目录

一、 [CSP-J 2025] 拼数 / number

题目描述

小 R 正在学习字符串处理。小 X 给了小 R 一个字符串 s，其中 s 仅包含小写英文字母及数字，且包含至少一个 $1 \sim 9$ 中的数字。小 X 希望小 R 使用 s 中的任意多个数字，按任意顺序拼成一个正整数。注意：小 R 可以选择 s 中相同的数字，但每个数字只能使用一次。例如，若 s 为 `1a01b`，则小 R 可以同时选择第 $1,3,4$ 个字符，分别为 $1,0,1$，拼成正整数 101 或 110；但小 R 不能拼成正整数 111，因为 s 仅包含两个数字 1。小 R 想知道，在他所有能拼成的正整数中，最大的是多少。你需要帮助小 R 求出他能拼成的正整数的最大值。

输入输出格式

输入：第一行，一个字符串 s，表示小 X 给小 R 的字符串。

输出：一行，一个正整数，表示小 R 能拼成的正整数的最大值。

输入示例	输出示例
5	5
290es1q0	92100

样例解释

样例1：字符串 $s =$ "5"，只有一个数字5，所以最大正整数就是5。

样例2：字符串 $s =$ "290es1q0"，包含数字 $2,9,0,1,0$。将这些数字从大到小排列得到 $9,2,1,0,0$，拼成的最大正整数为92100。

算法分析

1. 问题转化

   从字符串中提取所有数字字符，然后将这些数字按照从大到小的顺序排列，这样拼接起来的数字就是最大的正整数。

2. 关键步骤

   （1） 遍历字符串，统计每个数字出现的次数

   ```
   for (int i = 0; i < s.size(); i++) {
       if (s[i] >= '0' && s[i] <= '9') a[s[i] - '0']++;
   }
   ```

   （2） 从数字9到数字0，依次将每个数字拼接到结果字符串中

   ```
   for (int i = 9; i >= 0; i--) {
       while (a[i]-- > 0) res += i + '0';
   }
   ```

   （3） 输出结果字符串

3. 算法正确性证明

（1） 贪心策略的正确性

要得到最大的正整数，首先就是位数要尽可能多，意味着每一个数字都要用到，其次应该让高位数字尽可能大。因此，将数字从大到小排列是最优策略。

参考实现

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

using u64 = unsigned long long;
using i64 = long long;

// std::mt19937_64 rng {std::chrono::steady_clock::now().
    time_since_epoch().count()};
#define int long long
#define endl "\n"
constexpr i64 inf = 1e18;

void slu() {
    string s;
    cin >> s;
    vector<int> a(10, 0);
    for (int i = 0; i < s.size(); i++) {
        if (s[i] >= '0' && s[i] <= '9') a[s[i] - '0']++;
    }
    string res = "";
    for (int i = 9; i >= 0; i--) {
        while (a[i]-- > 0) res += i + '0';
    }
    cout << res << endl;
}

signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    int t = 1;
    // cin >> t;

