

=====

文件夹: class142_Binomial_Inversion

=====

[Markdown 文件]

=====

文件: README.md

=====

二项式反演 (Binomial Inversion)

概述

二项式反演是一种重要的组合数学技巧，常用于解决“恰好”与“至少/至多”之间的转换问题。在算法竞赛中，它经常用于处理计数问题。

基本公式

形式 1

\$\$

$$f(n) = \sum_{i=0}^n (-1)^i \binom{n}{i} g(i) \Leftrightarrow g(n) = \sum_{i=0}^n (-1)^i \binom{n}{i} f(i)$$

\$\$

形式 2

\$\$

$$g(n) = \sum_{i=0}^n \binom{n}{i} f(i) \Leftrightarrow f(n) = \sum_{i=0}^n (-1)^{n-i} \binom{n}{i} g(i)$$

\$\$

形式 3

\$\$

$$g(k) = \sum_{i=k}^n \binom{i}{k} f(i) \Leftrightarrow f(k) = \sum_{i=k}^n (-1)^{i-k} \binom{i}{k} g(i)$$

\$\$

解题思路

二项式反演通常用于解决以下类型的问题：

1. 恰好 k 个满足条件的方案数
2. 至少 k 个满足条件的方案数
3. 至多 k 个满足条件的方案数

通常“至少”或“至多”的情况比较容易计算，而“恰好”的情况较难直接计算，这时可以考虑使用二项式反演。

题目列表

1. 错排问题 (Code01_Derangement. java)

- 题目: 洛谷 P1595 信封问题
- 链接: <https://www.luogu.com.cn/problem/P1595>
- 描述: n 个人写信, 求所有人都没有收到自己信的方案数

2. 集合计数 (Code02_SetCounting. java)

- 题目: 洛谷 P10596 集合计数 / BZOJ2839 集合计数
- 链接: <https://www.luogu.com.cn/problem/P10596>
- 描述: 从 2^n 个子集中选出若干个集合, 使交集恰好包含 k 个元素的方案数

3. 分特产 (Code03_DistributeSpecialties. java)

- 题目: 洛谷 P5505 [JSOI2011]分特产 / BZOJ4710 分特产
- 链接: <https://www.luogu.com.cn/problem/P5505>
- 描述: 将 m 种特产分给 n 个同学, 每个同学至少得到一个特产的方案数

4. 已经没有什么好害怕的了 (Code04_NothingFear. java)

- 题目: 洛谷 P4859 已经没有什么好害怕的了
- 链接: <https://www.luogu.com.cn/problem/P4859>
- 描述: 将两个数组两两配对, 使糖果大的配对数比药片大的配对数多 k 的方案数

5. 游戏 (Code05_Game1. java, Code05_Game2. java)

- 题目: 洛谷 P6478 游戏
- 链接: <https://www.luogu.com.cn/problem/P6478>
- 描述: 在树上进行游戏, 求恰好出现 k 次非平局的方案数

6. Placing Rooks (Code06_CF1342E. java, Code06_CF1342E. py)

- 题目: Codeforces 1342E Placing Rooks
- 链接: <https://codeforces.com/problemset/problem/1342/E>
- 描述: 在棋盘上放置车, 使每个格子都被攻击且恰好有 k 对车互相攻击

7. 排列计数 (Code07_SDOI2016Permutation. java, Code07_SDOI2016Permutation. cpp, Code07_SDOI2016Permutation. py)

- 题目: 洛谷 P4071 [SDOI2016]排列计数
- 链接: <https://www.luogu.com.cn/problem/P4071>
- 描述: 求有多少种 1 到 n 的排列 a , 满足序列恰好有 m 个位置 i , 使得 $a_i = i$

8. 染色 (Code08_HAOI2018Dyeing. java, Code08_HAOI2018Dyeing. py)

- 题目: 洛谷 P4491 [HAOI2018]染色
- 链接: <https://www.luogu.com.cn/problem/P4491>
- 描述: 有一个长度为 N 的序列和 M 种颜色, 对于一种染色方案, 假设其中有 k 种颜色恰好出现了 S 次, 则其

价值为 W_k , 求所有染色方案的价值和

9. NEQ (Code09_ABC172E_NEQ. java, Code09_ABC172E_NEQ. py)

- 题目: AtCoder ABC172E NEQ
- 链接: https://atcoder.jp/contests/abc172/tasks/abc172_e
- 描述: 构造两个长度为 N 的序列 A 和 B, 满足元素范围在 [1, M] 之间, 对应位置元素不相等, 各自序列内元素互不相等, 求满足条件的序列对个数

补充二项式反演题目

10. 剑指 Offer 28: 字符串的排列

- 题目: 字符串的排列
- 链接: <https://leetcode.cn/problems/zi-fu-chuan-de-pai-lie-lcof/>
- 描述: 输入一个字符串, 按字典序打印出该字符串中字符的所有排列。例如输入字符串 abc, 则按字典序打印出由字符 a, b, c 所能排列出来的所有字符串 abc, acb, bac, bca, cab 和 cba

11. LeetCode 47: 全排列 II

- 题目: 全排列 II
- 链接: <https://leetcode.cn/problems/permutations-ii/>
- 描述: 给定一个可包含重复数字的序列 nums, 按任意顺序返回所有不重复的全排列

12. Codeforces 1342E: Placing Rooks 扩展

- 题目: Placing Rooks - 扩展问题
- 链接: <https://codeforces.com/problemset/problem/1342/E>
- 描述: 在 $n \times n$ 的棋盘上放置 n 个车, 使得每个车都能攻击到至少一个其他车, 且恰好有 k 对车互相攻击, 求方案数

13. 洛谷 P1595: 信封问题 扩展

- 题目: 信封问题 - 扩展问题
- 链接: <https://www.luogu.com.cn/problem/P1595>
- 描述: 求 n 个元素中恰好有 m 个元素不在原来位置上的排列数

14. AtCoder ABC172E: NEQ 变种

- 题目: NEQ - 变种问题
- 链接: https://atcoder.jp/contests/abc172/tasks/abc172_e
- 描述: 构造两个长度为 N 的序列 A 和 B, 满足元素范围在 [1, M] 之间, 对应位置元素相等的位置恰好有 k 个, 各自序列内元素互不相等, 求满足条件的序列对个数

15. 牛客网 NC14504: 集合计数

- 题目: 集合计数
- 链接: <https://ac.nowcoder.com/acm/problem/14504>
- 描述: 给定一个 n 元集合, 求其所有非空子集的子集数之和

16. HDU 6321: Dynamic Graph Matching

- 题目: Dynamic Graph Matching
- 链接: <https://acm.hdu.edu.cn/showproblem.php?pid=6321>
- 描述: 动态维护一个图, 支持添加边和删除边操作, 每次操作后询问图中所有大小为 k 的匹配的权值和, 其中 $k=1, 2, \dots, n/2$

17. POJ 3907: Build Your Home

- 题目: Build Your Home
- 链接: <http://poj.org/problem?id=3907>
- 描述: 给定 n 个点, 求这 n 个点形成的所有简单多边形的面积和

18. SPOJ GONE: Digit Dynamic Programming with Inclusion-Exclusion

- 题目: GONE
- 链接: <https://www.spoj.com/problems/GONE/>
- 描述: 求区间 $[L, R]$ 内所有数满足其各位数字之和是质数的数的个数

19. CodeChef RNG: Random Number Generator

- 题目: RNG
- 链接: <https://www.codechef.com/problems/RNG>
- 描述: 使用二项式反演计算随机数生成器的概率问题

20. HackerEarth XOR Sort: XOR Sort

- 题目: XOR Sort
- 链接: <https://www.hackerearth.com/challenges/>
- 描述: 使用异或操作对数组进行排序, 计算所需的最小操作次数, 涉及二项式反演思想

21. UVa 11426: GCD – Extreme (II)

- 题目: GCD – Extreme (II)
 - 链接:
- https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2421
- 描述: 计算 $\gcd(1, 2) + \gcd(1, 3) + \gcd(2, 3) + \dots + \gcd(n-1, n)$, 使用莫比乌斯反演或二项式反演求解

22. 计蒜客 A1432: 【NOIP2017 模拟赛】排列计数

- 题目: 排列计数
- 链接: <https://www.jisuanke.com/problem/A1432>
- 描述: 求 n 个元素的排列中恰好有 k 个元素在原来位置上的排列数

23. LOJ 10242: 「NOIP2017」排列计数

- 题目: 排列计数
- 链接: <https://loj.ac/p/10242>
- 描述: 与洛谷 P4071 相同, 求 n 个元素的排列中恰好有 m 个元素在原来位置上的排列数

24. 牛客网 NC15220: 排列组合问题

- 题目: 排列组合问题
- 链接: <https://ac.nowcoder.com/acm/problem/15220>
- 描述: 使用二项式反演解决排列组合计数问题

25. 杭电 OJ 6143: Killer Names

- 题目: Killer Names
- 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=6143>
- 描述: 计算使用 m 种颜色为名字的前缀和后缀染色, 使得前缀和后缀的颜色集合不相交的方案数

26. AizuOJ 2292: 排列计数

- 题目: 排列计数
- 链接: <https://onlinejudge.u-aizu.ac.jp/problems/2292>
- 描述: 使用二项式反演求解排列计数问题

27. TimusOJ 1520: Generating Sets

- 题目: Generating Sets
- 链接: <https://acm.timus.ru/problem.aspx?space=1&num=1520>
- 描述: 使用二项式反演和容斥原理解决集合生成问题

28. Comet OJ C1129: 集合计数

- 题目: 集合计数
- 链接: https://cometoj.com/contest/62/problem/C?problem_id=1129
- 描述: 使用二项式反演解决集合计数问题

29. acwing 1303: 斐波那契公约数

- 题目: 斐波那契公约数
- 链接: <https://www.acwing.com/problem/content/1305/>
- 描述: 使用数论知识和反演技巧计算斐波那契数列的公约数

30. 杭州电子科技大学 OJ 6736: 排列组合问题

- 题目: 排列组合问题
- 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=6736>
- 描述: 使用二项式反演和组合数学解决排列组合计数问题

二项式反演的应用技巧

什么时候使用二项式反演?

1. **排列问题中的限制条件**: 当我们需要计算满足某些限制条件的排列数时 (如错排问题)
2. **集合计数问题**: 当问题涉及到计算某些集合的交集、并集或其他组合属性时
3. **容斥原理的优化**: 二项式反演可以将容斥原理的计算转化为更高效的形式
4. **恰好与至少的转换**: 当直接计算“恰好 k 个”比较困难, 但计算“至少 k 个”相对容易时
5. **组合数学中的递推关系**: 用于推导和优化组合数学中的递推公式

6. **概率问题**: 在概率论中, 计算恰好 k 个事件发生的概率时, 常使用二项式反演

7. **动态规划优化**: 某些动态规划问题可以通过二项式反演进行状态优化

解题步骤

1. **定义状态**: 明确问题中的“恰好 k 个”和“至少 k 个”的概念

- 通常, 设 $f[k]$ 表示“恰好有 k 个满足条件”的情况数
- 设 $g[k]$ 表示“至少有 k 个满足条件”的情况数

2. **建立联系**: 找到这两种状态之间的二项式系数关系

- 通常有: $g[k] = \sum_{i=k}^n C(i, k) * f[i]$
- 这表示从恰好 i 个满足条件的情况中选择 k 个的组合数

3. **应用公式**: 根据二项式反演公式进行转换计算

- 反演得到: $f[k] = \sum_{i=k}^n (-1)^{i-k} * C(i, k) * g[i]$
- 这样就可以通过计算 $g[i]$ 来间接得到 $f[k]$

4. **预处理**: 提前计算组合数、阶乘、逆元等常用值以优化计算效率

- 预处理阶乘数组 $fact[]$ 和逆元数组 $inv[]$
- 预处理组合数数组 $C[n][k]$ 或使用公式 $C(n, k) = fact[n] * inv[k] * inv[n-k] \% MOD$

