2025 CSP-S 第二轮认证题解报告

T1 [CSP-S 2025] 社团招新 / club（民间数据）

题意

有 n 个人要去 3 个部门工作，每个人对每个部门都有一个"喜欢程度"评分，而且每个人对这三个部门的评分都不一样（除了特殊性质）。

但是有个规定：每个部门的人数不能超过总人数的一半（$\lfloor n/2 \rfloor$）。

在满足人数限制的前提下，让所有人选择部门后，**总喜欢程度评分和**最大。

部分分析

设：初始“满意度值”为选择最高评分部门的分值，而“不满意值”为转去其他部门后减少的分值。

性质A ($a_{i,2} = a_{i,3}$) 简化：

- 部门 2 和部门 3 完全等价，满意度相同
- 初始分配时只需在部门 1 和部门 2 之间选择
- 调整时可以在部门 2 和部门 3 间自由分配，没有满意度损失

性质A ($a_{i,2} = a_{i,3}$)

```cpp
#define ll long long
#define endl "\n"
void solve() {
    int n;
    cin >> n;
    ll total = 0;
    vector<int> cnt(3, 0);

    // 性质A: ai2 = ai3, 所以只需要考虑部门1和部门2
    for (int i = 0; i < n; i++) {
        int x, y, z;
        cin >> x >> y >> z;
        // 性质A: y = z, 所以部门2和部门3完全等价

        if (x >= y)
            total += x,cnt[0]++;
        else
            total += y,cnt[1]++; // 部门2和部门3等价, 可以任意分配
    }
```

```
    // 调整：如果部门1超过限制，将部分人转移到部门2
    if (cnt[0] > n / 2) {
        // 不需要实际计算损失，因为部门2和部门3完全等价
        // 直接将多余的人分配到部门2即可，没有满意度损失
        cnt[1] += cnt[0] - n / 2;
        cnt[0] = n / 2;
    }

    // 检查部门2是否超过限制（部门3人数为0）
    if (cnt[1] > n / 2)
        // 同样没有损失，只需在部门2和部门3间平衡
        cnt[1] = n / 2;

    cout << total << endl;
}

int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int t;
    cin >> t;
    while (t--)
        solve();
    return 0;
}
```

性质B ($a_{i,3} = 0$)

- 部门 3 满意度始终为 0，通常不会被选择
- 初始分配在部门 1 和部门 2 之间进行
- 调整时，从部门 1 转移到部门 2 没有损失，从部门 2 转移到部门 3 有损失

```cpp
#define ll long long
#define endl "\n"

void solve() {
    int n;
    cin >> n;
    ll total = 0;
    vector<int> cnt(3, 0);

    // 性质B：ai3 = 0，所以部门3的满意度总是0
    for (int i = 0; i < n; i++) {
        int x, y, z;
        cin >> x >> y >> z;
        // 性质B：z = 0，忽略部门3

        if (x >= y)
            total += x,cnt[0]++;
        else
            total += y,cnt[1]++;
        // 部门3人数始终为0
    }
```

```
    // 调整：如果部门1超过限制
    if (cnt[0] > n / 2) {
        // 计算需要转移的人数
        int transfer = cnt[0] - n / 2;
        // 由于部门3满意度为0，转移损失就是原部门满意度
        // 但我们可以选择将部分人转移到部门2（如果y>0）
        // 实际上，由于我们初始已经选择了最优，这里不需要调整满意度
        cnt[0] = n / 2;
        cnt[1] += transfer;
    }

    // 检查部门2是否超过限制
    if (cnt[1] > n / 2) {
        // 部门2超过时，只能转移到部门3（满意度为0）
        // 这会带来满意度损失，但题目要求必须满足约束
        int transfer = cnt[1] - n / 2;
        cnt[1] = n / 2;
    }

    cout << total << endl;
}

int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int t;
    cin >> t;
    while (t--)
        solve();
    return 0;
}
```

