苏州市人工智能学会 青少年人工智能核心算法素养考核（五级）题解报告

> 考核日期：2026年6月28日

目录

1. Token 管理

1.1 题目描述

某公司部署了 n 个 AI 模型服务，编号为 1 到 n。最开始（时刻 0），第 i 个模型剩余的 Token 配额为 a_i。

由于持续有用户调用推理接口，模型的 Token 配额会随着时间自动消耗：如果某个模型在某段时间内没有被充值，那么每经过 1 单位时间，它的 Token 配额就会减少 1。但 Token 配额最低不会降到 0 以下（即配额耗尽后不再继续扣减）。

接下来有 q 个事件按时间顺序发生。第 j 个事件发生在时刻 t_j，保证时间递增，即 $0 < t_1 < t_2 < \cdots < t_q$。

事件共有以下两种类型：

1. `1 t x v`：在时刻 t，为第 x 个模型充值 Token，使其配额增加 v。
2. `2 t x`：在时刻 t，查询第 x 个模型当前的剩余 Token 配额。

请按顺序处理所有的事件，并输出所有类型为 2 的事件的答案。

1.2 数据范围

- 对于 50% 的数据：$1 \le n, q \le 10$
- 对于 100% 的数据：$1 \le n, q \le 2 \times 10^5$，$0 \le a_i \le 10^9$，$1 \le t_i \le 10^9$，$1 \le x \le n$，$1 \le v \le 10^9$

1.3 题目分析

核心难点在于时间跨度可达 10^9，不能逐单位时间模拟。每个模型独立衰减，且衰减到 0 后不再继续扣减。

1.4 部分分解法（50%）

思路：每次事件发生时，将所有模型的 Token 统一减去时间差。虽然简单直观，但每次事件都要遍历所有 n 个模型，复杂度 $O(nq)$。

参考程序：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;          // 使用 long long 防止溢出
const int MAXN = 200005;       // 最大数据规模

ll a[MAXN];                    // a[i] 表示第 i 个模型的当前 Token 数

int main() {
    // 重定向文件输入输出
    freopen("manage.in", "r", stdin);
    freopen("manage.out", "w", stdout);

    int n, q;
    cin >> n >> q;

    // 读入每个模型的初始 Token 配额
    for (int i = 1; i <= n; i++) cin >> a[i];

    ll last_t = 0;              // 记录上一次处理事件的时间
    while (q--) {
        int op, t, x;
        cin >> op >> t;         // 读入事件类型和发生时刻

        // 计算从上一次事件到本次事件的时间差
        ll dt = t - last_t;

        // 所有模型统一扣除时间差对应的 Token 数
        for (int i = 1; i <= n; i++)
            a[i] = max(0LL, a[i] - dt);   // Token 不低于 0

        last_t = t;             // 更新最后处理时间

        if (op == 1) {
            // 充值事件：1 t x v
            ll v; cin >> x >> v;
            a[x] += v;          // 第 x 个模型增加 v 个 Token
        } else {
            // 查询事件：2 t x
            cin >> x;
            cout << a[x] << '\n';  // 输出第 x 个模型的当前 Token 数
        }
    }
    return 0;
}
```

1.5 满分解法（100%）

思路：延迟更新（Lazy Update）。每个模型只在实际被操作时才计算时间衰减。维护两个数组：

- `cur[x]`：模型 x 当前记录的 Token 数
- `last_time[x]`：模型 x 上次被更新的时刻

处理模型 x 的事件时：

1. 计算时间差 $dt = t - last\_time[x]$
2. 更新 $cur[x] = \max(0, cur[x] - dt)$
3. 更新 $last\_time[x] = t$

4. 如果是充值事件，$cur[x] += v$

这样每个事件只涉及一个模型，总复杂度 $O(n+q)$。

参考程序：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;           // 使用 long long 防止溢出
const int MAXN = 200005;        // 最大数据规模

ll cur[MAXN];                   // cur[i] 表示第 i 个模型当前记录的 Token 数
ll last_time[MAXN];             // last_time[i] 表示第 i 个模型上次被更新的时刻

int main() {
    // 重定向文件输入输出
    freopen("manage.in", "r", stdin);
    freopen("manage.out", "w", stdout);

    int n, q;
    cin >> n >> q;

    // 读入初始 Token 配额，并将每个模型的上次更新时间设为 0
    for (int i = 1; i <= n; i++) {
        cin >> cur[i];
        last_time[i] = 0;       // 时刻 0 开始
    }

    while (q--) {
        int op, t, x;
        cin >> op >> t;          // 读入事件类型和发生时刻

        if (op == 1) {
            // 充值事件：1 t x v
            ll v; cin >> x >> v;

            // 先计算从上次更新到现在的 Token 衰减
            ll dt = t - last_time[x];
            cur[x] = max(0LL, cur[x] - dt);   // 扣除时间差，不低于 0

            last_time[x] = t;   // 更新该模型的上次处理时间
            cur[x] += v;        // 增加 v 个 Token
        } else {
            // 查询事件：2 t x
            cin >> x;

            // 同样先计算衰减
            ll dt = t - last_time[x];
            cur[x] = max(0LL, cur[x] - dt);

            last_time[x] = t;   // 更新该模型的上次处理时间
            cout << cur[x] << '\n';  // 输出当前 Token 数
        }
    }
    return 0;
}
```