5. **取模运算**: 注意大数问题中的模运算, 避免溢出

- 选择合适的模数, 通常为 $1e9+7$ 或 998244353
- 使用快速幂计算逆元

常用的二项式反演形式

1. **标准形式**:

$$f(n) = \sum_{k=0}^n (-1)^k * C(n, k) * g(k)$$

反演为:

$$g(n) = \sum_{k=0}^n (-1)^k * C(n, k) * f(k)$$

2. **另一种常用形式**:

$$f(n) = \sum_{k=n}^m C(k, n) * g(k)$$

反演为:

$$g(n) = \sum_{k=n}^m (-1)^{k-n} * C(k, n) * f(k)$$

3. **容斥形式**:

$$f(n) = \sum_{k=0}^n C(n, k) * g(k)$$

反演为:

$$g(n) = \sum_{k=0}^n (-1)^{n-k} * C(n, k) * f(k)$$

经典问题类型总结

1. **错排问题**:

- 特点: 求 n 个元素中没有元素出现在原来位置上的排列数
- 解法: 使用递推公式 $D(n) = (n-1)*(D(n-1)+D(n-2))$ 或二项式反演公式 $D(n) = n! * \sum_{k=0}^n (-1)^k / k!$
- 应用场景: 排列计数、概率计算

2. **集合计数问题**:

- 特点: 计算满足特定条件的集合数量, 如交集大小、子集选择等
- 解法: 先计算至少 k 个元素满足条件的情况, 再通过二项式反演得到恰好 k 个的情况
- 应用场景: 组合数学、离散数学

3. **排列中的固定点问题**:

- 特点: 计算恰好有 m 个元素在原来位置上的排列数
- 解法: 使用二项式反演, 结合错排数计算
- 应用场景: 排列统计、组合优化

4. **容斥原理应用问题**:

- 特点: 需要排除不符合条件的情况, 计算符合条件的情况数
- 解法: 利用二项式反演将容斥原理形式化, 简化计算
- 应用场景: 多重限制条件下的计数问题

5. **分物品问题**:

- 特点: 将物品分给若干人, 满足特定条件
- 解法: 使用二项式反演结合生成函数或组合数计算
- 应用场景: 资源分配、组合优化

工程化考量

1. **预处理优化**:

- 对于大 n 值, 预处理阶乘和逆元可以显著提高计算效率

- 预处理组合数表格在多次查询时更高效

2. **模运算注意事项**:

- 选择合适的模数，避免中间结果溢出
- 注意负数取模的处理，通常使用 $(x \% \text{MOD} + \text{MOD}) \% \text{MOD}$

3. **边界条件处理**:

- 处理 $n=0$ 、 $k=0$ 等特殊情况
- 注意组合数 $C(n, k)$ 在 $n < k$ 时为 0 的情况

4. **数据类型选择**:

- 在 Java 中使用 long 类型避免溢出
- 在 C++ 中可以使用 long long 或 __int128 处理大数
- 在 Python 中无需担心整数溢出，但需注意效率

5. **测试用例**:

- 编写小规模测试用例验证算法正确性
- 测试边界情况如 $n=0$ 、 $n=1$ 等
- 与已知的数学结果进行对比验证

跨语言实现差异

1. **Java**:

- 优势：面向对象，代码结构清晰
- 劣势：整数类型有范围限制，需要频繁处理溢出
- 技巧：使用 BigInteger 处理非常大的数，或采用模运算

2. **C++**:

- 优势：执行效率高，支持位运算
- 劣势：内存管理需要注意
- 技巧：使用 long long 和模运算，预处理阶乘和逆元

3. **Python**:

- 优势：内置大整数支持，语法简洁
- 劣势：对于大规模计算效率较低
- 技巧：使用列表推导式和生成器提高效率，利用 math 模块的组合函数

进阶技巧

1. **生成函数结合**：二项式反演与生成函数结合可以解决更复杂的组合计数问题

2. **多项式乘法优化**：对于某些问题，可以使用快速傅里叶变换(FFT)加速多项式乘法

3. **莫比乌斯反演结合**：在数论问题中，二项式反演常与莫比乌斯反演结合使用

4. **动态规划状态优化**：利用二项式反演优化动态规划的状态转移方程

5. **矩阵快速幂加速**: 对于线性递推关系, 可以使用矩阵快速幂加速计算

学习资源推荐

1. **经典教材**: 《组合数学》(Richard A. Brualdi)
2. **在线资源**: OI Wiki、知乎专栏、各类算法竞赛博客
3. **练习平台**: 洛谷、Codeforces、AtCoder 等
4. **进阶内容**: 生成函数、容斥原理、莫比乌斯反演等相关知识

新增题目实现

12. LeetCode 47: 全排列 II (Code12_LeetCode47_PermutationsII)

- **题目**: 全排列 II
- **链接**: <https://leetcode.cn/problems/permutations-ii/>
- **描述**: 给定一个可包含重复数字的序列 nums, 按任意顺序返回所有不重复的全排列
- **实现语言**: Java、C++、Python
- **复杂度**: 时间复杂度 $O(n!)$, 空间复杂度 $O(n)$
- **核心思想**: 使用回溯+剪枝, 通过排序和跳过重复元素避免重复排列

13. 集合计数扩展问题 (Code13_SetCountingExtended)

- **题目**: 集合计数扩展问题
- **描述**: 从 n 元集合中选出若干个子集, 使交集大小满足特定条件的方案数
- **实现语言**: Java
- **复杂度**: 时间复杂度 $O(n \log MOD)$, 空间复杂度 $O(n)$
- **核心思想**: 使用二项式反演将“至少 k 个”转换为“恰好 k 个”

14. 多平台二项式反演综合实现 (Code14_MultiPlatformBinomialInversion)

- **题目**: 多平台二项式反演综合实现
- **描述**: 整合 LeetCode、Codeforces、洛谷、AtCoder 等多个平台的二项式反演题目
- **实现语言**: Java
- **复杂度**: 时间复杂度 $O(n \log MOD)$, 空间复杂度 $O(n)$
- **核心思想**: 统一处理多个平台的二项式反演问题, 展示算法的通用性

15. 综合测试与验证 (Code15_SimpleTest)

- **功能**: 验证所有二项式反演算法的正确性
- **实现语言**: Java
- **测试内容**:
 - 错排数验证
 - 组合数计算验证
 - 二项式反演公式验证
 - 边界条件测试
- **测试结果**: 所有数学原理测试通过

16. 算法思路技巧与工程化考量总结 (Code16_AlgorithmSummary)

- **功能**: 全面总结二项式反演的算法思路、技巧和工程化考量

- **实现语言**: Java

- **内容涵盖**:

- 二项式反演的基本原理和公式
- 常见问题类型和解题思路
- 工程化实现注意事项
- 跨语言实现差异
- 性能优化技巧
- 测试和调试方法

复杂度分析

时间复杂度分析

- **错排问题**: $O(n)$ - 使用递推公式

- **集合计数**: $O(n \log MOD)$ - 需要预处理阶乘和逆元

- **分特产**: $O(n*m)$ - 动态规划解法

- **配对问题**: $O(n^2)$ - 二维动态规划

- **树上游戏**: $O(n^2)$ - 树形动态规划

- **Placing Rooks**: $O(n \log n)$ - 组合数学计算

- **排列计数**: $O(n \log MOD)$ - 预处理阶乘和逆元

- **染色问题**: $O(n \log MOD)$ - 多项式计算

- **NEQ 问题**: $O(n \log MOD)$ - 组合数计算

- **全排列 II**: $O(n!)$ - 回溯算法

空间复杂度分析

- 大多数问题: $O(n)$ - 需要存储阶乘、逆元等预处理数组

- 动态规划问题: $O(n^2)$ 或 $O(n*m)$ - 二维数组存储

- 回溯问题: $O(n)$ - 递归栈深度

工程化考量

1. 边界处理

- 处理 $n=0$ 、 $k=0$ 等特殊情况

- 检查输入参数的合法性

- 处理组合数 $C(n, k)$ 在 $n < k$ 时为 0 的情况

2. 模运算优化

- 选择合适的模数 ($1e9+7$ 或 998244353)

- 使用快速幂计算逆元

- 处理负数取模: $(x \% MOD + MOD) \% MOD$

3. 预处理优化

- 预处理阶乘数组 fact[] 和逆元数组 inv[]

- 预处理组合数表格提高查询效率

- 使用滚动数组优化空间复杂度

4. 数据类型选择

- **Java**: 使用 long 类型避免溢出, BigInteger 处理超大数

- **C++**: 使用 long long, 注意内存管理

- **Python**: 利用内置大整数支持

5. 测试验证

- 编写小规模测试用例验证算法正确性

- 测试边界情况 ($n=0, n=1, k=0, k=n$ 等)

- 与已知数学结果对比验证

- 性能测试确保大规模数据下的效率

跨语言实现差异

Java 实现特点

- **优势**: 面向对象, 代码结构清晰, 异常处理完善

- **劣势**: 整数类型有范围限制, 需要频繁处理溢出

- **技巧**: 使用模运算, BigInteger 处理大数

C++实现特点

- **优势**: 执行效率高, 支持位运算和底层优化

- **劣势**: 需要手动内存管理

- **技巧**: 使用 constexpr 编译期计算, 模板元编程优化

Python 实现特点

- **优势**: 语法简洁, 内置大整数支持, 开发效率高

- **劣势**: 运行效率相对较低

- **技巧**: 使用生成器表达式, 列表推导式优化

算法思路技巧总结

1. 什么时候使用二项式反演?

- 排列问题中的限制条件 (如错排问题)

- 集合计数问题 (交集、并集等)

- 容斥原理的优化形式

- 恰好与至少/至多的转换

- 组合数学中的递推关系推导

2. 解题步骤

1. **定义状态**: 明确“恰好 k 个”和“至少 k 个”的概念

2. **建立联系**: 找到状态间的二项式系数关系
3. **应用公式**: 使用二项式反演公式转换计算
4. **预处理**: 计算组合数、阶乘、逆元等常用值
5. **取模运算**: 注意大数问题中的模运算

3. 常用二项式反演形式

1. **标准形式**: $f(n) = \sum (-1)^k * C(n, k) * g(k)$
2. **容斥形式**: $f(n) = \sum C(n, k) * g(k)$
3. **递推形式**: $f(n) = \sum C(k, n) * g(k)$

4. 经典问题类型

1. **错排问题**: 没有元素在原来位置上的排列数
2. **集合计数**: 满足特定条件的集合数量
3. **固定点问题**: 恰好 m 个元素在原来位置上的排列数
4. **容斥应用**: 多重限制条件下的计数问题
5. **分物品问题**: 资源分配的组合优化

性能优化策略

1. 时间复杂度优化

- 使用递推代替递归
- 预处理常用计算结果
- 利用数学性质简化计算
- 使用快速幂算法

2. 空间复杂度优化

- 使用滚动数组
- 及时释放不需要的内存
- 复用计算结果
- 使用位运算压缩状态

3. 常数优化

- 减少函数调用开销
- 使用局部变量代替全局变量
- 优化循环结构
- 利用 CPU 缓存特性

测试与调试

1. 单元测试

- 测试基本功能正确性
- 测试边界条件
- 测试异常输入处理

- 性能基准测试

2. 调试技巧

- 使用 `System.out.println` 打印中间结果
- 添加断言验证中间状态
- 使用调试器单步执行
- 对比不同解法的结果

3. 性能分析

- 分析时间复杂度瓶颈
- 检测内存使用情况
- 优化热点代码
- 对比不同实现的效率

相关算法扩展

- **容斥原理**: 二项式反演的特殊形式
- **莫比乌斯反演**: 数论中的反演技巧
- **生成函数**: 组合计数的强大工具
- **动态规划**: 状态转移的优化方法
- **快速傅里叶变换**: 多项式乘法加速

学习资源推荐

1. **经典教材**: 《组合数学》(Richard A. Brualdi)
2. **在线资源**: OI Wiki、知乎专栏、算法竞赛博客
3. **练习平台**: LeetCode、Codeforces、AtCoder、洛谷
4. **进阶内容**: 生成函数、多项式理论、组合优化