正解

1. **（贪心）先让大家都去最喜欢的部门**
    - 每个人选择评分最高的部门
    - 统计每个部门有多少人
    - 计算这时候的总满意度
2. **检查有没有部门人数超标**
    - 如果所有部门人数都不超过 $n/2$，那就完美了，直接输出结果
    - 如果某个部门人数超标了，继续下一步

3. **（贪心）处理超标的部门**
   - 实际上最多只有一个部门会超标（因为两个部门都超标的话，总人数就超过 n 了）
   - 假设部门 A 超标了，需要从 A 转出 k 个人（k = 部门 A 现有人数 - $n/2$）
4. **计算转部门的"代价"**
   - 对于部门 A 中的每个人，计算转到其他两个部门的代价
   - **代价 = 原来部门的评分 - 新部门的评分**
   - 每个人选择**代价较小**的那个转部门方案

5. **选择代价最小的人转部门**
    - 把部门 A 所有人的转部门代价**从小到大排序**
    - 选择代价最小的 k 个人转走
    - 这些人的代价总和就是总损失
6. **结果**
    - 最终总满意度 = 初始总满意度 - 转部门的总代价损失

代码实现

```
#define ll long long
#define endl "\n"

// 存储每个人对三个部门的评分
struct node {
    int x, y, z; // 分别对应部门1、2、3的评分
};
void solve() {
    ll res = 0, cnt1 = 0, cnt2 = 0, cnt3 = 0; // 总满意度、三个部门选择的人数
    int n;
    cin >> n;
    vector<node> a(n + 10);
    vector<int> b(n + 10); // 记录每个人最初选择的部门
    // 读取每个人的评分
    for (int i = 1; i <= n; i++)
        cin >> a[i].x >> a[i].y >> a[i].z;
    // 第一步：贪心的每个人优先选择最大分值的部门
    for (int i = 1; i <= n; i++)
        if (a[i].x >= max(a[i].z, a[i].y))
            res += a[i].x, cnt1++, b[i] = 1;
        else if (a[i].y >= max(a[i].x, a[i].z))
            res += a[i].y, cnt2++, b[i] = 2;
        else
            res += a[i].z, cnt3++, b[i] = 3;
```

```
    // 第二步：处理超标的部门
    /* 贪心策略：根据每个部门最多可选择半数的条件，贪心的：将超过半数的部门，
    将该部门所有人的贡献(最大值)- 选择另外一个部门(次大值) = 最小损失。*/
    vector<int> ans;
    for (int i = 1; i <= n; i++)
        if (cnt1 > n / 2 && b[i] == 1)
            ans.emplace_back(a[i].x - max(a[i].y, a[i].z));
        else if (cnt2 > n / 2 && b[i] == 2)
            ans.emplace_back(a[i].y - max(a[i].x, a[i].z));
        else if (cnt3 > n / 2 && b[i] == 3)
            ans.emplace_back(a[i].z - max(a[i].x, a[i].y));
    // 第三步：按代价从小到大排序，选择代价最小的人转部门
    sort(ans.begin(), ans.end());
    // 第四步：计算超 n/2 个人的部门(可能是 A/B/C )，减去转部门的总代价
    int i = 0;
    while (cnt1 > n / 2)
        res -= ans[i++], cnt1--;
    while (cnt2 > n / 2)
        res -= ans[i++], cnt2--;
    while (cnt3 > n / 2)
        res -= ans[i++], cnt3--;
    cout << res << endl;
}
int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    int t;
    cin >> t;
    while (t--)
        solve();
    return 0;
}
```

通俗的理解

把这个过程想象成：

1. 先让大家各就各位，去自己最想去的部门
2. 如果发现某个部门人太多了，就让这个部门中"转部门代价最小"的一些人转到其他部门
3. "转部门代价"就是：原来部门的评分 - 新部门的评分，这个值越小，说明这个人转部门越不"心疼"
4. 优先让那些转部门"不太心疼"的人转走，这样总的"心疼程度"最小