2. 矩阵中的我

2.1 题目描述

你正在训练一个大规模多模态 AI 模型，该模型包含一个 $n \times m$ 的**特征交互矩阵**，其中第 i 行第 j 列的元素表示**特征 i 与特征 j 的交叉注意力权重**，其值为 $i \times j$（行和列都从 1 开始编号）。

在模型推理过程中，你需要将这 $n \times m$ 个注意力权重按照**非严格单调递增**的顺序依次输入到**排序归一化层**中进行处理。

请问，你输入的第 k 个注意力权重是多少。

2.2 数据范围

- 对于 30% 的数据：$1 \leq n, m \leq 500$
- 对于 100% 的数据：$1 \leq n, m \leq 5 \times 10^5$，$1 \leq k \leq n \times m$

2.3 题目分析

矩阵规模最大可达 2.5×10^{11} 个元素，无法全部生成和排序。需要利用矩阵元素的特殊规律：$a[i][j] = i \times j$。

2.4 部分分解法（30%）

思路：生成所有 $n \times m$ 个元素，排序后取第 k 个。复杂度 $O(nm \log(nm))$。

参考程序：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;          // 使用 long long 防止 i*j 溢出

int main() {
    // 重定向文件输入输出
    freopen("grid.in", "r", stdin);
    freopen("grid.out", "w", stdout);

    int n, m; ll k;
    cin >> n >> m >> k;        // 读入行数、列数和第 k 个元素

    vector<ll> vals;           // 存储矩阵中所有元素
    // 双重循环生成所有 i*j 的值
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            vals.push_back(1LL * i * j);   // 注意转为 long long

    // 将整个数组排序，得到非严格单调递增序列
    sort(vals.begin(), vals.end());

    // 输出第 k 个元素（下标从 0 开始，所以取 k-1）
    cout << vals[k - 1] << '\n';
    return 0;
}
```

2.5 满分解法（100%）

思路：二分答案 + 计数。

对于猜测值 x，统计矩阵中 $\le x$ 的元素个数。由于第 i 行的元素为 $i \times 1, i \times 2, \ldots, i \times m$，因此第 i 行中 $\le x$ 的列数为 $\min(m, \lfloor x/i \rfloor)$。

总计数为：
$count(x) = \sum_{i=1}^{n} \min\left(m, \lfloor \frac{x}{i} \rfloor\right)$

二分查找最小的 x 使得 $count(x) \ge k$。

复杂度 $O(n \log(nm))$，约 2×10^7 次操作，1 秒内可通过。

参考程序：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;          // 使用 long long 防止溢出

/**
 * 统计矩阵中值 <= x 的元素个数
 * 第 i 行中 <= x 的列数为 min(m, x / i)
 */
ll count_leq(ll x, int n, int m) {
    ll cnt = 0;
    for (int i = 1; i <= n; i++) {
        // 第 i 行中值不超过 x 的元素个数 = min(m, floor(x / i))
        cnt += min((ll)m, x / i);
    }
    return cnt;
}

int main() {
    // 重定向文件输入输出
    freopen("grid.in", "r", stdin);
    freopen("grid.out", "w", stdout);

    int n, m; ll k;
    cin >> n >> m >> k;         // 读入行数、列数和第 k 个元素

    // 二分答案：在 [1, n*m] 范围内查找
    ll lo = 1, hi = 1LL * n * m;
    while (lo < hi) {
        ll mid = (lo + hi) / 2;             // 猜测中间值
        if (count_leq(mid, n, m) >= k)      // 如果 <= mid 的元素不少于 k 个
            hi = mid;                       // 答案在左半部分
        else
            lo = mid + 1;                   // 答案在右半部分
    }
    // lo 即为第 k 小的元素值
    cout << lo << '\n';
    return 0;
}
```