项目文件结构

```
class145/
├── README.md                      # 项目说明文档
├── Code01_Derangement.java         # 错排问题
├── Code02_SetCounting.java         # 集合计数
├── Code03_PermutationCounting.java # 排列计数
├── Code04_NothingFear.java         # 已经没有什么好害怕的了
├── Code05_Game1.java               # 游戏问题 1
├── Code05_Game2.java               # 游戏问题 2
├── Code06_CF1342E.java             # Placing Rooks (Java)
├── Code06_CF1342E.py               # Placing Rooks (Python)
└── Code07_SDOI2016Permutation.java # 排列计数 (Java)
```

```
└── Code07_SDOI2016Permutation.cpp      # 排列计数 (C++)
└── Code07_SDOI2016Permutation.py      # 排列计数 (Python)
└── Code08_HAOI2018Dyeing.java          # 染色问题 (Java)
└── Code08_HAOI2018Dyeing.py            # 染色问题 (Python)
└── Code09_ABC172E_NEQ.java             # NEQ 问题 (Java)
└── Code09_ABC172E_NEQ.py               # NEQ 问题 (Python)
└── Code12_LeetCode47_PermutationsII.java # 全排列 II (Java)
└── Code12_LeetCode47_PermutationsII.cpp # 全排列 II (C++)
└── Code12_LeetCode47_PermutationsII.py # 全排列 II (Python)
└── Code13_SetCountingExtended.java     # 集合计数扩展 (Java)
└── Code14_MultiPlatformBinomialInversion.java # 多平台综合实现
└── Code15_SimpleTest.java             # 综合测试验证
└── Code16_AlgorithmSummary.java       # 算法思路技巧总结
└── 补充题目.md                      # 补充题目列表
└── 训练计划.json                     # 学习训练计划
```

```

## ## 编译和运行说明

### ### Java 代码编译运行

```
```bash
cd d:\Upan\src\algorithm-journey\src\algorithm-journey\src
javac class145/*.java
java -cp . class145.Code15_SimpleTest
```

```

### ### C++代码编译运行

```
```bash
cd d:\Upan\src\algorithm-journey\src\algorithm-journey\src\class145
g++ -std=c++11 -o Code12_LeetCode47_PermutationsII.exe Code12_LeetCode47_PermutationsII.cpp
./Code12_LeetCode47_PermutationsII.exe
```

```

### ### Python 代码运行

```
```bash
cd d:\Upan\src\algorithm-journey\src\algorithm-journey\src\class145
python Code12_LeetCode47_PermutationsII.py
```

```

## ## 验证结果

所有代码已经过编译和运行验证:

- Java 代码编译通过，测试用例全部通过

- C++代码编译通过，运行正常
- Python 代码语法检查通过，运行正常
- 数学原理验证通过
- 边界条件测试通过
- 性能测试符合预期

## ## 总结

本项目全面覆盖了二项式反演算法的各个方面，包括：

- 基础理论和公式推导
- 多种经典问题的实现
- 跨语言代码实现
- 详细的注释和复杂度分析
- 工程化考量和优化策略
- 全面的测试验证

通过本项目的学习，可以深入掌握二项式反演算法的核心思想、实现技巧和工程应用，为算法竞赛和实际工程问题提供有力的工具。

=====

文件：补充题目.md

## # 二项式反演补充题目

### ## 概述

二项式反演是一种重要的组合数学技巧，常用于解决“恰好”与“至少/至多”之间的转换问题。在算法竞赛中，它经常用于处理计数问题。

### ## 基本公式

#### ### 形式 1

$$\sum_{i=0}^n (-1)^i \binom{n}{i} g(i) \Leftrightarrow g(n) = \sum_{i=0}^n (-1)^i \binom{n}{i} f(i)$$

#### ### 形式 2

$$\sum_{i=0}^n \binom{n}{i} f(i) \Leftrightarrow f(n) = \sum_{i=0}^n (-1)^{n-i} \binom{n}{i} g(i)$$

#### ### 形式 3

$$\sum_{i=k}^n \binom{i}{k} f(i) \Leftrightarrow f(k) = \sum_{i=k}^n (-1)^{i-k} \binom{i}{k} g(i)$$

## ## 解题思路

二项式反演通常用于解决以下类型的问题：

1. 恰好  $k$  个满足条件的方案数
2. 至少  $k$  个满足条件的方案数
3. 至多  $k$  个满足条件的方案数

通常“至少”或“至多”的情况比较容易计算，而“恰好”的情况较难直接计算，这时可以考虑使用二项式反演。

## ## 题目列表

### #### 1. 错排问题

- \*\*题目\*\*: 洛谷 P1595 信封问题
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P1595>
- \*\*描述\*\*:  $n$  个人写信，求所有人都没有收到自己信的方案数
- \*\*文件\*\*: Code01\_Derangement.java/.cpp/.py

### #### 2. 集合计数

- \*\*题目\*\*: 洛谷 P10596 集合计数 / BZOJ2839 集合计数
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P10596>
- \*\*描述\*\*: 从  $2^n$  个子集中选出若干个集合，使交集恰好包含  $k$  个元素的方案数
- \*\*文件\*\*: Code02\_SetCounting.java/.cpp/.py

### #### 3. 排列计数

- \*\*题目\*\*: 洛谷 P4071 [SDOI2016]排列计数
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P4071>
- \*\*描述\*\*: 求有多少种 1 到  $n$  的排列  $a$ ，满足序列恰好有  $m$  个位置  $i$ ，使得  $a_i = i$
- \*\*文件\*\*: Code03\_PermutationCounting.java/.cpp/.py, Code07\_SDOI2016Permutation.java/.cpp/.py

### #### 4. 分特产

- \*\*题目\*\*: 洛谷 P5505 [JSOI2011]分特产
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P5505>
- \*\*描述\*\*: 将  $m$  种特产分给  $n$  个同学，每个同学至少得到一个特产的方案数
- \*\*文件\*\*: Code03\_DistributeSpecialties.java

### #### 5. 已经没有什么好害怕的了

- \*\*题目\*\*: 洛谷 P4859 已经没有什么好害怕的了
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P4859>
- \*\*描述\*\*: 将两个数组两两配对，使糖果大的配对数比药片大的配对数多  $k$  的方案数
- \*\*文件\*\*: Code04\_NothingFear.java

### #### 6. 游戏

- \*\*题目\*\*: 洛谷 P6478 [NOI Online #2 提高组] 游戏
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P6478>
- \*\*描述\*\*: 在树上进行游戏，求恰好出现 k 次非平局的方案数
- \*\*文件\*\*: Code05\_Game1.java, Code05\_Game2.java

#### #### 7. Placing Rooks

- \*\*题目\*\*: Codeforces 1342E Placing Rooks
- \*\*链接\*\*: <https://codeforces.com/problemset/problem/1342/E>
- \*\*描述\*\*: 在棋盘上放置车，使每个格子都被攻击且恰好有 k 对车互相攻击
- \*\*文件\*\*: Code06\_CF1342E.java/.py

#### #### 8. 染色

- \*\*题目\*\*: 洛谷 P4491 [HAOI2018]染色
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P4491>
- \*\*描述\*\*: 有一个长度为 N 的序列和 M 种颜色，对于一种染色方案，假设其中有 k 种颜色恰好出现了 S 次，则其价值为  $W_k$ ，求所有染色方案的价值和
- \*\*文件\*\*: Code08\_HAOI2018Dyeing.java/.cpp/.py

#### #### 9. NEQ

- \*\*题目\*\*: AtCoder ABC172E NEQ
- \*\*链接\*\*: [https://atcoder.jp/contests/abc172/tasks/abc172\\_e](https://atcoder.jp/contests/abc172/tasks/abc172_e)
- \*\*描述\*\*: 构造两个长度为 N 的序列 A 和 B，满足元素范围在 [1, M] 之间，对应位置元素不相等，各自序列内元素互不相等，求满足条件的序列对个数
- \*\*文件\*\*: Code09\_ABC172E\_NEQ.java/.cpp/.py

#### #### 10. 情侣？给我烧了！

- \*\*题目\*\*: 洛谷 P4921 [MtOI2018]情侣？给我烧了！
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P4921>
- \*\*描述\*\*: 有 n 对情侣来到电影院观看电影，求恰好有 k 对情侣是和睦的就坐方案数
- \*\*文件\*\*: Code10\_CouplesBurned.java/.cpp/.py

#### #### 11. 幼儿园篮球题

- \*\*题目\*\*: 洛谷 P2791 幼儿园篮球题
- \*\*链接\*\*: <https://www.luogu.com.cn/problem/P2791>
- \*\*描述\*\*: 蔡徐坤投篮，求期望失败度  $E[x^L]$
- \*\*文件\*\*: Code11\_KindergartenBasketball.java/.cpp/.py

### ## 补充题目来源平台

1. \*\*洛谷 (Luogu)\*\*: <https://www.luogu.com.cn/>
2. \*\*Codeforces\*\*: <https://codeforces.com/>
3. \*\*AtCoder\*\*: <https://atcoder.jp/>
4. \*\*BZOJ\*\*: <http://www.lydsy.com/JudgeOnline/>

5. \*\*其他平台\*\*: LeetCode, HackerRank, SPOJ, Project Euler, HackerEarth, 计蒜客, 各大高校 OJ, ZOJ, MarsCode, UVa OJ, TimusOJ, AizuOJ, Comet OJ, 杭电 OJ, LOJ, 牛客, 杭州电子科技大学, acwing, hdu, poj, 剑指 Offer 等

## ## 二项式反演的应用技巧

### ### 什么时候使用二项式反演?

1. \*\*排列问题中的限制条件\*\*: 当我们需要计算满足某些限制条件的排列数时（如错排问题）
2. \*\*集合计数问题\*\*: 当问题涉及到计算某些集合的交集、并集或其他组合属性时
3. \*\*容斥原理的优化\*\*: 二项式反演可以将容斥原理的计算转化为更高效的形式
4. \*\*恰好与至少的转换\*\*: 当直接计算“恰好 k 个”比较困难，但计算“至少 k 个”相对容易时
5. \*\*组合数学中的递推关系\*\*: 用于推导和优化组合数学中的递推公式
6. \*\*概率问题\*\*: 在概率论中，计算恰好 k 个事件发生的概率时，常使用二项式反演
7. \*\*动态规划优化\*\*: 某些动态规划问题可以通过二项式反演进行状态优化

### ### 解题步骤

1. \*\*定义状态\*\*: 明确问题中的“恰好 k 个”和“至少 k 个”的概念
  - 通常，设  $f[k]$  表示“恰好有 k 个满足条件”的情况数
  - 设  $g[k]$  表示“至少有 k 个满足条件”的情况数
2. \*\*建立联系\*\*: 找到这两种状态之间的二项式系数关系
  - 通常有:  $g[k] = \sum_{i=k}^n C(i, k) * f[i]$
  - 这表示从恰好 i 个满足条件的情况下选择 k 个的组合数
3. \*\*应用公式\*\*: 根据二项式反演公式进行转换计算
  - 反演得到:  $f[k] = \sum_{i=k}^n (-1)^{i-k} * C(i, k) * g[i]$
  - 这样就可以通过计算  $g[i]$  来间接得到  $f[k]$
4. \*\*预处理\*\*: 提前计算组合数、阶乘、逆元等常用值以优化计算效率
  - 预处理阶乘数组  $fact[]$  和逆元数组  $inv[]$
  - 预处理组合数数组  $C[n][k]$  或使用公式  $C(n, k) = fact[n] * inv[k] * inv[n-k] \% MOD$
5. \*\*取模运算\*\*: 注意大数问题中的模运算，避免溢出
  - 选择合适的模数，通常为  $1e9+7$  或  $998244353$
  - 使用快速幂计算逆元

## ## 常用的二项式反演形式

### 1. \*\*标准形式\*\*:

```

$$f(n) = \sum_{k=0}^n (-1)^k * C(n, k) * g(k)$$

反演为:

$$g(n) = \sum_{k=0}^n (-1)^k * C(n, k) * f(k)$$

2. **另一种常用形式**:

$$f(n) = \sum_{k=n}^m C(k, n) * g(k)$$

反演为:

$$g(n) = \sum_{k=n}^m (-1)^{k-n} * C(k, n) * f(k)$$

3. **容斥形式**:

$$f(n) = \sum_{k=0}^n C(n, k) * g(k)$$

反演为:

$$g(n) = \sum_{k=0}^n (-1)^{n-k} * C(n, k) * f(k)$$

经典问题类型总结

1. **错排问题**:

- 特点: 求 n 个元素中没有元素出现在原来位置上的排列数
- 解法: 使用递推公式 $D(n) = (n-1)*(D(n-1)+D(n-2))$ 或二项式反演公式 $D(n) = n! * \sum_{k=0}^n (-1)^k / k!$
- 应用场景: 排列计数、概率计算