> 这其实也是贪心反悔的过程

T2 [CSP-S 2025] 道路修复 / road（民间数据）

题意

有 n 座城市和 m 条双向道路，所有道路都被地震破坏。还有 k 个乡镇可以选择进行城市化改造。

目标：以最小费用让原有的 n 座城市相互连通。

可选方案：

- 修复原有道路，费用为 w_i
- 选择乡镇进行城市化改造（费用 c_j），然后建造乡镇到城市的道路（费用 $a_{j,i}$）

约束条件：

$1 \le n \le 10^4$，$1 \le m \le 10^6$，$0 \le k \le 10$，可以选择任意多个乡镇（包括 0 个）

正解：状态压缩 + 最小生成树

由于 $k \leq 10$，我们可以枚举所有乡镇的选择方案（共 2^k 种状态），对每种状态构建新图并求最小生成树。

思路

1. **预处理原图的最小生成树**
   - 使用 Kruskal 算法求出原图的最小生成树
   - 保留这 $n-1$ 条边作为候选边集
2. **枚举乡镇选择状态**
   - 对于每个状态 $state \in [0, 2^k)$
   - 计算选择乡镇的改造费用总和
   - 构建包含城市和所选乡镇的图

3. **构建新图并求最小生成树**
   - 节点：n 个城市 + 所选乡镇
   - 边：原图最小生成树边 + 所选乡镇到所有城市的边
   - 使用 Kruskal 算法求新图的最小生成树
4. **更新答案**
   - 总费用 = 乡镇改造费用 + 最小生成树边权和
   - 取所有状态中的最小值

时间复杂度分析

- 状态数：$O(2^k)$，$k \leq 10$ 时约为 1024
- 每状态边数：$O(n + n \cdot k) \approx 10^4 + 10^5$
- 排序复杂度：$O((n + nk)\log(n + nk))$
- 总复杂度：$O(2^k \cdot (n + nk)\log(n + nk))$，在数据范围内可接受

参考代码(1)

```
#define ll long long
const int N = 1e4 + 15, M = 2e6;
struct edge {
    int u, v, w;
    bool operator<(const edge &b) const { return w < b.w; }
} e[M];
int n, m, k, fa[N], tot, cost[15], vis[15];
int find(int x) { return x == fa[x] ? x : fa[x] = find(fa[x]); }

int main() {
    cin >> n >> m >> k;

    // 初始化并查集
    for (int i = 1; i <= n; i++) fa[i] = i;

    // 读入原图边
    for (int u, v, w, i = 1; i <= m; i++) {
        cin >> u >> v >> w;
        e[i] = (edge){u, v, w};
    }

    // 步骤1：求原图最小生成树
    sort(e + 1, e + 1 + m);
    tot = 0;
    for (int i = 1; i <= m; i++) {
        int fu = find(e[i].u), fv = find(e[i].v);
        if (fu != fv) {
            fa[fu] = fv;
            e[++tot] = e[i]; // 保留最小生成树边
            if (tot == n - 1)
                break;
        }
    }
```

参考代码(2)

```
// 读入乡镇信息
// 乡镇编号为 n+1 到 n+k
for (int i = 1; i <= k; i++) {
    cin >> cost[i]; // 乡镇改造费用
    for (int j = 1, w; j <= n; j++) {
        cin >> w;
        e[++tot] = {j, n + i, w}; // 乡镇到城市的边
    }
}

// 当前边集：原图最小联通的边(n-1) + 所有村庄与城市的连边(n*k)
// krusual 贪心边集排序
sort(e + 1, e + 1 + tot);

ll res = 1e18; // 初始化最小答案为极大值
```

