2025 CSP-J 第二轮认证题解报告

T1 [CSP-J 2025] 拼数 / number

题意

给定一个字符串 s（$1 \le |s| \le 10^6$），仅包含小写英文字母和数字，且**至少包含一个 $1 \sim 9$ 的数字**。从字符串中选取若干数字字符（每个字符只能使用一次），按任意顺序拼接成一个正整数，求能拼成的最大正整数。

关键点：

- 只能使用数字字符（$0 \sim 9$）
- 每个字符只能使用一次
- 结果必须是正整数（不能有前导 0）
- 目标是数值最大化

分析

思路

贪心 + 桶：将数字字符按从大到小排序后直接拼接。

证明：

1. **数值最大化原则**：高位放置较大的数字能获得更大的数值
2. **无前导 0 保证**：题目保证至少有一个 $1 \sim 9$ 的数字，所以排序后第一个字符不会是 0
3. **贪心正确性**：对于任意两个数字 a 和 b，如果 $a > b$，那么 $ab > ba$（按数值比较）

时间复杂度

- 统计数字出现次数：$O(|s|)$
- 输出结果：$O(|s|)$
- 总复杂度：$O(|s|)$，满足 $|s| \leq 10^6$ 的要求

空间复杂度

- 使用大小为 10 的计数数组：$O(1)$

参考代码

```cpp
const int N = 10;          // 数字0-9共10种
int num[N];                // 桶数组，记录每个数字出现的次数

int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    string s;
    cin >> s;

    // 统计字符串中每个数字字符的出现次数
    for (int i = 0; i < s.length(); i++)
        if (s[i] >= '0' && s[i] <= '9') {
            num[s[i] - '0']++;  // 将字符转换为数字下标并计数
        }

    // 从数字9到0依次输出，确保数值最大
    for (int i = 9; i >= 0; i--)
        // 输出当前数字的所有出现次数
        while (num[i]--)
            cout << i;
    return 0;
}
```

T2 [CSP-J 2025] 座位 / seat

题意

- **考场规模**：$n \times m$ 名考生（$1 \leq n \leq 10, 1 \leq m \leq 10$）
- **成绩特性**：所有考生第一轮成绩**互不相同**
- **座位分配**：按成绩**从高到低**蛇形排列
  - 第1列：从上到下（第1行→第n行）
  - 第2列：从下到上（第n行→第1行）
  - 第3列：从上到下，依此类推
- **任务**：给定小R的成绩（在输入的第1个位置），确定他的座位（列，行）

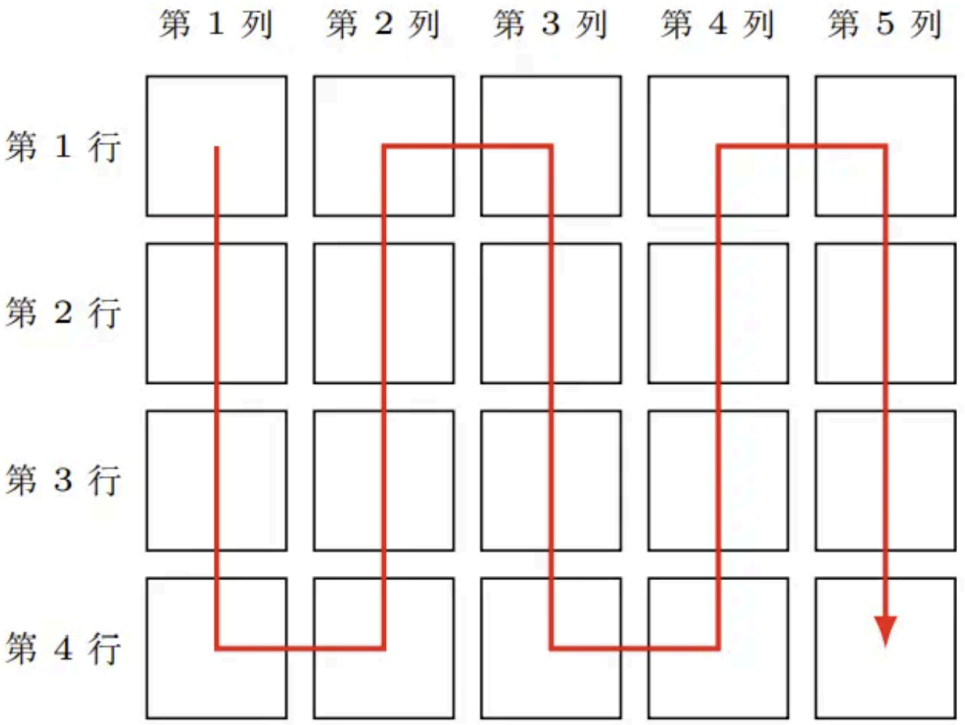
样例 1 解释：

成绩排序：100 > 99 > 98 > 97

座位分配：

- 第 1 列第 1 行：100
- 第 1 列第 2 行：99
- 第 2 列第 2 行：98
- 第 2 列第 1 行：97

  小 R 成绩 99 → 座位 (1,2)