    while (t--) slu();
    return 0;
}
```

拓展思考

- 如果题目修改为输出最小正整数，该如何实现？

二、 [CSP-J 2025] 座位 / seat

题目描述

CSP-J 2025 第二轮正在进行。小 R 所在的考场共有 $n \times m$ 名考生，其中所有考生的 CSP-J 2025 第一轮成绩互不相同。所有 $n \times m$ 名考生将按照 CSP-J 2025 第一轮的成绩，由高到低蛇形分配座位，排列成 n **行** m **列**。具体地，设小 R 所在的考场的所有考生的成绩从高到低分别为 $s_1 > s_2 > \cdots > s_{n\times m}$，则成绩为 s_1 的考生的座位为第 1 **列**第 1 **行**，成绩为 s_2 的考生的座位为第 1 **列**第 2 **行**，...，成绩为 s_n 的考生的座位为第 1 **列**第 n **行**，成绩为 s_{n+1} 的考生的座位为第 2 **列**第 n **行**，...，成绩为 s_{2n} 的考生的座位为第 2 **列**第 1 **行**，成绩为 s_{2n+1} 的考生的座位为第 3 **列**第 1 **行**，以此类推。

例如，若 $n = 4, m = 5$，则所有 $4 \times 5 = 20$ 名考生将按照 CSP-J 2025 第一轮成绩从高到低的顺序，根据下图中的箭头顺序分配座位。

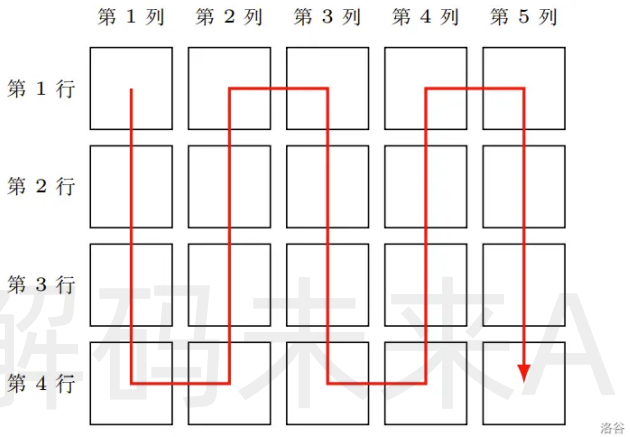

图 1: 蛇形排列示意图（$n = 4, m = 5$）

输入输出格式

输入：第一行，两个正整数 n, m，分别表示考场座位的**行数**与**列数**。

第二行，$n \times m$ 个正整数 $a_1, a_2, \ldots, a_{n\times m}$，表示所有考生 CSP-J 2025 第一轮的成绩，其中 a_1 为小 R 的成绩。

输出：一行两个正整数 c, r，表示小 R 的座位为第 c **列**第 r **行**。

输入示例	输出示例
2 2 99 100 97 98	1 2
2 2 98 99 100 97	2 2
3 3 94 95 96 97 98 99 100 93 92	3 1

样例解释

样例1：成绩从高到低排序为：100, 99, 98, 97。
座位分配过程如下图所示：

100	97
99	98

图 2: 样例1座位分配图（n=2, m=2）

小 R 的成绩为99，位于第1列第2行。
样例2：成绩从高到低排序为：100, 99, 98, 97。
座位分配与样例1相同，分配图如下：

100	97
99	98

图 3: 样例2座位分配图（n=2, m=2）

但小 R 的成绩为98，因此座位为第2列第2行。

算法分析

1. 问题转化
   本题的核心在于按照蛇形顺序填充矩阵，并找到目标成绩的坐标。首先，我们需要将所有考生的成绩从大到小排序。

2. 规律观察
   观察题目给出的蛇形排列方式，我们可以发现**列的奇偶性**决定了行的填充方向：

   （1） **奇数列（第1, 3, 5...列）**：成绩从上到下（行号 $1 \to n$）依次排列。

   （2） **偶数列（第2, 4, 6...列）**：成绩从下到上（行号 $n \to 1$）依次排列。

3. 算法步骤

   （1） 记录小 R 的原始成绩（输入数组的第一个数）。

   （2） 对成绩数组进行降序排序。

   （3） 使用一个全局指针 ‘idx‘ 指向排序后的成绩数组。

   （4） 外层循环枚举列号 c 从 1 到 m：

   - 若 c 为奇数，内层循环行号 r 从 1 到 n；

- 若 c 为偶数，内层循环行号 r 从 n 到 1。

（5） 在填充过程中，比较当前填入的成绩是否等于小 R 的成绩。若相等，立即输出当前的 (c,r) 并结束程序。

参考代码

```cpp
#include "bits/stdc++.h"
using namespace std;
using u32 = unsigned;
using i32 = int;
using u64 = unsigned long long;
using i64 = long long;
using u128 = unsigned __int128;
using i128 = __int128;

#define int long long
#define endl "\n"

constexpr i64 inf = 1e18;

void slu() {
    int n, m;
    cin >> n >> m;
    vector<int> a(n * m);
    for (auto& x : a) cin >> x;
    int aim = a[0];
    sort(a.begin(), a.end(), greater());
    vector<vector<int>> M(n, vector<int>(m, 0));

    int x = 0, y = 0;
    int T = 0;
    int cur = 0;
    while (x < n && y < m) {
        M[x][y] = a[cur++];
        if (M[x][y] == aim) {
            cout << y + 1 << " " << x + 1;
            return;
        }
        if (T < n - 1) x++;
        else if (T == n - 1) y++;
        else if (T != 2 * n - 1) x--;
        else y++;