参考代码(3)

```
    // 步骤2：枚举所有乡镇选择方案
    for (int state = 0; state < (1 << k); state++) {
        int num = 0;          // 选择的乡镇数量
        ll current_cost = 0; // 当前状态总费用

        // 计算乡镇改造费用
        for (int i = 1; i <= k; i++)
            if ((state >> (i - 1)) & 1) {
                num++, current_cost += cost[i];
                vis[i] = 1; // 标记该乡镇被选择
            } else
                vis[i] = 0;

        // 初始化并查集（城市 + 所选乡镇）
        for (int i = 1; i <= n + k; i++) fa[i] = i;

        int need = num + n - 1; // 联通 n 个城市所需边数
        int edges_used = 0;     // 已使用的边数

        // 步骤3：构建最小生成树
        for (int i = 1; i <= tot && edges_used < need; i++) {
            int u = e[i].u, v = e[i].v;

            // 如果边连接了未选择的乡镇，跳过
            if (v > n && !vis[v - n])
                continue;

            int fu = find(u), fv = find(v);
            if (fu != fv) {
                fa[fu] = fv, current_cost += e[i].w;
                edges_used++;
            }
        }
        // 步骤4：更新答案
        if (edges_used == need)
            res = min(res, current_cost);
    }
    cout << res << endl;
    return 0;
}
```

小结

1. **状态压缩枚举** $k \leq 10$ 的特点，枚举所有 2^k 种乡镇选择方案
2. **图构建技巧**：将乡镇视为特殊节点（编号 $n+1$ 到 $n+k$），统一处理
3. **边集预处理**：先求原图最小生成树，减少后续计算量

T3 [CSP-S 2025] 谐音替换 / replace（民间数据）

题意

给定 n 个字符串替换规则 $(s_{i,1}, s_{i,2})$，其中 $|s_{i,1}| = |s_{i,2}|$。对于 q 个询问，每个询问给出两个字符串 t_1 和 t_2，求有多少种将 t_1 的某个子串 y（等于某个 $s_{i,1}$）替换为对应的 $s_{i,2}$ 后能得到 t_2 的方法。两种方法不同当且仅当子串位置不同或使用的规则不同。

数据范围：$n, q \leq 2 \times 10^5$，$\sum |s_{i,1}| + |s_{i,2}| \leq 5 \times 10^6$，$\sum |t_{j,1}| + |t_{j,2}| \leq 5 \times 10^6$。

分析

思路：构建特殊的 AC 自动机，其中 Trie 树的边以字符对 (c_1, c_2) 为标识，同时匹配 t_1 和 t_2 的对应字符。

1. **特殊 Trie 树构建**：将每个规则 (s_1, s_2) 插入 Trie 树，边标识为 $(s_1[i], s_2[i])$ 组成的整数 $(s_1[i] -' a') \times 26 + (s_2[i] -' a')$。
2. **AC 自动机构建**：在 Trie 树上构建 Fail 指针和 Fail 树，用于快速跳转。

3. **查询处理**：对于每个查询 (t_1, t_2)：
   - 找到第一个不同位置 l 和最后一个不同位置 r
   - 在 AC 自动机上匹配字符对序列 $(t_1[i], t_2[i])$
   - 对于每个位置 $i \geq r-1$，在 Fail 树上找到满足深度条件的节点，统计规则数量