分析

排序+数学计算

1. **确定排名**：将所有成绩排序，找到小R成绩的排名位置
2. **计算座位**：根据排名和蛇形规则计算具体的行和列

网格计算原理

设排名为 $rank$（从 1 开始），则：

- **列号**：$col = \lceil \frac{rank}{n} \rceil = \frac{rank-1}{n} + 1$
- **在列中的偏移**：$offset = (rank - 1) \bmod n$
- **行号**：
  - 奇数列：$row = offset + 1$（从上到下）
  - 偶数列：$row = n - offset$（从下到上）

时间复杂度

- 排序：$O(nm\log(nm))$，最大 $O(100\log 100)$
- 查找排名：$O(nm)$
- 总体：$O(nm\log(nm))$，完全可行

空间复杂度

- 存储成绩数组：$O(nm)$

暴力模拟 $O(nm)$

```cpp
int main() {
    int n, m;
    cin >> n >> m;
    vector<int> scores(n * m);
    for (int i = 0; i < n * m; i++)
        cin >> scores[i];

    int r_score = scores[0]; // 小R的成绩

    // 模拟构造整个座位表
    vector<vector<int>> seat(n + 1, vector<int>(m + 1));
    vector<int> sorted = scores;
    sort(sorted.rbegin(), sorted.rend());

    int idx = 0;
    for (int col = 1; col <= m; col++)
        if (col % 2 == 1)  // 奇数列：从上到下
            for (int row = 1; row <= n; row++)
                seat[row][col] = sorted[idx++];

         else  // 偶数列：从下到上
            for (int row = n; row >= 1; row--)
                seat[row][col] = sorted[idx++];
    // 查找小R的位置
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= m; j++)
            if (seat[i][j] == r_score) {
                cout << j << " " << i << endl;
                return 0;
            }
    return 0;
}
```

优化代码

```cpp
const int N = 110;  // 最大可能人数：10*10=100

int n, m, _ansi, _ansj;  // 行列变量（实际未使用原变量名）
int a[N], row;           // 成绩数组和行号变量

int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    cin >> n >> m;

    // 读入所有考生成绩
    for (int i = 1; i <= n * m; i++)
        cin >> a[i];

    int rank = 1, RScore = a[1];  // 小R的成绩在第一个位置

    // 将成绩从高到低排序
    sort(a + 1, a + 1 + n * m, greater<int>());
```

```
    // 在排序后的数组中查找小R成绩的排名
    for (int i = 1; i <= n * m; i++)
        if (RScore == a[i]) {
            rank = i;  // 记录排名（从1开始）
            break;
        }

    // 计算列号：每列n个人，列号从1开始
    int col = (rank - 1) / n + 1;
    // 计算在列中的偏移量（0-based）
    int offset = (rank - 1) % n;

    // 根据列号奇偶性确定行号
    if (col % 2 == 1)          // 奇数列：从上到下分配
        row = offset + 1;      // 偏移0→第1行，偏移1→第2行...
    else                       // 偶数列：从下到上分配
        row = n - offset;      // 偏移0→第n行，偏移1→第n-1行...

    // 输出列和行（注意题目要求顺序）
    cout << col << " " << row << endl;
    return 0;
}
```

T3 [CSP-J 2025] 异或和 / xor

题意

- **序列长度**：n（$1 \le n \le 5 \times 10^5$）
- **目标值**：k（$0 \le k < 2^{20}$）
- **任务**：从序列中选择尽可能多的**不相交**区间，使得每个区间的异或和都等于 k
- **不相交定义**：任意两个区间不能共享任何下标

> **样例1解释：**
> 序列：[2, 1, 0, 3], k = 2
> 选择区间：[1,1] (异或和 =2) 和 [2,4] ($1 \oplus 0 \oplus 3 = 2$)
> 最多可以选择 2 个不相交区间

> **样例3解释：**
> 序列：[2, 1, 0, 3], k=0
> 只能选择1个区间，如 [3,3] (异或和 =0)
> 注意不能同时选 [3,3] 和 [1,4]，因为下标 3 重叠

分析

动态规划 + 前缀异或和优化

1. **前缀异或和性质：**
   - 定义 $p_i = a_1 \oplus a_2 \oplus \cdots \oplus a_i$
   - 区间 $[l, r]$ 的异或和 = $p_r \oplus p_{l-1}$
   - 区间异或和为 $k \Leftrightarrow p_{l-1} = p_r \oplus k$