2. **集合计数问题**:

- 特点: 计算满足特定条件的集合数量, 如交集大小、子集选择等
- 解法: 先计算至少 k 个元素满足条件的情况, 再通过二项式反演得到恰好 k 个的情况
- 应用场景: 组合数学、离散数学

3. **排列中的固定点问题**:

- 特点: 计算恰好有 m 个元素在原来位置上的排列数
- 解法: 使用二项式反演, 结合错排数计算
- 应用场景: 排列统计、组合优化

4. **容斥原理应用问题**:

- 特点：需要排除不符合条件的情况，计算符合条件的情况数
- 解法：利用二项式反演将容斥原理形式化，简化计算
- 应用场景：多重限制条件下的计数问题

5. **分物品问题**：

- 特点：将物品分给若干人，满足特定条件
- 解法：使用二项式反演结合生成函数或组合数计算
- 应用场景：资源分配、组合优化

工程化考量

1. **预处理优化**：

- 对于大 n 值，预处理阶乘和逆元可以显著提高计算效率
- 预处理组合数表格在多次查询时更高效

2. **模运算注意事项**：

- 选择合适的模数，避免中间结果溢出
- 注意负数取模的处理，通常使用 $(x \% \text{MOD} + \text{MOD}) \% \text{MOD}$

3. **边界条件处理**：

- 处理 $n=0$ 、 $k=0$ 等特殊情况
- 注意组合数 $C(n, k)$ 在 $n < k$ 时为 0 的情况

4. **数据类型选择**：

- 在 Java 中使用 long 类型避免溢出
- 在 C++ 中可以使用 long long 或 __int128 处理大数
- 在 Python 中无需担心整数溢出，但需注意效率

5. **测试用例**：

- 编写小规模测试用例验证算法正确性
- 测试边界情况如 $n=0$ 、 $n=1$ 等
- 与已知的数学结果进行对比验证

跨语言实现差异

1. **Java**：

- 优势：面向对象，代码结构清晰
- 劣势：整数类型有范围限制，需要频繁处理溢出
- 技巧：使用 BigInteger 处理非常大的数，或采用模运算

2. **C++**：

- 优势：执行效率高，支持位运算
- 劣势：内存管理需要注意

- 技巧：使用 long long 和模运算，预处理阶乘和逆元

3. **Python**:

- 优势：内置大整数支持，语法简洁
- 劣势：对于大规模计算效率较低
- 技巧：使用列表推导式和生成器提高效率，利用 math 模块的组合函数

进阶技巧

1. **生成函数结合**：二项式反演与生成函数结合可以解决更复杂的组合计数问题
2. **多项式乘法优化**：对于某些问题，可以使用快速傅里叶变换(FFT)加速多项式乘法
3. **莫比乌斯反演结合**：在数论问题中，二项式反演常与莫比乌斯反演结合使用
4. **动态规划状态优化**：利用二项式反演优化动态规划的状态转移方程
5. **矩阵快速幂加速**：对于线性递推关系，可以使用矩阵快速幂加速计算

学习资源推荐

1. **经典教材**：《组合数学》(Richard A. Brualdi)
2. **在线资源**：OI Wiki、知乎专栏、各类算法竞赛博客
3. **练习平台**：洛谷、Codeforces、AtCoder 等
4. **进阶内容**：生成函数、容斥原理、莫比乌斯反演等相关知识

[代码文件]

文件：Code01_Derangement.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>
#include <stdexcept>

/***
 * 错排问题 (Derangement Problem)
 * 题目：洛谷 P1595 信封问题
 * 链接：https://www.luogu.com.cn/problem/P1595
 * 描述：n 个人写信，求所有人都没有收到自己信的方案数
 *
 * C++实现特点：
 * - 使用 constexpr 和静态数组提升性能
 * - 提供多种计算方法：动态规划、二项式反演、空间优化版本
```

```
* - 详细的时间和空间复杂度分析
* - 完整的异常处理和边界条件检查
* - 包含性能测试和单元测试
*/
```

```
using namespace std;
using namespace chrono;

const long long MOD = 1000000007; // 模数

/***
 * 快速幂算法 - 计算 base^exponent % mod
 *
 * 时间复杂度: O(log exponent)
 * 空间复杂度: O(1)
 *
 * @param base 底数
 * @param exponent 指数
 * @param mod 模数
 * @return 计算结果
*/
long long quick_pow(long long base, long long exponent, long long mod) {
    long long result = 1;
    base %= mod;

    while (exponent > 0) {
        if (exponent & 1) { // 如果当前二进制位为 1
            result = (result * base) % mod;
        }
        base = (base * base) % mod; // 底数自乘
        exponent >>= 1; // 右移一位
    }

    return result;
}

/***
 * 计算错排数 - 动态规划方法
 *
 * 算法原理: 使用递推关系式 D(n) = (n-1) * (D(n-1) + D(n-2))
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
*/
```

```

*
* @param n 元素个数
* @return n 个元素的错排数，对 MOD 取模
* @throws invalid_argument 当 n 为负数时抛出异常
*/
long long derangement_dp(int n) {
    if (n < 0) {
        throw invalid_argument("输入参数必须是非负整数");
    }

    // 边界条件
    if (n == 0) return 1; // 空排列视为一种错排
    if (n == 1) return 0; // 1 个元素不可能错排

    // 动态规划数组
    vector<long long> dp(n + 1, 0);
    dp[0] = 1;
    dp[1] = 0;

    // 递推计算
    for (int i = 2; i <= n; ++i) {
        dp[i] = (i - 1) * ((dp[i - 1] + dp[i - 2]) % MOD) % MOD;
    }

    return dp[n];
}

/***
* 计算错排数 - 二项式反演方法
*
* 算法原理：使用公式  $D(n) = n! * \sum_{i=0}^n (-1)^i / i!$ 
*
* 时间复杂度：O(n)
* 空间复杂度：O(n)
*
* @param n 元素个数
* @return n 个元素的错排数，对 MOD 取模
* @throws invalid_argument 当 n 为负数时抛出异常
*/
long long derangement_inversion(int n) {
    if (n < 0) {
        throw invalid_argument("输入参数必须是非负整数");
    }
}

```

```

// 边界条件
if (n == 0) return 1;
if (n == 1) return 0;

// 预处理阶乘和逆元
vector<long long> fact(n + 1, 1);      // 阶乘数组
vector<long long> inv_fact(n + 1, 1); // 阶乘的逆元数组

// 计算阶乘
for (int i = 1; i <= n; ++i) {
    fact[i] = (fact[i - 1] * i) % MOD;
}

// 使用费马小定理计算逆元
// 因为 MOD 是质数，所以  $(n!)^{-1} \equiv (n!)^{(MOD-2)} \pmod{MOD}$ 
inv_fact[n] = quick_pow(fact[n], MOD - 2, MOD);

// 倒序计算其他阶乘的逆元
for (int i = n - 1; i >= 0; --i) {
    inv_fact[i] = (inv_fact[i + 1] * (i + 1)) % MOD;
}

// 应用二项式反演公式计算错排数
long long result = 0;
for (int k = 0; k <= n; ++k) {
    // 计算符号:  $(-1)^k$ 
    long long sign = (k % 2 == 0) ? 1 : MOD - 1; // -1 mod MOD

    // 计算项:  $n! * (-1)^k / k! = fact[n] * inv_fact[k] * sign$ 
    long long term = (fact[n] * inv_fact[k]) % MOD;
    term = (term * sign) % MOD;

    // 累加结果
    result = (result + term) % MOD;
}

return result;
}

/**
 * 计算错排数 - 空间优化的动态规划方法
 *

```

```

* 算法原理：基于递推式，但只保存前两个状态
*
* 时间复杂度：O(n)
* 空间复杂度：O(1)
*
* @param n 元素个数
* @return n 个元素的错排数，对 MOD 取模
* @throws invalid_argument 当 n 为负数时抛出异常
*/
long long derangement_optimized(int n) {
    if (n < 0) {
        throw invalid_argument("输入参数必须是非负整数");
    }

    // 边界条件
    if (n == 0) return 1;
    if (n == 1) return 0;

    // 只保存前两个状态
    long long a = 1; // D(0)
    long long b = 0; // D(1)
    long long res = 0;

    for (int i = 2; i <= n; ++i) {
        res = (i - 1) * ((a + b) % MOD) % MOD;

        // 更新状态
        a = b;
        b = res;
    }

    return res;
}

/***
* 单元测试函数
* 测试不同方法计算错排数的正确性
*/
void run_unit_tests() {
    // 测试用例：已知的错排数结果
    vector<pair<int, long long>> test_cases = {
        {0, 1}, // 0 个元素的错排数为 1
        {1, 0}, // 1 个元素的错排数为 0
    };
}

```

```

{2, 1},    // 2 个元素的错排数为 1
{3, 2},    // 3 个元素的错排数为 2
{4, 9},    // 4 个元素的错排数为 9
{5, 44},   // 5 个元素的错排数为 44
{6, 265}   // 6 个元素的错排数为 265
};

cout << "错排数单元测试: " << endl;
cout << "===== " << endl;
cout << "n 预期结果 动态规划方法 二项式反演方法 空间优化方法 结果" << endl;
cout << "===== " << endl;

bool all_passed = true;

for (const auto& [n, expected] : test_cases) {
    try {
        long long dp_res = derangement_dp(n);
        long long inv_res = derangement_inversion(n);
        long long opt_res = derangement_optimized(n);

        // 小数值时直接比较，大数值时比较模后的值
        bool dp_correct = (n <= 6) && (dp_res == expected);
        bool inv_correct = (n <= 6) && (inv_res == expected);
        bool opt_correct = (n <= 6) && (opt_res == expected);

        bool test_passed = dp_correct && inv_correct && opt_correct;
        all_passed &= test_passed;

        cout << n << " " << expected << " " << dp_res << " " << inv_res << " " << opt_res
        << " ";
        cout << (test_passed ? "√" : "✗") << endl;
    } catch (const exception& e) {
        cout << n << " " << expected << " 异常 异常 异常 ✗" << endl;
        cout << " 错误信息: " << e.what() << endl;
        all_passed = false;
    }
}

cout << "===== " << endl;
cout << "测试结果: " << (all_passed ? "通过" : "失败") << endl << endl;
}

/***

```

```

* 性能测试函数
* 比较不同方法在大规模数据下的性能
*/
void run_performance_tests() {
    cout << "性能测试: " << endl;
    cout << "===== " << endl;

    int large_n = 1000000; // 100 万

    // 测试动态规划方法
    auto start = high_resolution_clock::now();
    long long dp_result = derangement_dp(large_n);
    auto end = high_resolution_clock::now();
    auto dp_duration = duration_cast<milliseconds>(end - start).count();

    // 测试二项式反演方法
    start = high_resolution_clock::now();
    long long inv_result = derangement_inversion(large_n);
    end = high_resolution_clock::now();
    auto inv_duration = duration_cast<milliseconds>(end - start).count();

    // 测试空间优化方法
    start = high_resolution_clock::now();
    long long opt_result = derangement_optimized(large_n);
    end = high_resolution_clock::now();
    auto opt_duration = duration_cast<milliseconds>(end - start).count();

    cout << "动态规划方法 (n=" << large_n << "): 结果 = " << dp_result
        << ", 耗时 = " << dp_duration << " ms" << endl;
    cout << "二项式反演方法 (n=" << large_n << "): 结果 = " << inv_result
        << ", 耗时 = " << inv_duration << " ms" << endl;
    cout << "空间优化方法 (n=" << large_n << "): 结果 = " << opt_result
        << ", 耗时 = " << opt_duration << " ms" << endl;

    cout << "===== " << endl;
}

/**
 * 边界情况测试
*/
void run_edge_case_tests() {
    cout << "边界情况测试: " << endl;
    try {

```

```

        cout << "测试负数输入 n=-1: " << endl;
        derangement_dp(-1);
        cout << " 错误: 未捕获到异常" << endl;
    } catch (const invalid_argument& e) {
        cout << " 成功: 捕获到异常 - " << e.what() << endl;
    }
    cout << endl;
}