4. **优化**：使用倍增数组快速在 Fail 树上跳转，预处理每个节点的深度和祖先信息。

时间复杂度：$O(L_1 + L_2 + (n+q)\log n)$，其中 L_1, L_2 分别是规则和查询的总长度。

参考代码

```
const int N = 2e5 + 5, M = 5e6 + 5, MM = 25e5;

int n, q, tot, fail[MM], dep[MM];
char s1[M], s2[M];
int tag[MM], fa[MM][22];
unordered_map<int, int> trie[MM]; // 使用哈希表存储Trie树，键为字符对编码
vector<int> g[MM];                // Fail树

// 向Trie树中添加一个替换规则
void add() {
    int now = 0; // 从根节点开始
    for (int i = 1; s1[i]; i++) {
        // 将字符对(s1[i], s2[i])编码为一个整数
        int p = (s1[i] - 'a') * 26 + s2[i] - 'a';
        // 如果当前节点没有这个字符对对应的边，创建新节点
        if (!trie[now][p]) {
            trie[now][p] = ++tot;
            dep[tot] = dep[now] + 1; // 记录节点深度
        }
        now = trie[now][p]; // 移动到下一个节点
    }
    ++tag[now]; // 标记该节点为一个规则的终点
}
```

```cpp
// 构建AC自动机的Fail指针
void getfail() {
    queue<int> q;
    q.push(0); // 根节点的Fail指针指向自己
    while (!q.empty()) {
        int x = q.front();
        q.pop();
        // 遍历当前节点的所有子节点
        for (auto pp : trie[x]) {
            int p = pp.first;   // 字符对编码
            int to = pp.second; // 子节点编号
            int now = fail[x];  // 从父节点的Fail指针开始

            // 在Fail链上寻找匹配的节点
            while (now && trie[now].find(p) == trie[now].end())
                now = fail[now];

            // 如果找到匹配，设置子节点的Fail指针
            if (x && trie[now].find(p) != trie[now].end())
                now = trie[now][p];
            fail[to] = now;

            // 构建Fail树
            g[now].push_back(to);
            q.push(to);
        }
    }
}
```

```
// DFS遍历Fail树，预处理倍增数组和标记前缀和
void dfs(int x) {
    for (int to : g[x]) {
        tag[to] += tag[x]; // 标记前缀和
        fa[to][0] = x;     // 直接父节点

        // 预处理倍增数组
        for (int i = 1; i <= 21; i++)
            fa[to][i] = fa[fa[to][i - 1]][i - 1];

        dfs(to);
    }
}

int read(char *s) {
    char c;
    int i = 0;
    while ((c = getchar()) < 'a' || c > 'z') continue;
    s[++i] = c;
    while ((c = getchar()) >= 'a' && c <= 'z') s[++i] = c;
    s[i + 1] = 0;
    return i;
}

int main() {
    scanf("%d%d", &n, &q);

    // 读取所有替换规则并构建Trie树
    for (int i = 1; i <= n; i++)
        read(s1), read(s2), add();

    // 构建AC自动机
    getfail();
    dfs(0);
    // ...
```

```
// 处理每个查询
for (int i = 1; i <= q; i++) {
    int len = read(s1), len2 = read(s2);

    // 如果长度不同，直接输出0
    if (len != len2) {
        puts("0");
        continue;
    }

    // 找到第一个和最后一个不同的字符位置
    int l = len + 1, r = 0;
    for (int i = 1; i <= len; i++)
        if (s1[i] != s2[i]) {
            l = i; // 第一个不同位置
            break;
        }
    for (int i = len; i; i--)
        if (s1[i] != s2[i]) {
            r = i; // 最后一个不同位置
            break;
        }
    // ...
```

```
        int now = 0, ans = 0;
        // 在AC自动机上匹配字符对序列
        for (int i = 1; i <= len; i++) {
            int p = (s1[i] - 'a') * 26 + s2[i] - 'a';

            // 在Fail链上寻找匹配
            while (now && trie[now].find(p) == trie[now].end())
                now = fail[now];

            // 如果找到匹配，移动到对应节点
            if (trie[now].find(p) != trie[now].end())
                now = trie[now][p];

            // 当当前位置超过最后一个不同位置时，统计答案
            if (i >= r) {
                // 检查当前节点深度是否足够覆盖不同区间
                if (dep[now] < i - l + 1) continue;

                // 使用倍增找到深度刚好小于(i-l+1)的节点
                int x = now;
                for (int j = 21; j >= 0; j--)
                    if (dep[fa[x][j]] >= i - l + 1)
                        x = fa[x][j];
                x = fa[x][0]; // 移动到第一个深度不足的节点

                // 统计深度在[i-l+1, dep[now]]之间的规则数量
                ans += tag[now] - tag[x];
            }
        }

        printf("%d\n", ans);
    }
    return 0;
}
```