3. 数据集指派

3.1 题目描述

哪吒和孙悟空正在合作标注一个包含 n 条样本的数据集，样本编号为 1 到 n。第 i 条样本记录了当前标注负责人 S_i 和标注难度系数 P_i。当 S_i 为 `N` 时，负责人是哪吒（Nezha）；当 S_i 为 `W` 时，负责人是孙悟空（Wukong）。

反转负责人是指将 `N` 改为 `W`，将 `W` 改为 `N`。

给定 Q 个询问。对于第 j 个询问，你可以执行以下操作至多一次（也可以选择不执行操作）：

> **操作**：选择一对整数 (l, r)，满足 $L_j \le l \le r \le R_j$，并将第 l 到第 r 条样本（含两端）的所有标注负责人反转。

考虑操作后（或不执行操作时），在第 L_j 到第 R_j 条样本中，由**孙悟空（W）**负责的样本的标注难度系数之和。求当操作选择最优时，这个总和的最小值。

每个询问是独立的。

3.2 数据范围

- 对于 30% 的数据：$1 \le n \le 1000$
- 对于另 40% 的数据：$1 \le n \le 10000$
- 对于 100% 的数据：$1 \le n \le 10^5$，$1 \le Q \le 100$，$1 \le P_i \le 10^9$，$1 \le L_j \le R_j \le n$

3.3 题目分析

关键转化：定义每个位置的"价值"（翻转后 W 和的变化量）：

- 若 $S_i = N$：翻转后变为 W，W 和增加 P_i，价值 $= +P_i$
- 若 $S_i = W$：翻转后变为 N，W 和减少 P_i，价值 $= -P_i$

设区间 $[L, R]$ 的原始 W 和为 sum_W，翻转 $[l, r]$ 后，新 W 和 = $sum_W + \sum_{i=l}^{r} val_i$。

要使总和最小，即求区间内的**最小子段和**（允许空区间，即不翻转，此时最小子段和为 0）。

3.4 部分分解法（30%）

思路：枚举所有可能的 $[l, r]$ 子区间，计算翻转后的 W 和，取最小值。复杂度 $O(Q \cdot n^2)$。

参考程序：

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;           // 使用 long long 防止 P_i 求和溢出
const int MAXN = 100005;        // 最大数据规模

char s[MAXN];                   // s[i] 表示第 i 个样本的负责人（'N' 或 'W'）
ll p[MAXN];                     // p[i] 表示第 i 个样本的标注难度系数

int main() {
    // 重定向文件输入输出
    freopen("change.in", "r", stdin);
    freopen("change.out", "w", stdout);

    int n, Q;
    cin >> n >> Q;

    // 读入每个样本的负责人和难度系数
    for (int i = 1; i <= n; i++)
        cin >> s[i] >> p[i];

```

```
    while (Q--) {
        int L, R;
        cin >> L >> R;          // 读入当前询问的区间范围

        // 计算区间 [L, R] 内原始 W 的难度系数之和
        ll orig_sum = 0;
        for (int i = L; i <= R; i++)
            if (s[i] == 'W') orig_sum += p[i];

        ll ans = orig_sum;      // 初始化为不执行操作时的结果

        // 枚举所有可能的翻转子区间 [l, r]
        for (int l = L; l <= R; l++) {
            ll sum_all = 0;     // 子区间内所有样本的难度系数之和
            ll sum_w = 0;       // 子区间内 W 样本的难度系数之和
            for (int r = l; r <= R; r++) {
                sum_all += p[r];
                if (s[r] == 'W') sum_w += p[r];