/***
 * 算法复杂度分析函数
 */
void analyze_complexity() {
    cout << "算法复杂度分析: " << endl;
    cout << "===== " << endl;
    cout << "动态规划方法: " << endl;
    cout << " 时间复杂度: O(n) - 需要 n 次迭代计算每个位置的错排数" << endl;
    cout << " 空间复杂度: O(n) - 需要存储大小为 n+1 的 dp 数组" << endl << endl;

    cout << "二项式反演方法: " << endl;
    cout << " 时间复杂度: O(n) - 预处理阶乘和逆元需要 O(n), 计算结果也需要 O(n)" << endl;
    cout << " 空间复杂度: O(n) - 需要存储阶乘和逆元数组" << endl << endl;

    cout << "空间优化方法: " << endl;
    cout << " 时间复杂度: O(n) - 同样需要 n 次迭代" << endl;
    cout << " 空间复杂度: O(1) - 只需要常数级额外空间" << endl << endl;

    cout << "工程化考量: " << endl;
    cout << " 1. 对于大规模数据, 空间优化方法更节省内存" << endl;
    cout << " 2. 对于多次调用, 建议预处理阶乘和逆元以提高效率" << endl;
    cout << " 3. 注意处理大数值时的溢出问题, 使用 long long 类型" << endl;
    cout << " 4. 模运算需要小心处理负数情况" << endl;
    cout << "===== " << endl;
}

/***
 * 主函数
 */
int main() {
    // 运行各种测试
    run_unit_tests();
    run_edge_case_tests();
    run_performance_tests();
}

```

```

analyze_complexity();

// 处理用户输入
try {
    int n;
    cout << "请输入 n: " << endl;
    cin >> n;

    // 计算并输出结果
    long long result = derangement_optimized(n);
    cout << "错排数 D(" << n << ") = " << result << endl;
} catch (const exception& e) {
    cerr << "错误: " << e.what() << endl;
    return 1;
}

return 0;
}

```

=====

文件: Code01_Derangement.java

=====

```

package class145;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

/**
 * 错排问题 (Derangement Problem)
 * 题目: 洛谷 P1595 信封问题
 * 链接: https://www.luogu.com.cn/problem/P1595
 * 描述: n 个人写信, 求所有人都没有收到自己信的方案数
 *
 * 错排问题背景:
 * - 也称为信封问题, 是组合数学中的一个经典问题
 * - 要求找到排列中所有元素都不在原来位置上的排列数
 * - 时间限制: 1 <= n <= 20 (数据范围较小, 可以使用长整型计算)
 */

```

```

public class Code01_Derangement {

    /**
     * 程序主入口
     *
     * @param args 命令行参数
     * @throws IOException 输入输出异常
     */
    public static void main(String[] args) throws IOException {
        // 使用快速输入方式处理输入数据
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取输入的 n 值
        in.nextToken();
        int n = (int) in.nval;

        // 调用错排计算方法并输出结果
        // 两种方法都可以使用，这里使用二项式反演方法
        out.println(ways2(n));

        // 确保输出缓冲区被刷新并关闭资源
        out.flush();
        out.close();
        br.close();
    }

    /**
     * 计算错排数 - 动态规划方法
     *
     * 算法原理：
     * - 使用递推关系式： $D(n) = (n-1) * (D(n-1) + D(n-2))$ 
     * - 推导思路：第 n 个元素必须放在前 n-1 个位置中的某一个位置 k
     *   1. 如果第 k 个元素放在第 n 个位置，则剩下 n-2 个元素需要错排
     *   2. 如果第 k 个元素不放在第 n 个位置，则剩下的问题相当于 n-1 个元素的错排
     *
     * 时间复杂度：O(n) - 只需要一次遍历计算每个位置的错排数
     * 空间复杂度：O(n) - 需要一个长度为 n+1 的数组存储中间结果
     *
     * @param n 元素个数
     * @return n 个元素的错排数
     */
}

```

```

public static long ways1(int n) {
    // 边界条件检查
    if (n <= 0) {
        return 1; // 0 个元素的错排数定义为 1 (空排列)
    }

    // 动态规划数组: dp[i] 表示 i 个元素的错排数
    long[] dp = new long[n + 1];

    // 填充 dp 数组
    for (int i = 1; i <= n; i++) {
        if (i == 1) {
            dp[i] = 0; // 1 个元素无法错排
        } else if (i == 2) {
            dp[i] = 1; // 2 个元素只有一种错排方式: (2, 1)
        } else {
            // 应用递推公式
            dp[i] = (i - 1) * (dp[i - 1] + dp[i - 2]);
        }
    }

    return dp[n];
}

/***
 * 计算错排数 - 二项式反演/容斥原理方法
 *
 * 算法原理:
 * - 使用容斥原理计算:  $D(n) = n! * \sum_{i=0 \text{ to } n} ((-1)^i / i!)$ 
 * - 推导思路: 计算所有排列数减去至少有一个元素在原位置的排列数
 *   加上至少有两个元素在原位置的排列数, 依此类推 (容斥原理)
 *
 * 时间复杂度: O(n) - 计算阶乘和累加各项都只需要 O(n) 时间
 * 空间复杂度: O(1) - 只需要常数级额外空间
 *
 * @param n 元素个数
 * @return n 个元素的错排数
 */
public static long ways2(int n) {
    // 边界条件检查
    if (n <= 0) {
        return 1; // 0 个元素的错排数定义为 1
    }
}

```

```

// 计算 n 的阶乘
long facn = 1; // n!
for (int i = 1; i <= n; i++) {
    facn *= i;
}

// 初始化为 i=0 时的项，即 n!
long ans = facn;

// 计算 i! 并逐项累加
long faci = 1; // i!
for (int i = 1; i <= n; i++) {
    // 计算 i 的阶乘
    faci = faci * i;

    // 计算项: (-1)^i * (n! / i!)
    if ((i & 1) == 0) {
        // i 为偶数, (-1)^i = 1, 加上该值
        ans += facn / faci;
    } else {
        // i 为奇数, (-1)^i = -1, 减去该值
        ans -= facn / faci;
    }
}

return ans;
}

/***
 * 计算错排数 - 空间优化的动态规划方法
 *
 * 算法原理: 基于递推式, 但只保存前两个状态
 * 时间复杂度: O(n)
 * 空间复杂度: O(1) - 只需要常数级额外空间
 *
 * 工程化考虑:
 * - 当 n 较大时, 该方法比 ways1 更节省内存
 * - 但在本题数据范围 (n<=20) 内, 内存优化效果不明显
 *
 * @param n 元素个数
 * @return n 个元素的错排数
 */

```

```

public static long ways3(int n) {
    // 边界条件处理
    if (n <= 0) return 1;
    if (n == 1) return 0;
    if (n == 2) return 1;

    // 只保存前两个状态
    long prev1 = 1; // D(2)
    long prev2 = 0; // D(1)
    long curr = 0; // 当前计算的 D(n)

    // 从 3 开始递推计算
    for (int i = 3; i <= n; i++) {
        curr = (i - 1) * (prev1 + prev2);
        // 更新状态
        prev2 = prev1;
        prev1 = curr;
    }

    return curr;
}

```

文件: Code01_Derangement.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""
错排问题 (Derangement Problem)

题目: 洛谷 P1595 信封问题

链接: <https://www.luogu.com.cn/problem/P1595>

描述: n 个人写信, 求所有人都没有收到自己信的方案数

二项式反演应用: 将“至少 k 个元素在原位置”转化为“恰好 0 个元素在原位置”的问题

Python 实现特点:

- 提供多种计算方法: 动态规划、二项式反演、空间优化版本
- 包含详细的复杂度分析和注释
- 提供单元测试和性能测试
- 处理边界情况和异常

"""

```
MOD = 10**9 + 7 # 模数
```

```
def derangement_dp(n: int) -> int:
```

"""

计算错排数 - 动态规划方法

算法原理:

使用递推关系式: $D(n) = (n-1) * (D(n-1) + D(n-2))$

时间复杂度: $O(n)$ - 只需要一次遍历计算每个位置的错排数

空间复杂度: $O(n)$ - 需要一个长度为 $n+1$ 的数组存储中间结果

Args:

n: 元素个数

Returns:

n 个元素的错排数, 结果对 MOD 取模

Raises:

ValueError: 当 n 为负数时抛出异常

"""

异常处理: 输入检查

```
if n < 0:
```

```
    raise ValueError("输入必须是非负整数")
```

边界条件处理

```
if n == 0:
```

```
    return 1 # 0 个元素的错排数为 1 (空排列)
```

```
if n == 1:
```

```
    return 0 # 1 个元素的错排数为 0
```

动态规划数组: dp[i] 表示 i 个元素的错排数

```
dp = [0] * (n + 1)
```

```
dp[0] = 1
```

```
dp[1] = 0
```

```
for i in range(2, n + 1):
```

递推公式: $D(n) = (n-1) * (D(n-1) + D(n-2))$

```
    dp[i] = (i - 1) * (dp[i - 1] + dp[i - 2]) % MOD
```

```
return dp[n]
```

```

def derangement_inversion(n: int) -> int:
    """
    计算错排数 - 二项式反演方法

    算法原理:
    使用二项式反演公式: D(n) = n! * Σ (i=0 到 n) (-1)^i / i!

    时间复杂度: O(n) - 预处理阶乘和逆元需要 O(n), 计算结果也需要 O(n)
    空间复杂度: O(n) - 需要存储阶乘和逆元数组

    Args:
        n: 元素个数

    Returns:
        n 个元素的错排数, 结果对 MOD 取模

    Raises:
        ValueError: 当 n 为负数时抛出异常
    """
    # 异常处理: 输入检查
    if n < 0:
        raise ValueError("输入必须是非负整数")

    # 边界条件处理
    if n == 0:
        return 1
    if n == 1:
        return 0

    # 预处理阶乘和逆元
    fact = [1] * (n + 1)  # fact[i] = i! mod MOD
    inv = [1] * (n + 1)   # inv[i] = (i!)^(-1) mod MOD

    # 计算阶乘数组
    for i in range(1, n + 1):
        fact[i] = fact[i - 1] * i % MOD

    # 使用费马小定理计算逆元
    # 因为 MOD 是质数, 所以 (i!)^(-1) ≡ (i!)^(MOD-2) mod MOD
    inv[n] = pow(fact[n], MOD - 2, MOD)
    for i in range(n - 1, -1, -1):
        inv[i] = inv[i + 1] * (i + 1) % MOD

```

```

# 使用二项式反演公式计算错排数
res = 0
for k in range(n + 1):
    # 计算符号: (-1)^k
    sign = 1 if k % 2 == 0 else -1

    # 计算项: n! * (-1)^k / k! = fact[n] * inv[k] * sign
    term = fact[n] * inv[k] % MOD

    # 处理负数情况
    if sign < 0:
        term = (MOD - term) % MOD

    # 累加结果
    res = (res + term) % MOD

return res

```

```
def derangement_optimized(n: int) -> int:
```

```
"""

```

计算错排数 - 空间优化的动态规划方法

算法原理：基于递推式，但只保存前两个状态

时间复杂度：O(n)

空间复杂度：O(1) - 只需要常数级额外空间

Args:

n: 元素个数

Returns:

n 个元素的错排数，结果对 MOD 取模

Raises:

ValueError: 当 n 为负数时抛出异常

```
"""

```

```
if n < 0:
    raise ValueError("输入必须是非负整数")

```

```
if n == 0:
    return 1

```

```
if n == 1:
    return 0

```

```

# 只需要保存前两个状态
a = 1 # dp[0]
b = 0 # dp[1]
res = 0

for i in range(2, n + 1):
    res = (i - 1) * (a + b) % MOD

    # 更新状态
    a, b = b, res

return res

def run_tests():
    """
    单元测试函数
    测试不同方法计算错排数的正确性
    """

    # 测试用例: 已知的错排数结果
    test_cases = [
        (0, 1),    # 0 个元素的错排数为 1
        (1, 0),    # 1 个元素的错排数为 0
        (2, 1),    # 2 个元素的错排数为 1
        (3, 2),    # 3 个元素的错排数为 2
        (4, 9),    # 4 个元素的错排数为 9
        (5, 44),   # 5 个元素的错排数为 44
        (6, 265)   # 6 个元素的错排数为 265
    ]

    print("错排数测试: ")
    print("=" * 70)
    print(f"{'n':<5} {'预期结果':<15} {'动态规划方法':<25} {'二项式反演方法':<25}")
    print("=" * 70)

    all_passed = True

    for n, expected in test_cases:
        try:
            dp_res = derangement_dp(n)
            inv_res = derangement_inversion(n)
            opt_res = derangement_optimized(n)

            if dp_res != expected or inv_res != expected or opt_res != expected:
                print(f"n={n} | DP: {dp_res}, Inv: {inv_res}, Opt: {opt_res} | Expected: {expected}")
                all_passed = False
        except Exception as e:
            print(f"Error for n={n}: {e}")
            all_passed = False

```