        T = (T + 1) % (2 * n);
    }
}

signed main() {
    ios_base::sync_with_stdio(false);
```

```
    cin.tie(nullptr);
    cout.tie(nullptr);

    int t = 1;
    // cin >> t;

    while (t--) slu();
    return 0;
}
```

拓展思考

- 数学推导（$O(1)$ 解法）：假设小 R 的成绩在排序后排在第 k 名（下标从 0 开始）。
  1. 所在的列号：$c = k/n + 1$。
  2. 所在的行号：
     - 若 c 为奇数，行号 $r = (k\%n) + 1$；
     - 若 c 为偶数，行号 $r = n - (k\%n)$。

  这种方法无需模拟整个矩阵，效率更高。

```
int n, m;
cin >> n >> m;
vector<int> a(n *m);
for (auto &x : a) cin >> x;

int target = a[0];
sort(a.begin(), a.end(), greater<int>());

int idx = -1;
for (int i = 0; i < n * m; i++) {
    if (a[i] == target) {
        idx = i;
        break;
    }
}

int c = idx / n + 1;
int r;
if (c % 2 != 0) {
    r = (idx % n) + 1;
} else {
    r = n - (idx % n);
}

cout << c << " " << r << endl;
```

三、 [CSP-J 2025] 异或和 / xor

题目描述

小 R 有一个长度为 n 的非负整数序列 $a_1, a_2, \ldots, a_n$。定义一个区间 $[l, r]$ $(1 \le l \le r \le n)$ 的权值为 $a_l, a_{l+1}, \ldots, a_r$ 的二进制按位异或和，即 $a_l \oplus a_{l+1} \oplus \cdots \oplus a_r$，其中 $\oplus$ 表示二进制按位异或。

小 X 给了小 R 一个非负整数 k。小 X 希望小 R 选择序列中尽可能多的不相交的区间，使得每个区间的权值均为 k。两个区间 $[l_1, r_1], [l_2, r_2]$ 相交当且仅当两个区间同时包含至少一个相同的下标，即存在 $1 \le i \le n$ 使得 $l_1 \le i \le r_1$ 且 $l_2 \le i \le r_2$。

例如，对于序列 $[2, 1, 0, 3]$，若 $k = 2$，则小 R 可以选择区间 $[1, 1]$ 和区间 $[2, 4]$，权值分别为 2 和 $1 \oplus 0 \oplus 3 = 2$；若 $k = 3$，则小 R 可以选择区间 $[1, 2]$ 和区间 $[4, 4]$，权值分别为 $1 \oplus 2 = 3$ 和 3。

输入输出格式

输入：第一行，两个非负整数 n, k，分别表示序列长度和给定的非负整数。

第二行，n 个非负整数 $a_1, a_2, \ldots, a_n$，表示小 R 的序列。

输出：一行一个非负整数，表示能选出的不相交区间数量的最大值。

输入示例	输出示例
4 2 2 1 0 3	2
4 3 2 1 0 3	2
4 0 2 1 0 3	1

样例解释

样例1：可以选择区间 $[1, 1]$ 和区间 $[2, 4]$，异或和分别为 2 和 $1 \oplus 0 \oplus 3 = 2$，且两个区间不相交。

样例2：可以选择区间 $[1, 2]$ 和区间 $[4, 4]$，异或和分别为 $1 \oplus 2 = 3$ 和 3，且两个区间不相交。

样例3：可以选择区间 $[3, 3]$，异或和为 0。不能同时选择区间 $[3, 3]$ 和区间 $[1, 4]$，因为这两个区间相交。

算法分析

1. 前缀异或性质

   定义 $pre[i]$ 为序列前 i 个元素的异或和（即 $a_1 \oplus a_2 \oplus \cdots \oplus a_i$），规定 $pre[0] = 0$。根据异或的性质，任意区间 $[l, r]$ 的异或和可以表示为：