T4 [CSP-S 2025] 员工招聘 / employ（民间数据）

题意

有 n 个人应聘，公司希望录用至少 m 人。面试持续 n 天，每天面试一个人，顺序由排列 p 决定。每天面试题难度 $s_i \in \{0, 1\}$：$s_i = 0$ 表示题难，无人能做出，面试者被拒绝；$s_i = 1$ 表示题易，所有人都能做出，面试者被录用。但每个人有耐心上限 c_i，如果在他面试之前被拒绝或放弃的人数不少于 c_i，则他会放弃面试。求有多少种排列 p 使得最终录用人数至少为 m。

数据范围：$1 \le m \le n \le 500$，$0 \le c_i \le n$。

分析

观察数据范围，允许 $O(n^3)$ 算法，考虑动态规划。设状态 $dp_{i,j,k}$ 表示已经过了 i 天，有 j 个人被拒绝或放弃，且前 i 天中有 k 个人满足 $c_k \leq j$ 时的方案数（不考虑 $c_k > j$ 的人之间的差别）。定义 cnt_x 为满足 $c_i = x$ 的人数，pre_x 为 cnt_x 的前缀和。

转移时根据第 $i+1$ 天的难度 s_{i+1} 分类：

- 若 $s_{i+1}=0$：则当天面试者会被拒绝，因此 j 增加 1。枚举可能的 k'（满足 $k \le k' \le k+cnt_{j+1}$），计算新状态。转移系数涉及组合数和排列数，具体为：

$$\text{new} = dp_{i,j,k} \times \binom{i-k}{k'-k} \times \binom{cnt_{j+1}}{k'-k} \times (k'-k)!$$

  然后根据当天面试者的 c 值是否 $\le j+1$ 进一步转移：

  - 若 $\le j+1$，则 $dp_{i+1,j+1,k'+1} \leftarrow \text{new} \times (pre_{j+1}-k')$。
  - 若 $> j+1$，则 $dp_{i+1,j+1,k'} \leftarrow \text{new}$。

- 若 $s_{i+1}=1$：当天面试者可能被录用或放弃。先处理 $c > j$ 的情况，直接转移到 $dp_{i+1,j,k}$；对于 $c \le j$ 的情况，类似 $s_{i+1}=0$ 处理，但仅转移到 $dp_{i+1,j+1,k'+1}$，系数为 $\text{new} \times (pre_j - k)$。