```

# 由于取模的原因，我们需要在小数值时比较实际值，大数值时比较模后的值
dp_correct = (n <= 6) and (dp_res == expected)
inv_correct = (n <= 6) and (inv_res == expected)
opt_correct = (n <= 6) and (opt_res == expected)

test_passed = dp_correct and inv_correct and opt_correct
all_passed &= test_passed

print(f"{{n:<5} {{expected:<15} {{dp_res:<25} {{inv_res:<25} {{'✓' if test_passed else '✗'}}}}}")
except Exception as e:
    print(f"{{n:<5} {{expected:<15} {{'异常':<25} {{'异常':<25} {{'✗'}}}}}")
    print(f"  错误信息: {e}")
    all_passed = False

print("=" * 70)
print(f"测试{{'通过' if all_passed else '失败'}}")
print()

# 性能测试
print("性能测试: ")
print("=" * 70)

import time

large_n = 100000

# 测试动态规划方法
start_time = time.time()
dp_res = derangement_dp(large_n)
dp_time = (time.time() - start_time) * 1000 # 转换为毫秒
print(f"动态规划方法 (n={large_n}): {dp_res} (耗时: {dp_time:.2f}ms)")

# 测试二项式反演方法
start_time = time.time()
inv_res = derangement_inversion(large_n)
inv_time = (time.time() - start_time) * 1000
print(f"二项式反演方法 (n={large_n}): {inv_res} (耗时: {inv_time:.2f}ms)")

# 测试空间优化方法
start_time = time.time()
opt_res = derangement_optimized(large_n)
opt_time = (time.time() - start_time) * 1000
print(f"空间优化方法 (n={large_n}): {opt_res} (耗时: {opt_time:.2f}ms)")

```

```
print("=" * 70)

def main():
    """
    主函数，读取输入并计算错排数
    """

    # 运行单元测试
    run_tests()

    # 边界情况测试
    print("边界情况测试：")
    try:
        print("n=-1: 异常测试")
        derangement_dp(-1)
    except ValueError as e:
        print(f"捕获到异常: {e}")

    # 读取用户输入
    try:
        n = int(input("请输入 n: "))
        result = derangement_optimized(n)
        print(f"错排数 D({n}) = {result}")
    except ValueError as e:
        print(f"输入错误: {e}")

if __name__ == "__main__":
    main()
```

```
=====
文件: Code02_SetCounting.cpp
=====

#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>
#include <chrono>
#include <tuple>

/***
 * 集合计数问题 (Set Counting Problem)
 * 题目: 洛谷 P10596 集合计数 / BZOJ2839 集合计数
```

- * 链接: <https://www.luogu.com.cn/problem/P10596>
- * 描述: 从 2^n 个子集中选出若干个集合, 使交集恰好包含 k 个元素的方案数
- *
- * C++实现特点:
- * - 使用类封装问题求解
- * - 预处理阶乘和逆元
- * - 二项式反演算法
- * - 详细的异常处理和性能分析
- */

```
using namespace std;
using namespace chrono;

const long long MOD = 1000000007; // 模数

class SetCounting {
private:
    int n; // 元素总数
    int k; // 交集恰好包含的元素个数
    vector<long long> fact; // 阶乘数组
    vector<long long> inv_fact; // 阶乘逆元数组

    /**
     * 快速幂算法 - 计算 base^exponent % mod
     *
     * 时间复杂度: O(log exponent)
     * 空间复杂度: O(1)
     *
     * @param base 底数
     * @param exponent 指数
     * @param mod 模数
     * @return 计算结果
     */
    long long quick_pow(long long base, long long exponent, long long mod) {
        long long result = 1;
        base %= mod;

        while (exponent > 0) {
            if (exponent & 1) { // 如果当前二进制位为 1
                result = (result * base) % mod;
            }
            base = (base * base) % mod; // 底数自乘
            exponent >>= 1; // 右移一位
        }
        return result;
    }

    long long choose(int n, int k) {
        if (n < k) return 0;
        if (k == 0) return 1;
        return fact[n] * inv_fact[k] % MOD * inv_fact[n - k] % MOD;
    }

    long long calculate() {
        long long ans = 1;
        for (int i = 0; i < k; i++) {
            ans *= choose(n, i);
            ans %= MOD;
        }
        return ans;
    }
};
```

```

    }

    return result;
}

/***
 * 预处理阶乘和阶乘的逆元
 * 时间复杂度: O(max_n)
 * 空间复杂度: O(max_n)
 *
 * @param max_n 预处理的最大阶乘值
 */
void precompute(int max_n) {
    fact.resize(max_n + 1);
    inv_fact.resize(max_n + 1);

    // 计算阶乘
    fact[0] = 1;
    for (int i = 1; i <= max_n; ++i) {
        fact[i] = (fact[i - 1] * i) % MOD;
    }

    // 计算最大阶乘的逆元
    inv_fact[max_n] = quick_pow(fact[max_n], MOD - 2, MOD);

    // 倒序计算其他阶乘的逆元
    for (int i = max_n - 1; i >= 0; --i) {
        inv_fact[i] = (inv_fact[i + 1] * (i + 1)) % MOD;
    }
}

public:
    /**
     * 构造函数
     *
     * @param n 元素总数
     * @param k 交集恰好包含的元素个数
     *
     * @throws invalid_argument 当参数无效时抛出异常
     */
    SetCounting(int n, int k) {
        // 参数验证
        if (n < 0) {

```

```

        throw invalid_argument("参数 n 必须是非负整数");
    }

    if (k < 0 || k > n) {
        throw invalid_argument("参数 k 必须在 0 到 n 之间");
    }

    this->n = n;
    this->k = k;

    // 预处理阶乘和逆元
    int max_needed = max(n, 1); // 确保至少为 1
    precompute(max_needed);
}

/***
 * 计算组合数 C(a, b) = a!/(b! * (a-b)!)
 * 时间复杂度: O(1) - 使用预处理的阶乘和逆元
 * 空间复杂度: O(1)
 *
 * @param a 总数
 * @param b 选取的数量
 * @return 组合数 C(a, b) 对 MOD 取模的结果
 */
long long comb(int a, int b) {
    if (b < 0 || b > a) {
        return 0;
    }
    if (a == 0 && b == 0) {
        return 1;
    }

    // 确保 a 在预处理范围内
    if (a >= fact.size()) {
        // 如果 a 大于预处理的范围, 重新预处理
        precompute(a);
    }

    return (fact[a] * inv_fact[b] % MOD) * inv_fact[a - b] % MOD;
}

/***
 * 计算集合计数问题的解
 *

```

```

* 算法原理:
* 1. 定义 f(k) 为交集恰好有 k 个元素的方案数
* 2. 定义 g(k) 为交集至少有 k 个元素的方案数
* 3. 通过二项式反演,  $f(k) = \sum_{i=k}^n [(-1)^{i-k} * C(i, k) * g(i)]$ 
* 4.  $g(i) = C(n, i) * (2^{(2^{(n-i)})} - 1)$  表示选择 i 个固定元素, 其余元素任意组合
*
* 时间复杂度分析:  $O(n \log \max\_pow)$  - 计算 2 的高次幂需要  $O(\log \max\_pow)$  时间
* 空间复杂度分析:  $O(n)$  - 需要存储阶乘和逆元数组
*
* @return 交集恰好有 k 个元素的方案数, 对 MOD 取模
*/
long long compute() {
    // 边界情况处理
    if (k == 0 && n == 0) {
        return 0;
    }

    // 计算选择 k 个元素作为交集的方案数: C(n, k)
    long long c_n_k = comb(n, k);

    // 计算剩下的 m = n - k 个元素的可能组合
    int m = n - k;

    // 如果 m 为 0, 那么只有一种可能
    if (m == 0) {
        return c_n_k * 1 % MOD;
    }

    // 结果初始化为 0
    long long result = 0;

    // 应用二项式反演公式, 计算 f(k)
    for (int i = 0; i <= m; ++i) {
        // 计算 C(m, i)
        long long c_m_i = comb(m, i);

        // 计算符号  $(-1)^i$ 
        long long sign = (i % 2 == 0) ? 1 : MOD - 1; // -1 mod MOD

        // 计算  $2^{(2^{(m-i)})} \bmod \text{MOD}$ 
        // 使用费马小定理简化计算, 因为 MOD 是质数
        long long exponent = quick_pow(2, m - i, MOD - 1);
        long long power_val = quick_pow(2, exponent, MOD);
    }
}

```

```

// 计算项: C(m, i) * (-1)^i * (2^(2^(m-i)) - 1)
long long term = (c_m_i * ((power_val - 1 + MOD) % MOD)) % MOD;
term = (term * sign) % MOD;

// 累加到结果中
result = (result + term) % MOD;
}

// 最终结果: C(n, k) * f(k)
return (c_n_k * result) % MOD;
}

/***
* 运行测试用例
*
* @return bool 所有测试通过返回 true, 否则返回 false
*/
bool test() {
    // 测试用例
    vector<tuple<int, int, long long>> test_cases = {
        {3, 1, 9}, // n=3, k=1, 期望结果 9
        {4, 2, 18}, // n=4, k=2, 期望结果 18
        {1, 0, 1}, // n=1, k=0, 期望结果 1
        {1, 1, 1}, // n=1, k=1, 期望结果 1
        {0, 0, 0} // n=0, k=0, 期望结果 0
    };

    bool all_passed = true;

    cout << "集合计数问题测试: " << endl;
    cout << "===== " << endl;
    cout << "n  k  预期结果  实际结果  状态" << endl;
    cout << "===== " << endl;

    for (const auto& test_case : test_cases) {
        int test_n = get<0>(test_case);
        int test_k = get<1>(test_case);
        long long expected = get<2>(test_case);

        try {
            SetCounting problem(test_n, test_k);
            long long actual = problem.compute();

```

```

        bool passed = (actual == expected);
        all_passed &= passed;

        cout << test_n << " " << test_k << " " << expected << " " << actual << " "
            << (passed ? "/" : "X") << endl;
    } catch (const exception& e) {
        cout << test_n << " " << test_k << " " << expected << " 异常 X" << endl;
        cout << " 错误信息: " << e.what() << endl;
        all_passed = false;
    }
}

cout << "===== " << endl;
cout << "测试结果: " << (all_passed ? "通过" : "失败") << endl << endl;

return all_passed;
}

};

/***
 * 算法复杂度分析函数
 */
void analyze_complexity() {
    cout << "算法复杂度分析: " << endl;
    cout << "===== " << endl;
    cout << "时间复杂度: " << endl;
    cout << " - 预处理阶乘和逆元: O(n)" << endl;
    cout << " - 计算结果: O(n log(2^(n-k))) = O(n^2) 最坏情况下" << endl;
    cout << " - 组合数计算: O(1) 每次查询" << endl << endl;

    cout << "空间复杂度: " << endl;
    cout << " - 阶乘和逆元数组: O(n)" << endl;
    cout << " - 其他变量: O(1)" << endl << endl;

    cout << "优化点: " << endl;
    cout << " 1. 预处理阶乘和逆元以加速组合数计算" << endl;
    cout << " 2. 使用费马小定理计算大指数的模幂" << endl;
    cout << " 3. 处理负数模运算时的溢出问题" << endl;
    cout << "===== " << endl;
}

/***
 * 主函数

```

```
/*
int main() {
    // 运行测试
    try {
        SetCounting problem(1, 1);
        problem.test();
    } catch (const exception& e) {
        cerr << "测试失败: " << e.what() << endl;
    }

    // 边界情况测试
    cout << "边界情况测试: " << endl;
    try {
        cout << "测试负数输入 n=-1: " << endl;
        SetCounting problem1(-1, 0);
        cout << " 错误: 未捕获到异常" << endl;
    } catch (const invalid_argument& e) {
        cout << " 成功: 捕获到异常 - " << e.what() << endl;
    }

    try {
        cout << "测试 k > n: " << endl;
        SetCounting problem2(5, 6);
        cout << " 错误: 未捕获到异常" << endl;
    } catch (const invalid_argument& e) {
        cout << " 成功: 捕获到异常 - " << e.what() << endl;
    }
    cout << endl;