$$a_l \oplus \cdots \oplus a_r = pre[r] \oplus pre[l-1]$$

题目要求区间异或和为 k，即 $pre[r] \oplus pre[l-1] = k$，移项得：

$$pre[l-1] = pre[r] \oplus k$$

2. **动态规划状态定义**

   设 $dp[i]$ 表示在序列的前 i 个数中，能够选出的满足条件的不相交区间的**最大数量**。

3. **状态转移方程**

   对于当前位置 i，我们有两种选择策略：

   （1） **不以 i 作为区间的结尾**：此时最大数量继承自前一个位置，即 $dp[i] = dp[i-1]$。

   （2） **寻找一个区间以 i 结尾**：我们需要找到一个下标 j $(0 \le j < i)$，使得区间 $[j+1, i]$ 的异或和为 k。

   - 由前缀异或性质可知，需要满足 $pre[j] = pre[i] \oplus k$。
   - 如果存在这样的 j，则可以转移：$dp[i] = \max(dp[i], dp[j] + 1)$。

   综上，状态转移方程为：

$$dp[i] = \max(dp[i-1], \max_{j \text{ satisfying condition}}(dp[j] + 1))$$

4. **哈希表优化**

   为了快速找到满足 $pre[j] = pre[i] \oplus k$ 的最佳下标 j，我们使用哈希表 map<int, int>

   - **键 (Key)**：前缀异或值 val。
   - **值 (Value)**：该前缀异或值对应的下标 j。
   - **贪心更新策略**：为了保证 $dp[i]$ 最大，我们在哈希表中记录该前缀异或值出现时对应的**最大 dp 值**的下标。即：当计算出新的 $dp[i]$ 后，如果它比之前记录的同前缀异或值的 dp 结果更优，才更新哈希表。

参考代码

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

using u32 = unsigned;
using i32 = int;
using u64 = unsigned long long;
using i64 = long long;
using u128 = unsigned __int128;
using i128 = __int128;

#define int long long
#define endl "\n"

constexpr i64 inf = 1e18;
```

```cpp
void slu() {
    int n, k;
    cin >> n >> k;
    vector<int> a(n);
    for (auto &x : a) cin >> x;
    vector<int> pre(n + 1, 0);
    map<int, int> mp;
    vector<int> dp(n + 1, 0);
    for (int i = 1; i <= n; i++) {
        pre[i] = pre[i - 1] ^ a[i - 1];
    }

    mp[0] = 0;
    for (int i = 1; i <= n; i++) {
        int l = pre[i] ^ k;
        dp[i] = dp[i - 1];
        if (mp.contains(l)) {
            dp[i] = max(dp[i], dp[mp[l]] + 1);
        }
        if (dp[i] > dp[mp[pre[i]]]) {
            mp[pre[i]] = i;
        }
    }
    cout << dp[n];
}

signed main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    int t = 1;
    // cin >> t;

    while (t--) slu();
    return 0;
}
```