                // 翻转 [l, r] 后，W 的难度系数和变为：
                // orig_sum - sum_w + (sum_all - sum_w) = orig_sum + sum_all - 2*sum_w
                ans = min(ans, orig_sum + sum_all - 2 * sum_w);
            }
        }
        cout << ans << '\n';    // 输出当前询问的最优结果
    }
    return 0;
}
```

3.5 满分解法（100%）

思路：使用 Kadane 算法求最小子段和。遍历区间时维护当前子段和 `cur`，如果 `cur` > 0 则重置为 0（因为正数不会让总和更小），同时记录最小子段和。解决最小子段和的问题除了使用下文使用的 基于DP的$O(n)$ 解法，还可以采用类似于归并思路的分治解法，其时间复杂度是 $O(nlogn)$，此处就不给出解答程序了，请查询其他相关资料。另外，如果将Q的范围也扩大到10^5 ,则可能需要考虑使用线段树的方法解决，解决一次询问的时间复杂度为 $O(logn)$ ，已经超过了SACCC五级考查范围，有兴趣的同学可以自行研究。

由于 $Q \leq 100$，每个询问 $O(n)$ 处理，总复杂度 $O(nQ) \leq 10^7$。

参考程序:

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;           // 使用 long long 防止溢出
const int MAXN = 100005;        // 最大数据规模

char s[MAXN];                   // s[i] 表示第 i 个样本的负责人（'N' 或 'W'）
ll p[MAXN];                     // p[i] 表示第 i 个样本的标注难度系数

int main() {
    // 重定向文件输入输出
    freopen("change.in", "r", stdin);
    freopen("change.out", "w", stdout);

    int n, Q;
```

```
    cin >> n >> Q;

    // 读入每个样本的负责人和难度系数
    for (int i = 1; i <= n; i++)
        cin >> s[i] >> p[i];

    while (Q--) {
        int L, R;
        cin >> L >> R;          // 读入当前询问的区间范围

        // 计算区间 [L, R] 内原始 W 的难度系数之和
        ll orig_sum = 0;
        for (int i = L; i <= R; i++)
            if (s[i] == 'W') orig_sum += p[i];

        // Kadane 算法求最小子段和（允许空区间，即 min_sum 初始为 0）
        ll min_sum = 0;         // 记录遍历过程中的最小子段和
        ll cur = 0;             // 当前正在累积的子段和
        for (int i = L; i <= R; i++) {
            // 计算每个位置翻转后的"价值"：
            // N -> W 增加 P_i（正价值），W -> N 减少 P_i（负价值）
            ll val = (s[i] == 'N') ? p[i] : -p[i];

            cur += val;         // 累加当前子段
            if (cur > 0) cur = 0;   // 如果子段和为正，舍弃它（不会让总和更小）
            min_sum = min(min_sum, cur);  // 更新最小子段和
        }

        // 最优结果 = 原始 W 和 + 最小子段和（最小子段和 <= 0）
        cout << orig_sum + min_sum << '\n';
    }
    return 0;
}
```

4. 神仙跳舞

4.1 题目描述

哪吒和孙悟空各自训练了一个 AI 舞步生成模型，用于生成由左右步伐组成的舞蹈编排。哪吒的模型输出由字符串 s 表示，孙悟空的模型输出由字符串 t 表示，其中每个字符为 `0`（左步）或 `1`（右步）。两个模型的输出中均**不包含连续三个相同的步伐**（即没有 `000` 或 `111` 这样的子串）。

在正式演出前的**人工调优阶段**，哪吒可以对模型输出进行最多 K 次**人工修正**，每次修正操作如下：

- 选择 s 或 t 中的**一个位置**，将该位置的步伐翻转（`0` 变 `1`，`1` 变 `0`）。
- 但修正后，该位置的步伐必须与同一序列中**左右相邻的所有位置**不同。

调优完成后，哪吒将 s 和 t 中的某些步伐配对，形成"**同步点**"。配对必须满足以下所有条件：

- 每个配对由 s 中的一个位置 i 和 t 中的一个位置 j 组成。
- 只有满足 $s_i = t_j$ 的位置才能配对（步伐相同才能同步）。
- s 中的每个位置和 t 中的每个位置**最多只能参与一个配对**。
- 配对之间**不得交叉**。即对于两对 (i, j) 和 (i', j')，若 $i < i'$，则必须满足 $j < j'$。

请计算在**最优的人工修正和配对策略**下，最多能形成多少个同步点。

4.2 数据范围

- 对于 30% 的数据：$1 \le |s| \le 10$，$1 \le |t| \le 10$
- 对于 100% 的数据：$1 \le |s| \le 2 \times 10^3$，$1 \le |t| \le 2 \times 10^3$，$0 \le K \le |s| + |t|$