    // 分析复杂度
    analyze_complexity();

    // 性能测试
    cout << "\n 性能测试: " << endl;
    cout << "===== " << endl;

    try {
        // 测试较大的数据规模
        int test_n = 1000;
        int test_k = 500;

        auto start = high_resolution_clock::now();
        SetCounting large_problem(test_n, test_k);
    }
```

```

long long result = large_problem.compute();
auto end = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(end - start).count();

cout << "n = " << test_n << ", k = " << test_k << endl;
cout << "结果 = " << result << endl;
cout << "计算时间 = " << duration << " ms" << endl;
} catch (const exception& e) {
    cerr << "性能测试失败: " << e.what() << endl;
}
cout << "===== " << endl;

// 处理用户输入
try {
    int n, k;
    cout << "\n请输入 n 和 k: " << endl;
    cout << "n = ";
    cin >> n;
    cout << "k = ";
    cin >> k;

    SetCounting user_problem(n, k);
    long long user_result = user_problem.compute();
    cout << "从 2^" << n << "个子集中选出若干个集合，使交集恰好包含" << k
        << "个元素的方案数为: " << user_result << endl;
}

} catch (const exception& e) {
    cerr << "错误: " << e.what() << endl;
    return 1;
}

return 0;
}
=====
```

文件: Code02_SetCounting.java

```

package class145;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

/***
 * 集合计数问题
 * 题目: 洛谷 P10596 集合计数 / BZOJ2839 集合计数
 * 链接: https://www.luogu.com.cn/problem/P10596
 * 描述: 从  $2^n$  个子集中选出若干个集合, 使交集恰好包含 k 个元素的方案数
 *
 * 数据范围:
 * -  $1 \leq n \leq 10^6$ 
 * -  $0 \leq k \leq n$ 
 * - 结果对 1000000007 取模
 *
 * 二项式反演应用: 将"恰好 k 个元素"转化为"至少 k 个元素"的问题
 */
public class Code02_SetCounting {

    // 最大数据范围, 根据题目要求设置为  $10^{6+1}$ 
    public static int MAXN = 1000001;

    // 模数, 题目要求结果对  $1e9+7$  取模
    public static int MOD = 1000000007;

    // 阶乘数组, 用于计算组合数
    public static long[] fac = new long[MAXN];

    // 阶乘的逆元数组, 用于计算组合数
    public static long[] inv = new long[MAXN];

    // g[i] 表示选出若干集合, 使得交集至少包含 i 个元素的方案数
    public static long[] g = new long[MAXN];

    // 输入参数: n 是元素个数, k 是目标交集大小
    public static int n, k;

    /**
     * 预处理阶乘和阶乘的逆元
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     *
     * 工程化考虑:
    
```

```

* - 使用递推方式计算阶乘，避免重复计算
* - 使用费马小定理计算逆元，因为 MOD 是质数
* - 逆元采用倒序计算，提高效率
*/
public static void build() {
    // 初始化阶乘数组
    fac[0] = inv[0] = 1;
    fac[1] = 1;

    // 计算阶乘: fac[i] = i! mod MOD
    for (int i = 2; i <= n; i++) {
        fac[i] = ((long) i * fac[i - 1]) % MOD;
    }

    // 使用费马小定理计算最大 n 的阶乘逆元
    // 费马小定理: 当 MOD 是质数时,  $a^{(MOD-1)} \equiv 1 \pmod{MOD}$ , 因此  $a^{(MOD-2)} \equiv a^{-1} \pmod{MOD}$ 
    inv[n] = power(fac[n], MOD - 2);

    // 倒序计算其他阶乘的逆元
    // 利用性质: inv[i] = inv[i+1] * (i+1) mod MOD
    for (int i = n - 1; i >= 1; i--) {
        inv[i] = ((long) (i + 1) * inv[i + 1]) % MOD;
    }
}

/**
 * 快速幂运算，计算  $x^p \% MOD$ 
 *
 * 时间复杂度:  $O(\log p)$  - 二进制快速幂算法
 * 空间复杂度:  $O(1)$  - 只需要常数级额外空间
 *
 * @param x 底数
 * @param p 指数
 * @return  $x^p \% MOD$ 
*/
public static long power(long x, long p) {
    long ans = 1;
    x %= MOD; // 先取模避免溢出

    while (p > 0) {
        // 如果当前二进制位为 1，则乘上当前的  $x^{2^i}$ 
        if ((p & 1) == 1) {

```

```

        ans = (ans * x) % MOD;
    }
    // x 自乘, 相当于  $x^{(2^{(i+1)})}$ 
    x = (x * x) % MOD;
    // p 右移一位, 处理下一个二进制位
    p >>= 1;
}
return ans;
}


$$C(n, k) = \frac{n!}{k!(n-k)!}$$


* 时间复杂度: O(1) - 直接利用预处理的阶乘和逆元计算
* 空间复杂度: O(1)

*
* @param n 总数
* @param k 选取的数量
* @return C(n, k) mod MOD
* @throws IllegalArgumentException 当 k < 0 或 k > n 时抛出异常
*/
public static long c(int n, int k) {
    // 边界条件检查
    if (k < 0 || k > n) {
        return 0; // C(n, k)=0 当 k<0 或 k>n
    }
    // 利用预处理的阶乘和逆元计算组合数
    return (((fac[n] * inv[k]) % MOD) * inv[n - k]) % MOD;
}


$$f(k) = \sum_{i=k}^n [(-1)^{i-k} * C(i, k) * g(i)]$$

* 1. 定义  $f(k)$  为交集恰好有  $k$  个元素的方案数
* 2. 定义  $g(k)$  为交集至少有  $k$  个元素的方案数
* 3. 通过二项式反演,  $f(k) = \sum_{i=k}^n [(-1)^{i-k} * C(i, k) * g(i)]$ 
* 4.  $g(i) = C(n, i) * (2^{(2^{(n-i)})} - 1)$  表示选择  $i$  个固定元素, 其余元素任意组合
*
* 时间复杂度分析:
* - 预处理阶乘和逆元: O(n)
* - 计算  $g$  数组: O(n)
* - 计算最终答案: O(n)

```

```

* 总时间复杂度: O(n)
*
* 空间复杂度分析:
* - 阶乘和逆元数组: O(n)
* - g 数组: O(n)
* 总空间复杂度: O(n)
*
* @return 交集恰好有 k 个元素的方案数, 对 MOD 取模
*/
public static long compute() {
    // 预处理阶乘和逆元
    build();

    // 计算 g 数组
    // 注意: 这里采用逆序计算, 从 i=n 开始, 这样可以高效计算  $2^{(2^{(n-i)})}$ 
    long tmp = 2; // 初始值:  $2^{(2^0)} = 2^1 = 2$ 
    for (int i = n; i >= 0; i--) {
        g[i] = tmp; // g[i]暂时保存  $2^{(2^{(n-i)})}$ 
        // 计算下一个 tmp =  $2^{(2^{(n-i+1)})} = (2^{(2^{(n-i)})})^2$ 
        tmp = tmp * tmp % MOD;
    }

    // 计算完整的 g[i] = C(n, i) *  $(2^{(2^{(n-i)})} - 1)$ 
    // 注意: 减去 1 相当于加上 MOD-1 (模运算中的负数处理)
    for (int i = 0; i <= n; i++) {
        g[i] = (g[i] + MOD - 1) * c(n, i) % MOD;
    }

    // 应用二项式反演公式计算 f(k)
    long ans = 0;
    for (int i = k; i <= n; i++) {
        // 计算符号:  $(-1)^{(i-k)}$ 
        if (((i - k) & 1) == 0) {
            // 偶数次幂, 符号为正
            ans = (ans + c(i, k) * g[i] % MOD) % MOD;
        } else {
            // 奇数次幂, 符号为负, 相当于乘以 MOD-1
            ans = (ans + c(i, k) * g[i] % MOD * (MOD - 1) % MOD) % MOD;
        }
    }

    return ans;
}

```

```
/**  
 * 主函数，处理输入输出  
 *  
 * @param args 命令行参数  
 * @throws IOException 输入输出异常  
 */  
public static void main(String[] args) throws IOException {  
    // 使用快速输入方式  
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
    StreamTokenizer in = new StreamTokenizer(br);  
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));  
  
    // 读取输入参数 n 和 k  
    in.nextToken();  
    n = (int) in.nval;  
    in.nextToken();  
    k = (int) in.nval;  
  
    // 验证输入参数的合法性  
    if (n < 0 || k < 0 || k > n) {  
        throw new IllegalArgumentException("输入参数不合法：n 应大于等于 0, k 应在 0 到 n 之间");  
    }  
  
    // 调用计算函数并输出结果  
    out.println(compute());  
  
    // 确保输出被刷新并关闭资源  
    out.flush();  
    out.close();  
    br.close();  
}  
  
/**  
 * 单元测试函数（用于验证算法正确性）  
 *  
 * 工程化考虑：  
 * - 测试边界条件  
 * - 测试小规模输入  
 * - 与已知结果对比  
 */  
private static void test() {  
    // 测试用例 1: n=1, k=0
```

```

n = 1; k = 0;
long result1 = compute();
System.out.printf("n=%d, k=%d, 结果=%d (期望: 1)\n", n, k, result1);

// 测试用例 2: n=2, k=1
n = 2; k = 1;
long result2 = compute();
System.out.printf("n=%d, k=%d, 结果=%d (期望: 2)\n", n, k, result2);

// 测试用例 3: n=3, k=1
n = 3; k = 1;
long result3 = compute();
System.out.printf("n=%d, k=%d, 结果=%d (期望: 12)\n", n, k, result3);
}

}

```

文件: Code02_SetCounting.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

集合计数问题 (Set Counting Problem)

题目: 洛谷 P10596 集合计数 / BZOJ2839 集合计数

链接: <https://www.luogu.com.cn/problem/P10596>

描述: 从 2^n 个子集中选出若干个集合, 使交集恰好包含 k 个元素的方案数

Python 实现特点:

- 使用二项式反演将“恰好 k 个元素”转换为“至少 k 个元素”问题
- 预处理阶乘和逆元以优化组合数计算
- 包含详细的复杂度分析和代码注释
- 提供完整的异常处理和测试用例

"""

```
MOD = 10**9 + 7 # 模数
```

```
class SetCounting:
```

```
    def __init__(self, n, k):
        """
```

初始化集合计数问题

Args:

n: 元素总数

k: 交集恰好包含的元素个数

Raises:

ValueError: 当参数无效时抛出异常

"""

参数验证

if not isinstance(n, int) or not isinstance(k, int):

 raise ValueError("参数 n 和 k 必须是整数")

if n < 0:

 raise ValueError("参数 n 必须是非负整数")

if k < 0 or k > n:

 raise ValueError("参数 k 必须在 0 到 n 之间")

self.n = n

self.k = k

self.fact = [] # 阶乘数组

self.inv_fact = [] # 阶乘逆元数组

预处理阶乘和逆元

self.precompute()

def precompute(self):

"""

预处理阶乘和阶乘的逆元

时间复杂度: O(n)

空间复杂度: O(n)

"""

为避免重复计算, 确保 n 至少为 1

max_n = max(1, self.n)

self.fact = [1] * (max_n + 1)

self.inv_fact = [1] * (max_n + 1)

计算阶乘

for i in range(1, max_n + 1):

 self.fact[i] = (self.fact[i-1] * i) % MOD

使用费马小定理计算逆元

self.inv_fact[max_n] = pow(self.fact[max_n], MOD-2, MOD)

倒序计算其他阶乘的逆元

```

for i in range(max_n-1, -1, -1):
    self.inv_fact[i] = (self.inv_fact[i+1] * (i+1)) % MOD

def comb(self, a, b):
    """
    计算组合数 C(a, b) = a!/(b! * (a-b)!)
    时间复杂度: O(1) - 使用预处理的阶乘和逆元

    Args:
        a: 总数
        b: 选取的数量

    Returns:
        组合数 C(a, b)对 MOD 取模的结果
    """
    if b < 0 or b > a:
        return 0
    if a == 0 and b == 0:
        return 1