拓展思考

- 优化空间：可以使用 `unordered_map` 替代 `map`，将单次查找的时间复杂度从 $O(\log n)$ 优化到期望 $O(1)$，从而将总时间复杂度优化到 $O(n)$。

四、 [CSP-J 2025] 多边形 / polygon

题目描述

小 R 有 n 根小木棍，第 i $(1 \le i \le n)$ 根小木棍的长度为 a_i。小 X 希望小 R 从这 n 根小木棍中选出若干根小木棍，将它们按任意顺序首尾相连拼成一个多边形。拼成多边形的条件：对于长度分别为 $l_1, l_2, \ldots, l_m$ 的 m 根小木棍，能拼成一个多边形当且仅当 $m \ge 3$ 且所有小木棍的长度之和大于所有小木棍的长度最大值的两倍，即 $\sum_{i=1}^{m} l_i > 2 \times \max_{i=1}^{m} l_i$。

小 X 提出的问题是：有多少种选择小木棍的方案，使得选出的小木棍能够拼成一个多边形？两种方案不同当且仅当选择的小木棍的下标集合不同。由于答案可能较大，你只需要求出答案对 $998,244,353$ 取模后的结果。

输入输出格式

输入：第一行，一个正整数 n，表示小木棍的数量。

第二行，n 个正整数 $a_1, a_2, \ldots, a_n$，表示小木棍的长度。

输出：一行一个非负整数，表示能拼成多边形的方案数对 $998,244,353$ 取模后的结果。

输入示例	输出示例
5 1 2 3 4 5	9
5 2 2 3 8 10	6

样例解释

样例1：共有9种选择方案能使选出的小木棍拼成多边形，具体方案如下：

1. 选择第 2,3,4 根小木棍，长度之和为 $2+3+4=9$，满足 $9 > 2 \times 4 = 8$；
2. 选择第 2,4,5 根小木棍，长度之和为 $2+4+5=11$，满足 $11 > 2 \times 5 = 10$；
3. 选择第 3,4,5 根小木棍，长度之和为 $3+4+5=12$，满足 $12 > 2 \times 5 = 10$；
4. 选择第 1,2,3,4 根小木棍，长度之和为 $1+2+3+4=10$，满足 $10 > 2 \times 4 = 8$；
5. 选择第 1,2,3,4,5 根小木棍，长度之和为 $1+2+3+4+5=15$，满足 $15 > 2 \times 5 = 10$。
6. ...

算法分析

1. 问题转化与补集思想

   直接计算合法方案较为困难，我们采用正难则反的策略。

- **总方案数推导**：对于 n 根木棍，每一根木棍都有“选”或“不选”两种状态。根据数学中的乘法原理，总的组合数为 $2 \times 2 \times \cdots \times 2 = 2^n$ 种。由于题目要求选出木棍（空集没有意义），我们扣除掉“一根都不选”的情况，因此总方案数：

$$Total = 2^n - 1$$

- **合法条件**：集合总和 $Sum > 2 \times Max$。
- **非法条件**：集合总和 $Sum \leq 2 \times Max$。
- 设集合中除最大值以外的元素和为 S_{rest}，则非法条件等价于：

$$S_{rest} + Max \leq 2 \times Max \implies S_{rest} \leq Max$$

- **结论**：若除最大边外的其余边之和不超过最大边，则该集合无法构成多边形。

2. 排序与枚举

（1） 首先将数组 a 从小到大排序。

（2） 枚举每一个 a_i 作为当前子集的**最大值**。

（3） 此时，我们只能从下标 $[0, i-1]$ 的木棍中选择若干根，使得它们的和 $\leq a_i$。这些组合即为以 a_i 为最大值的**非法方案**。

3. 动态规划（01背包）

- **定义**：$dp[j]$ 表示从之前的木棍中选出若干根，长度之和为 j 的方案数。
- **非法统计**：对于当前的 a_i，所有满足 $j \leq a_i$ 的 $dp[j]$ 之和，即为以 a_i 为最大值的非法方案数。

```
for (int j = 0; j <= a[i]; j++) {
    illegal = (illegal + dp[j]) % MOD;
}
```

- **转移**：统计完成后，将 a_i 加入背包供后续使用（01背包倒序更新）：$dp[j] = dp[j] + dp[j - a_i]$。

```
for (int j = Max; j >= a[i]; j--) {
    dp[j] = (dp[j] + dp[j - a[i]]) % MOD;
}
```

- **最终答案**：$Total - Illegal = (2^n - 1) - \sum$(针对每个 a_i 的非法方案)。

算法复杂度分析

- 时间复杂度：$O(n \times V)$，其中 $V = \max(a_i)$。我们需要遍历 n 个元素，对于每个元素，都需要更新大小为 V 的背包数组。
- 空间复杂度：$O(V)$，其中 $V = \max(a_i)$，用于存储动态规划数组 dp。

代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
using i64 = long long;
#define int long long
constexpr i64 inf = 1e18;
constexpr i64 MOD = 998244353;

int qpow(int a, int b, int m) {
    a %= m;
    int res = 1;
    while (b > 0) {
        if (b & 1) res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res % m;
}

void slu() {
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++) cin >> a[i];
    sort(a.begin(), a.end());

    int Max = a.back(),illegal = 0;
    int res = qpow(2, n, MOD) - 1;
    vector<int> dp(Max + 1, 0);
    dp[0] = 1;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= a[i]; j++) {
            illegal = (illegal + dp[j]) % MOD;
        }
        for (int j = Max; j >= a[i]; j--) {
            dp[j] = (dp[j] + dp[j - a[i]]) % MOD;
        }
    }
    res = (res - illegal + MOD) % MOD;
    cout << res << endl;
}

signed main() {
    ios_base::sync_with_stdio(false);cin.tie(nullptr);
    slu();
    return 0;
}
```