4.3 题目分析

本题包含两个子问题：

1. **翻转可行性分析**：哪些位置可以翻转？
2. **最大同步点**：翻转后求两个字符串的**最长公共子序列（LCS）**长度。

翻转可行性分析

一个位置 i 翻转后必须与所有相邻位置不同：

- **内部位置**（有左右邻居）：需要 $s[i] = s[i-1]$ 且 $s[i] = s[i+1]$，即三个连续相同。但题目保证原串无三个连续相同，因此**内部位置不可翻转。**
- **左端点**（$i = 0$）：需要 $s[0] = s[1]$ 才可翻转（翻转后变成与 $s[1]$ 不同）。
- **右端点**（$i = n-1$）：需要 $s[n-1] = s[n-2]$ 才可翻转。
- **长度为 1 的串**：无相邻位置，**总是可翻转。**

因此每个字符串最多 2 个可翻转位置（两个端点），每个最多翻转一次。

4.4 部分分解法（30%）

思路：暴力枚举所有可能的翻转组合（$2^{|s|+|t|}$ 种），对每种组合计算 LCS。适用于 $|s|, |t| \le 10$。

参考程序：

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

/**
 * 判断字符串 s 中位置 pos 是否可翻转
 * 翻转后该位置必须与所有相邻位置不同
 */
bool can_flip(const string& s, int pos) {
    int n = s.size();
    if (n == 1) return true;                // 长度为 1，无相邻位置，总是可翻转
    if (pos == 0) return s[0] == s[1];      // 左端点：需与右邻居相同，翻转后变不同
    if (pos == n - 1) return s[n - 1] == s[n - 2];  // 右端点：需与左邻居相同
    return false;                           // 内部位置不可翻转（原串无三个连续相同）
}

/**
 * 计算两个字符串的最长公共子序列（LCS）长度
 * 使用标准 DP，dp[i][j] 表示 a[0..i-1] 和 b[0..j-1] 的 LCS 长度
 */
int lcs(const string& a, const string& b) {
    int n = a.size(), m = b.size();
    vector<vector<int>> dp(n + 1, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++) {
            if (a[i - 1] == b[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;  // 字符相等，可以配对
            dp[i][j] = max({dp[i][j], dp[i - 1][j], dp[i][j - 1]});
```

```
            // 取三种情况的最大值：当前配对 / 跳过 a[i] / 跳过 b[j]
        }
    return dp[n][m];
}

int main() {
    // 重定向文件输入输出
    freopen("dance.in", "r", stdin);
    freopen("dance.out", "w", stdout);

    string s, t;
    int K;
    cin >> s >> t >> K;         // 读入两个字符串和最大修正次数

    int n = s.size(), m = t.size(), ans = 0;

    // 暴力枚举 s 的所有翻转子集（2^n 种可能）
    for (int ms = 0; ms < (1 << n); ms++) {
        string s2 = s;          // 复制原始字符串
        int fs = 0;             // 记录 s 中实际翻转的次数
        bool ok = true;

        // 检查并执行翻转
        for (int i = 0; i < n && ok; i++)
            if (ms >> i & 1) {                // 第 i 位需要翻转
                if (!can_flip(s, i)) ok = false;  // 不可翻转则跳过此方案
                else { s2[i] ^= 1; fs++; }    // 翻转字符（0<->1）
            }
        if (!ok || fs > K) continue;          // 非法或超出次数限制

        // 暴力枚举 t 的所有翻转子集（2^m 种可能）
        for (int mt = 0; mt < (1 << m); mt++) {
            string t2 = t;
            int ft = 0;
            ok = true;

            for (int i = 0; i < m && ok; i++)
                if (mt >> i & 1) {
                    if (!can_flip(t, i)) ok = false;
                    else { t2[i] ^= 1; ft++; }
                }
            if (!ok || fs + ft > K) continue;  // 总翻转次数不能超过 K