2. **动态规划状态**：
   - dp_i：前 i 个元素能选择的最大不相交区间数
   - 转移：$dp_i = \max(dp_{i-1}, best[p_i \oplus k] + 1)$
     - dp_{i-1}：不选择以 i 结尾的区间
     - $best[p_i \oplus k] + 1$：选择以 i 结尾的区间

3. **best数组优化**：
   - $best[v]$：记录前缀异或值为 v 时的最大 dp 值
   - 通过 $best$ 数组在 $O(1)$ 时间内找到最优转移

时间复杂度

- 遍历序列：$O(n)$
- 异或操作：$O(1)$
- 总复杂度：$O(n)$，满足 $n \leq 5 \times 10^5$ 的要求

空间复杂度

- $best$ 数组：$O(2^{20}) = O(1.05 \times 10^6)$
- 其他变量：$O(1)$

部分分策略

贪心法（特殊性质B：$a_i \leq 1$）

```cpp
int main() {
    int n, k;
    cin >> n >> k;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];
    // 当a_i只能是0或1时的特判
    if (k == 0) {
        // 统计连续的0的段数
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (a[i] == 0) {
                count++;
                while (i < n && a[i] == 0) i++;
            }
        }
        cout << count << endl;
    } else if (k == 1) {
        // 只能选择单个1的区间
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (a[i] == 1) count++;
        }
        cout << count << endl;
    }

    return 0;
}
```

正解代码

```cpp
// 算法: [动态规划] [前缀异或和] [状态优化] [不相交区间]
// 异或值最大范围为2^20
const int N = 1 << 20, INF = 0x3f3f3f3f;

int best[N]; // best[i]: 记录前缀异或值为i时的最大区间数量
int n, k;

int main() {
    ios::sync_with_stdio(false), cin.tie(nullptr);
    cin >> n >> k;

    // 初始化best数组为负无穷，表示不可达状态
    for (int i = 0; i < N; i++)
        best[i] = -INF;

    // 初始状态: 前缀异或和为0时，可以选择0个区间
    best[0] = 0;

    int x = 0,      // 当前前缀异或和
        dp = 0;     // 到当前位置为止能选择的最大区间数量
```

```
    for (int i = 1, a; i <= n; i++) {
        cin >> a;
        x ^= a; // 更新前缀异或和

        /*
         * 关键思路：查找是否存在前缀异或值等于 x^k 的位置
         * 如果存在，说明可以形成一个以当前位置为结尾的区间，其异或和为k
         * 因为：p[r] ⊕ p[l-1] = k  => p[l-1] = p[r] ⊕ k
         */
        int t = best[x ^ k];

        // nd表示考虑当前位置时的最大区间数量
        // 初始值为不选择当前区间的情况（沿用上一个位置的dp值）
        int nd = dp;

        // 如果找到了匹配的前缀异或值，考虑选择以当前位置结尾的区间
        if (t > -INF)
            nd = max(nd, t + 1); // 在t的基础上增加一个区间

        dp = nd; // 更新前缀最大dp值

        /*
         * 更新best数组：记录当前前缀异或值x对应的最大区间数量
         * 注意：先使用best数组进行计算，再更新，避免自引用
         */
        if (best[x] < dp)
            best[x] = dp;
    }

    cout << dp << endl;
    return 0;
}
```

算法正确性证明

关键点1：不相交性保证

- 在计算 dp_i 时，使用的 $best[p_i \oplus k]$ 对应的是在 i 之前结束的区间
- 更新 $best[x]$ 在计算完 dp_i 之后，确保不会引用到包含当前区间的状态

关键点2：最优子结构

- 每个位置 i 的最优解要么继承 $i-1$ 的最优解，要么在某个 $j < i$ 的基础上增加一个区间 $[j+1, i]$
- 通过 $best$ 数组快速找到所有可能的 j 中的最优值

关键点3：完备性

- 考虑了所有可能的不相交区间组合
- DP 覆盖了所有前缀的最优解

T4 [CSP-J 2025] 多边形 / polygon

题意

- **木棍数量**：n（$3 \le n \le 5000$）
- **木棍长度**：a_i（$1 \le a_i \le 5000$）
- **多边形条件**：选择 $m \ge 3$ 根木棍，满足 $\sum_{i=1}^{m} l_i > 2 \times \max_{i=1}^{m} l_i$
- **任务**：计算能组成多边形的选择方案数（对 998244353 取模）

> **多边形条件解释**：
> 这实际上是多边形存在的充要条件：最长边小于其他边长度之和
> 因为 $\sum l_i > 2 \times \max l_i \Leftrightarrow \max l_i < \sum l_i - \max l_i$

