之前的最近变化必须已经位于该区间左端点之后，即 $p \ge L[e] + 1$。下降转移同理。直接枚举新数值、旧数值和可能的 p，时间复杂度为 $O(26^2N^2 + M \log M)$，空间复杂度为 $O(26N + M)$。

subtask 3（限制构成排列路径）

给定的 $N-1$ 条限制按某个排列 P 串成一条路径。每条限制仍对应一个边区间和方向。把区间按左端点分桶，从大左端点向小左端点扫描；并查集跳过已赋值的边，就能给每个位置找到覆盖它且左端点最大的区间。每个位置只会被赋值一次。之后对数值和前缀端点做同样的 DP，但用前缀和在 $O(26)$ 时间完成一条边的所有转移。总复杂度为 $O(26N + M + N\alpha(N))$，空间复杂度为 $O(26N + M)$。

subtask 4（$N \le 20,000$）

使用完整的区间边界预处理和数值前缀和 DP。对于每种数值，前缀和能在常数时间查询允许范围 $p \in [L[e] + 1, e]$；按数值从小到大或从大到小累积，就能一次处理所有更小或更大的旧数值。$N \le 20,000$ 时可直接使用这套 DP；更大的输入则先验证它满足排列路径条件，再运行同一套算法。因此它覆盖本子任务，并额外通过了第 3 子任务的路径数据。时间复杂度为 $O(26N + M \log M)$，空间复杂度为 $O(26N + M)$。

subtask 5（无额外限制）

完整范围也使用上面的优化 DP。按方向把 M 条限制转为闭区间，扫描边时用最大堆维护当前覆盖区间中最大的左端点。随后维护每个数值在所有最近变化位置上的前缀和，单条边只需分别从小数值到大数值、从大数值到小数值扫描一次。总复杂度为 $O(26N + M \log M)$，空间复杂度为 $O(26N + M)$。

满分

更具体地说，令 $L_{\uparrow}[e]$ 为覆盖边 e 且要求第一条变化边下降的限制中最大左端点，令 $L_{\downarrow}[e]$ 为覆盖边 e 且要求第一条变化边上升的限制中最大左端点。若不存在对应区间，取 -1。新边为上升时，只能从最近变化位置 $p \in [L_{\uparrow}[e] + 1, e]$ 的前缀转移；下降时使用 $L_{\downarrow}[e]$。相等时最近变化位置不变。

令 `prefix[c][x]` 保存数值 c 且最近变化位置小于 x 的方案数前缀和。上升时依次枚举新数值 c，累加数值小于 c 的所有合法旧状态；下降时反向枚举，累加数值大于 c 的旧状态。对每个数值做一次区间前缀和查询，避免枚举所有旧位置。相等关系直接延续同一个数值的状态。最后把所有末数值和最近变化位置的状态相加，对 $10^9 + 7$ 取模。

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

static constexpr int MOD = 1'000'000'007;
static constexpr int SIGMA = 26;

struct Interval {
    int left;
    int right;
};
```

```
vector<int> maximumActiveLeft(int n, vector<Interval> intervals) {
    sort(intervals.begin(), intervals.end(), [](const Interval &a, const Interval &b) {
        return a.left < b.left;
    });

    priority_queue<pair<int, int>> active;
    vector<int> result(n - 1, -1);
    size_t next = 0;
    for (int edge = 0; edge + 1 < n; ++edge) {
        while (next < intervals.size() && intervals[next].left <= edge) {
            active.push({intervals[next].left, intervals[next].right});
            ++next;
        }
        while (!active.empty() && active.top().second < edge) active.pop();
        if (!active.empty()) result[edge] = active.top().first;
    }
    return result;
}

int main() {
    freopen("ledger.in", "r", stdin);
    freopen("ledger.out", "w", stdout);


    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    if (!(cin >> n >> m)) return 0;

    vector<Interval> increasingGroup, decreasingGroup;
    increasingGroup.reserve(m);
    decreasingGroup.reserve(m);
    for (int i = 0; i < m; ++i) {
        int a, b;
        cin >> a >> b;
        --a;
        --b;
        if (a < b) increasingGroup.push_back({a, b - 1});
        else decreasingGroup.push_back({b, a - 1});
    }

    const vector<int> leftIncreasing = maximumActiveLeft(n, move(increasingGroup));
    const vector<int> leftDecreasing = maximumActiveLeft(n, move(decreasingGroup));

    vector<vector<int>> dp(SIGMA, vector<int>(n, 0));
    vector<vector<int>> prefix(SIGMA, vector<int>(n + 1, 0));
    for (int letter = 0; letter < SIGMA; ++letter) {
        dp[letter][0] = 1;
        prefix[letter][1] = 1;
    }

    for (int edge = 0; edge + 1 < n; ++edge) {
        int lower = leftIncreasing[edge] + 1;
        int accumulated = 0;
        for (int letter = 0; letter < SIGMA; ++letter) {
            dp[letter][edge + 1] += accumulated;
            if (dp[letter][edge + 1] >= MOD) dp[letter][edge + 1] -= MOD;
            int rangeSum = prefix[letter][edge + 1] - prefix[letter][lower];
            if (rangeSum < 0) rangeSum += MOD;
            accumulated += rangeSum;
            if (accumulated >= MOD) accumulated -= MOD;
        }

        lower = leftDecreasing[edge] + 1;
        accumulated = 0;
        for (int letter = SIGMA - 1; letter >= 0; --letter) {
            dp[letter][edge + 1] += accumulated;
            if (dp[letter][edge + 1] >= MOD) dp[letter][edge + 1] -= MOD;
            int rangeSum = prefix[letter][edge + 1] - prefix[letter][lower];
            if (rangeSum < 0) rangeSum += MOD;
            accumulated += rangeSum;
            if (accumulated >= MOD) accumulated -= MOD;
        }

        for (int letter = 0; letter < SIGMA; ++letter) {
            prefix[letter][edge + 2] = prefix[letter][edge + 1] + dp[letter][edge + 1];
            if (prefix[letter][edge + 2] >= MOD) prefix[letter][edge + 2] -= MOD;
        }
    }

    int answer = 0;
    for (int letter = 0; letter < SIGMA; ++letter) {
        for (int lastChange = 0; lastChange < n; ++lastChange) {
            answer += dp[letter][lastChange];
            if (answer >= MOD) answer -= MOD;
        }
    }
```

```
    cout << answer << '\n';
    return 0;
}
```