            // 计算翻转后两个字符串的 LCS，更新答案
            ans = max(ans, lcs(s2, t2));
        }
    }
    cout << ans << '\n';        // 输出最大同步点数量
    return 0;
}
```

4.5 满分解法（100%）

思路：由于每个字符串最多 2 个可翻转位置，枚举所有翻转组合（最多 $4 \times 4 = 16$ 种），对每种组合计算 LCS。

LCS 使用滚动数组优化空间，复杂度 $O(16 \times |s| \times |t|) \approx 16 \times 4 \times 10^6 = 6.4 \times 10^7$，1 秒内可通过。其实即使不适用滚动数组，使用空间也是满足要求的。另外，最多可能有4个可翻转位置，可以考虑采用二进制枚举 $0000 - 1111$，然后根据每个二进制位逐个和可以翻转的位置对应，记录每个改动位置，如果改动数量不超过K，则可以执行LCS，不管怎么样，执行完LCS都需要将每个改动位置复原成初始 0、1 字符。

参考程序:

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

/**
 * 计算两个字符串的最长公共子序列（LCS）长度
 * 使用滚动数组优化空间：dp[2][m+1] 替代 dp[n+1][m+1]
 */
int lcs(const string& a, const string& b) {
    int n = a.size(), m = b.size();
    // 只保留两行：当前行 (i & 1) 和上一行 ((i-1) & 1)
    vector<vector<int>> dp(2, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (a[i - 1] == b[j - 1])
                // 字符相等，继承左上角的值 + 1
                dp[i & 1][j] = dp[(i - 1) & 1][j - 1] + 1;
            else
                dp[i & 1][j] = 0;  // 先置 0，后面取 max
            // 取三种情况的最大值
            dp[i & 1][j] = max({dp[i & 1][j],
                                dp[(i - 1) & 1][j],
                                dp[i & 1][j - 1]});
        }
    }
    return dp[n & 1][m];        // 最后一行最后一列即为答案
}

int main() {
    // 重定向文件输入输出
    freopen("dance.in", "r", stdin);
    freopen("dance.out", "w", stdout);

    string s, t;
    int K;
    cin >> s >> t >> K;         // 读入两个字符串和最大修正次数

    int n = s.size(), m = t.size();

    // 找出 s 中所有可翻转的位置（最多 2 个：两端）
    vector<int> flips;
    if (n == 1) {
        flips.push_back(0);     // 长度为 1，唯一位置可翻转
    } else {
        if (s[0] == s[1]) flips.push_back(0);           // 左端点可翻转
        if (s[n - 1] == s[n - 2]) flips.push_back(n - 1);  // 右端点可翻转
    }

    // 找出 t 中所有可翻转的位置（最多 2 个：两端）
    vector<int> flipt;
```

```cpp
    if (m == 1) {
        flipt.push_back(0);
    } else {
        if (t[0] == t[1]) flipt.push_back(0);
        if (t[m - 1] == t[m - 2]) flipt.push_back(m - 1);
    }

    int ans = 0;

    // 枚举 s 的所有翻转组合（最多 2^2 = 4 种）
    for (int ms = 0; ms < (1 << flips.size()); ms++) {
        string s2 = s;
        int fs = 0;             // s 中翻转的次数
        for (int i = 0; i < (int)flips.size(); i++)
            if (ms >> i & 1) {
                s2[flips[i]] ^= 1;  // 翻转该位置
                fs++;
            }
        if (fs > K) continue;  // 超出总次数限制

        // 枚举 t 的所有翻转组合（最多 2^2 = 4 种）
        for (int mt = 0; mt < (1 << flipt.size()); mt++) {
            string t2 = t;
            int ft = 0;         // t 中翻转的次数
            for (int i = 0; i < (int)flipt.size(); i++)
                if (mt >> i & 1) {
                    t2[flipt[i]] ^= 1;
                    ft++;
                }
            if (fs + ft > K) continue;  // 总翻转次数不能超过 K

            // 计算翻转后两个字符串的 LCS，更新答案
            ans = max(ans, lcs(s2, t2));
        }
    }
    cout << ans << '\n';        // 输出最大同步点数量
    return 0;
}
```

附录：考点总结

题目	核心考点	部分分策略	满分策略	时间复杂度
Token 管理	时间衰减模拟、延迟更新	全局统一更新	每个模型独立延迟更新	$O(n+q)$
矩阵中的我	第 k 小、二分答案、计数	全量生成排序	二分 + 行内计数	$O(n\log(nm))$
数据集指派	区间操作、最小子段和	枚举子区间	Kadane 算法	$O(nQ)$
神仙跳舞	翻转可行性分析、LCS	暴力枚举翻转	端点枚举 + 滚动数组 LCS	$O(16\cdot nm)$