> **样例 1 解释**：
> 从 5 根木棍 $[1, 2, 3, 4, 5]$ 中能组成多边形的9种方案已列出
> 注意：方案不同指选择的木棍下标集合不同

分析

思路

正难则反 + 动态规划

1. **总方案计算**：所有非空子集数为 $2^n - 1$
2. **排除无效方案**：
    - 木棍数少于 3：$n + C(n, 2)$
    - 木棍数 ≥ 3 但不能组成多边形的情况
3. **关键观察**：不能组成多边形 $\Leftrightarrow$ 最长木棍长度 $\geq$ 其他木棍长度之和

DP

状态定义：

- 对木棍按长度排序后，设 $f[j]$ 表示能组成长度和为 j 的方案数
- 使用 [滚动数组] 优化空间

流程：

1. 将木棍按长度排序
2. 遍历每根木棍，假设它作为最长木棍
3. 统计在它之前的木棍中，长度和 ≤ 当前木棍长度的方案数
4. 这些方案加上当前木棍后不满足多边形条件
5. 更新DP数组，加入当前木棍的影响

时间复杂度

- 排序：$O(n \log n)$
- 动态规划：$O(n \times V)$，其中 $V = 5000$
- 总复杂度：$O(nV)$，满足 $n \leq 5000$ 的要求

空间复杂度

- DP 数组：$O(V)$
- 其他：$O(n)$

暴力枚举法（$n \leq 20$）

```cpp
// [算法: 暴力枚举]
const int MOD = 998244353;

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

    long long ans = 0;
    // 枚举所有非空子集
    for (int mask = 1; mask < (1 << n); mask++) {
        vector<int> selected;
        for (int i = 0; i < n; i++) {
            if (mask >> i & 1) selected.push_back(a[i]);
        }

        if (selected.size() < 3) continue;

        // 检查多边形条件
        int max_len = *max_element(selected.begin(), selected.end());
        int sum = accumulate(selected.begin(), selected.end(), 0);

        if (sum > 2 * max_len) {
            ans = (ans + 1) % MOD;
        }
    }

    cout << ans << endl;
    return 0;
}
```

参考代码

```
// 算法: [动态规划] [组合数学] [正难则反] [排序优化]
#define ll long long

// 模数和数组大小定义
const ll p = 998244353;
const int N = 5005, M = 5005;

int n,a[N];          // 存储木棍长度
ll ans, f[2][M];       // DP数组，f[0]和f[1]交替使用，f[i][j]表示长度为j的方案数

// 模运算函数，确保结果在[0, p-1]范围内
ll mod(ll x) { return (x % p + p) % p; }

// 快速幂函数，计算a^b mod p
ll qpow(ll a, ll b) {
    ll res = 1;
    while (b) {
        if (b & 1)
            res = mod(res * a);
        a = mod(a * a);
        b >>= 1;
    }
    return res;
}
```

```cpp
int main() {
    cin >> n;
    // 读入木棍长度
    for (int i = 1; i <= n; i++)
        cin >> a[i];

    // 将木棍按长度排序，便于确定最长边
    sort(a + 1, a + 1 + n);
    int v = a[n]; // 最大木棍长度

    // 动态规划计算不能组成多边形的方案数
    for (int i = 1; i <= n; i++) {
        // 统计以当前木棍为最长边时，不能组成多边形的方案数
        // 条件：其他木棍长度之和 <= 当前木棍长度
        for (int j = 1; j <= a[i]; j++)
            ans = mod(ans + f[1][j]);

        // 更新DP数组：考虑加入当前木棍
        // 使用倒序更新避免重复计数
        for (int j = v - a[i]; j >= 0; j--)
            f[1][j + a[i]] = mod(f[1][j + a[i]] + f[1][j] + f[0][j]);

        // 记录单根木棍的情况
        f[0][a[i]]++;
    }

    // 计算最终答案：总方案数 = 2^n - 1 (排除空集)
    // 减去：单根木棍情况(n)，两根木棍情况(C(n,2))，不能组成多边形的方案数(ans)
    ll total = qpow(2, n) - 1, single = n;  // 总非空子集数，单根木棍的情况数
    ll pair = mod(n * (n - 1) * qpow(2, p - 2)); // C(n,2) = n*(n-1)/2
    ll result = mod(total - single - pair - ans);

    cout << result << endl;
    return 0;
}
```

算法正确性证明

关键点1：条件等价转换

- 多边形条件：$\sum l_i > 2 \times \max l_i$
- 等价于：$\max l_i < \sum l_i - \max l_i$
- 即最长边小于其他边之和

关键点2：不重复计数

- 对木棍排序后，按顺序处理确保当前木棍是最长的
- 倒序更新DP数组避免同一木棍被多次使用
- 统计时只考虑严格小于当前木棍的情况

算法正确性证明

关键点3：完备性

- 考虑了所有可能的子集选择
- 正确排除了木棍数少于3的情况
- 准确统计了不满足多边形条件的方案

2025 CSP-J 第二轮认证 end！