最终答案需要统计所有 $j \leq n - m$ 的状态，并乘上 $c_k > j$ 的人的排列方案数，即：

$$\text{ans} = \sum_{j=0}^{n-m} dp_{j,pre_j} \times (n - pre_j)!$$

时间复杂度：$O(n^3)$，空间优化后使用二维数组实现。

参考代码

```
const int mod = 998244353, N = 505;
long long f[N], g[N], inv[N];          // f:阶乘，g:阶乘逆元，inv:逆元
int n, m, cnt[N], pre[N], dp[N][N]; // cnt[c]: 耐心上限为 c 的人数，pre: cnt 的前缀和
char s[N];                             // 每天的面试题难度
// ...
```

```
int main() {
    scanf("%d %d", &n, &m);
    // 预处理阶乘、逆元和阶乘逆元
    f[0] = f[1] = g[0] = g[1] = inv[1] = 1;
    for (int i = 2; i <= n; i++) {
        f[i] = f[i - 1] * i % mod;                      // 计算 i 的阶乘
        inv[i] = inv[mod % i] * (mod - mod / i) % mod; // 线性求逆元
        g[i] = g[i - 1] * inv[i] % mod;                 // 计算阶乘逆元
    }

    scanf("%s", s + 1);
    for (int i = 1, c; i <= n; i++) {
        scanf("%d", &c);
        cnt[c]++; // 统计每个耐心上限的人数
    }

    // 计算前缀和：pre[i] = 耐心上限<=i的总人数
    pre[0] = cnt[0];
    for (int i = 1; i <= n; i++)
        pre[i] = pre[i - 1] + cnt[i];

    // dp初始化：0 天，0 人被拒，0 个c<=0的人
    dp[0][0] = 1;
```

```
// 动态规划：按天数转移
for (int i = 0; i < n; i++) // 已进行 i 天
{
    for (int j = i; j >= 0; j--) // 已被拒绝或放弃的人数（倒序）
    {
        for (int k = pre[j]; k >= 0; k--) // 前 i 天中c<=j的人数（倒序）
        {
            int now = dp[j][k]; // 当前状态方案数
            dp[j][k] = 0;       // 清空当前状态（滚动数组）

            if (s[i + 1] == '0') // 第 i+1 天题目难，必被拒
            {
                // 枚举可能的 k2：新状态中 c<=j+1 的人数
                for (int k2 = max(k, i - (n - pre[j + 1]));
                     k2 <= k + cnt[j + 1] && k2 <= i; k2++) {
                    const int tot = i - k;      // c>j 的人数
                    const int ths = k2 - k;     // 新增的 c=j+1 的人数
                    const int num = cnt[j + 1]; // 耐心上限 =j+1 的总人数

                    // 计算转移系数：组合数 × 排列数
                    const long long x = f[tot] * g[ths] % mod * g[tot - ths] % mod * f[num] % mod * g[num - ths] % mod;

                    // 第 i+1 天的人 c>j+1：转移到 dp[j+1][k2]
                    if (j + 1 != n && pre[j + 1] - k2 != n - i)
                        dp[j + 1][k2] = (dp[j + 1][k2] + x * now) % mod;

                    // 第 i+1 天的人 c<=j+1：转移到 dp[j+1][k2+1]
                    dp[j + 1][k2 + 1] = (dp[j + 1][k2 + 1] + x * now % mod * (pre[j + 1] - k2)) % mod;
                }
            }
            // else ...
```

```
                else // 第 i+1 天题目简单，可能被录用
                {
                    // 枚举可能的 k2：新状态中 c<=j+1 的人数
                    for (int k2 = max(k, i - (n - pre[j + 1]));
                         k2 <= k + cnt[j + 1] && k2 <= i; k2++) {
                        const int tot = i - k;      // c>j 的人数
                        const int ths = k2 - k;     // 新增的 c=j+1 的人数
                        const int num = cnt[j + 1]; // 耐心上限 =j+1 的总人数

                        // 计算转移系数
                        const long long x = f[tot] * g[ths] % mod *
                                            g[tot - ths] % mod * f[num] % mod *
                                            g[num - ths] % mod;

                        // 第 i+1 天的人 c<=j：被录用，但耐心不足会放弃，转移到 dp[j+1][k2+1]
                        dp[j + 1][k2 + 1] =
                            (dp[j + 1][k2 + 1] + x * now % mod * (pre[j] - k)) %
                            mod;
                    }

                    // 第 i+1 天的人 c>j：被录用且不会放弃，直接转移到 dp[j][k]
                    if (pre[j] - k != n - i)
                        dp[j][k] = now;
                }
            }
        }
    }

    // 统计答案：录用人数>=m，即拒绝人数<=n-m
    int ans = 0;
    for (int j = 0; j <= n - m; j++) {
        const int k = pre[j]; // 所有c<=j的人数
        // 乘以剩余人的排列数（c>j的人可以任意排列）
        ans = (ans + dp[j][k] * f[n - k]) % mod;
    }
    printf("%d", ans);
    return 0;
}
```

2025 CSP-S 第二轮认证 End！