    # 确保 a 在预处理范围内
    if a >= len(self.fact):
        # 如果 a 大于预处理的范围, 重新预处理
        old_n = len(self.fact) - 1
        max_n = a
        self.fact.extend([1] * (max_n - old_n))
        self.inv_fact.extend([1] * (max_n - old_n))

        for i in range(old_n + 1, max_n + 1):
            self.fact[i] = (self.fact[i-1] * i) % MOD

        self.inv_fact[max_n] = pow(self.fact[max_n], MOD-2, MOD)
        for i in range(max_n-1, old_n, -1):
            self.inv_fact[i] = (self.inv_fact[i+1] * (i+1)) % MOD

    return (self.fact[a] * self.inv_fact[b] % MOD) * self.inv_fact[a - b] % MOD

def compute(self):
    """
    计算集合计数问题的解

```

算法原理:

1. 定义 $f(k)$ 为交集恰好有 k 个元素的方案数

2. 定义 $g(k)$ 为交集至少有 k 个元素的方案数
3. 通过二项式反演, $f(k) = \sum_{i=k}^n [(-1)^{i-k} * C(i, k) * g(i)]$
4. $g(i) = C(n, i) * (2^{(2^{(n-i)}) - 1})$ 表示选择 i 个固定元素, 其余元素任意组合

时间复杂度分析: $O(n \log \max_pow)$ – 计算 2 的高次幂需要 $O(\log \max_pow)$ 时间

空间复杂度分析: $O(n)$ – 需要存储阶乘和逆元数组

Returns:

交集恰好有 k 个元素的方案数, 对 MOD 取模

"""

```
if self.k > self.n:
    return 0
```

预计算 g 数组

```
g = [0] * (self.n + 1)
```

计算 $g[i] = C(n, i) * (2^{(2^{(n-i)}) - 1})$

```
tmp = 1 #  $2^{(2^0)} = 2^1 = 2$ 
```

```
for i in range(self.n, -1, -1):
```

$g[i]$ 暂时保存 $2^{(2^{(n-i)})}$

```
g[i] = tmp
```

计算下一个 $tmp = 2^{(2^{(n-i+1)})} = (2^{(2^{(n-i)})})^2$

```
tmp = tmp * tmp % MOD
```

计算完整的 $g[i] = C(n, i) * (2^{(2^{(n-i)}) - 1})$

```
for i in range(self.n + 1):
```

减去 1 相当于加上 MOD-1 (模运算中的负数处理)

```
g[i] = (g[i] + MOD - 1) * self.comb(self.n, i) % MOD
```

应用二项式反演公式计算 $f(k)$

```
result = 0
```

```
for i in range(self.k, self.n + 1):
```

计算符号: $(-1)^{i-k}$

```
if (i - self.k) % 2 == 0:
```

偶数次幂, 符号为正

```
result = (result + self.comb(i, self.k) * g[i] % MOD) % MOD
```

```
else:
```

奇数次幂, 符号为负, 相当于乘以 MOD-1

```
result = (result + self.comb(i, self.k) * g[i] % MOD * (MOD - 1) % MOD) % MOD
```

```
return result
```

```
def test(self):
```

```
"""
```

运行测试用例

Returns:

bool: 所有测试通过返回 True, 否则返回 False

```
"""
```

测试用例

```
test_cases = [
    (3, 1, 9),  # n=3, k=1, 期望结果 9
    (4, 2, 18), # n=4, k=2, 期望结果 18
    (1, 0, 1),  # n=1, k=0, 期望结果 1
    (1, 1, 1),  # n=1, k=1, 期望结果 1
    (0, 0, 0)   # n=0, k=0, 期望结果 0
]
```

```
all_passed = True
```

```
print("集合计数问题测试: ")
```

```
print("=" * 60)
```

```
print(f"{'n':<5} {'k':<5} {'预期结果':<10} {'实际结果':<10} {'状态'}")
```

```
print("=" * 60)
```

```
for n, k, expected in test_cases:
```

```
    try:
```

```
        problem = SetCounting(n, k)
        actual = problem.compute()
        passed = actual == expected
        all_passed &= passed
```

```
        print(f"{'n':<5} {'k':<5} {expected:<10} {actual:<10} {'✓' if passed else '✗'}")
```

```
    except Exception as e:
```

```
        print(f"{'n':<5} {'k':<5} {expected:<10} {'异常':<10} {'✗'}")
```

```
        print(f"  错误信息: {e}")
```

```
        all_passed = False
```

```
print("=" * 60)
```

```
print(f"测试 {'通过' if all_passed else '失败'}")
```

```
print()
```

```
return all_passed
```

```
def main():
    """
```

```

主函数，处理输入并计算结果
"""

# 运行测试
SetCounting(1, 1).test()

# 边界情况测试
print("边界情况测试: ")
try:
    SetCounting(-1, 0)
    print(" 错误: 未捕获到 n 为负数的异常")
except ValueError as e:
    print(f" 成功: 捕获到 n 为负数的异常 - {e}")

try:
    SetCounting(5, 6)
    print(" 错误: 未捕获到 k 大于 n 的异常")
except ValueError as e:
    print(f" 成功: 捕获到 k 大于 n 的异常 - {e}")

# 处理用户输入
try:
    print("\n请输入 n 和 k: ")
    n = int(input("n = "))
    k = int(input("k = "))

    problem = SetCounting(n, k)
    result = problem.compute()
    print(f"从 2^{n} 个子集中选出若干个集合，使交集恰好包含 {k} 个元素的方案数为: {result}")

except ValueError as e:
    print(f"输入错误: {e}")

if __name__ == "__main__":
    main()

```

```

=====
文件: Code03_DistributeSpecialties.java
=====

package class145;

// 分特产
// 一共有 m 种特产， arr[i] 表示 i 种特产有几个

```

```
// 一共有 n 个同学，每个同学至少要得到一个特产  
// 返回分配特产的方法数，答案对 1000000007 取模  
// 0 <= n、m <= 1000  
// 0 <= arr[i] <= 1000  
// 测试链接 : https://www.luogu.com.cn/problem/P5505  
// 提交以下的 code，提交时请把类名改成"Main"，可以通过所有测试用例
```

```
/*
```

二项式反演在分特产问题中的应用：

问题描述：

有 m 种特产，第 i 种特产有 arr[i] 个。

有 n 个同学，每个同学至少要得到一个特产。

求分配特产的方法数。

解题思路：

设 $f(i)$ 表示恰好有 i 个同学没有分到特产的方案数

设 $g(i)$ 表示至少有 i 个同学没有分到特产的方案数（钦定 i 个同学不分特产）

显然， $g(i)$ 更容易计算：

1. 从 n 个同学中选出 i 个同学不分特产，方案数为 $C(n, i)$
2. 对于每种特产 j ，将 $arr[j]$ 个特产分给剩下的 $(n-i)$ 个同学，这是经典的插板法问题
方案数为 $C(arr[j] + (n-i) - 1, (n-i) - 1) = C(arr[j] + n - i - 1, n - i - 1)$
3. 所有特产的分配方案相乘得到总方案数

因此： $g(i) = C(n, i) * \prod_{j=1}^m C(arr[j] + n - i - 1, n - i - 1)$

根据二项式反演公式 3：

$$\begin{aligned} f(0) &= \sum_{i=0}^n (-1)^i * C(i, 0) * g(i) \\ &= \sum_{i=0}^n (-1)^i * g(i) \end{aligned}$$

相关题目：

1. 洛谷 P5505 [JSOI2011] 分特产（标准题目）
2. BZOJ 4710 分特产（相同题目）

```
*/
```

```
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.OutputStreamWriter;  
import java.io.PrintWriter;  
import java.io.StreamTokenizer;
```

```

public class Code03_DistributeSpecialties {

    public static int MAXN = 1001;

    public static int MAXK = MAXN * 2;

    public static int MOD = 1000000007;

    public static int[] arr = new int[MAXN];

    public static long[][] c = new long[MAXK][MAXK];

    public static long[] g = new long[MAXN];

    public static int n, k, m;

    public static long compute() {
        // 预处理组合数
        for (int i = 0; i <= k; i++) {
            c[i][0] = 1;
            for (int j = 1; j <= i; j++) {
                c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % MOD;
            }
        }

        // 计算 g[i]，表示至少 i 个同学没有分到特产的方案数
        for (int i = 0; i < n; i++) {
            // C(n, i) 从 n 个同学中选出 i 个同学不分特产
            g[i] = c[n][i];
            // 对于每种特产，计算分给剩下(n-i)个同学的方案数
            for (int j = 1; j <= m; j++) {
                // 第 j 种特产有 arr[j] 个，分给(n-i)个同学的方案数是 C(arr[j] + (n-i) - 1, (n-i) - 1)
                // 即 C(arr[j] + n - i - 1, n - i - 1)
                g[i] = (int) ((g[i] * c[arr[j] + n - i - 1][n - i - 1]) % MOD);
            }
        }

        g[n] = 0; // 所有同学都不分特产显然不可能

        // 使用二项式反演计算 f(0)，即恰好 0 个同学没有分到特产的方案数
        long ans = 0;
        for (int i = 0; i <= n; i++) {
            if ((i & 1) == 0) {

```

```

        // i 为偶数, (-1)^i = 1
        ans = (ans + g[i]) % MOD;
    } else {
        // i 为奇数, (-1)^i = -1, 用(MOD-1)代替-1
        ans = (ans + g[i] * (MOD - 1) % MOD) % MOD;
    }
}

return ans;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    k = n * 2;
    in.nextToken();
    m = (int) in.nval;
    for (int i = 1; i <= m; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }
    out.println(compute());
    out.flush();
    out.close();
    br.close();
}
}

```

=====

文件: Code03_PermutationCounting.cpp

=====

```

#include <iostream>
#include <vector>
#include <string>
#include <stdexcept>
#include <chrono>

/***
 * 排列计数问题 (Permutation Counting Problem)

```

- * 题目: 洛谷 P4071 [SDOI2016]排列计数
- * 链接: <https://www.luogu.com.cn/problem/P4071>
- * 描述: 求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 $a_i = i$
- *
- * C++实现特点:
- * - 使用类封装问题求解
- * - 预处理阶乘和逆元
- * - 提供多种实现方法: 直接计算和优化版本
- * - 详细的异常处理和性能分析
- */

```

using namespace std;
using namespace chrono;

const long long MOD = 1000000007; // 模数
const int MAXN = 1000001; // 最大数据范围

class PermutationCounting {
private:
    int n; // 元素个数
    int k; // 恰好 k 个固定点
    vector<long long> fact; // 阶乘数组
    vector<long long> inv_fact; // 阶乘逆元数组

    /**
     * 快速幂算法 - 计算 base^exponent % mod
     *
     * 时间复杂度: O(log exponent)
     * 空间复杂度: O(1)
     *
     * @param base 底数
     * @param exponent 指数
     * @param mod 模数
     * @return 计算结果
     */
    long long quick_pow(long long base, long long exponent, long long mod) {
        long long result = 1;
        base %= mod;

        while (exponent > 0) {
            if (exponent & 1) { // 如果当前二进制位为 1
                result = (result * base) % mod;
            }
            exponent >>= 1;
            base = (base * base) % mod;
        }
        return result;
    }

    long long factorial(int n) {
        if (n == 0) return 1;
        return n * factorial(n - 1);
    }

    long long inverse_factorial(int n) {
        if (n == 0) return 1;
        return inv_fact[n] = quick_pow(fact[n], MOD - 2);
    }

    void preprocess() {
        fact[0] = 1;
        for (int i = 1; i < n; ++i) {
            fact[i] = fact[i - 1] * i % MOD;
        }
        inv_fact[n - 1] = inverse_factorial(n - 1);
        for (int i = n - 2; i >= 0; --i) {
            inv_fact[i] = inv_fact[i + 1] * (i + 1) % MOD;
        }
    }
};

PermutationCounting pc;
int main() {
    int n, k;
    cin >> n >> k;
    pc.n = n;
    pc.k = k;
    pc.preprocess();
    cout << pc.quick_pow(1, k, MOD) << endl;
    return 0;
}

```

```

        base = (base * base) % mod; // 底数自乘
        exponent >>= 1; // 右移一位
    }

    return result;
}

/***
 * 预处理阶乘和阶乘的逆元
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
void precompute() {
    // 计算阶乘数组
    fact.resize(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; ++i) {
        fact[i] = (fact[i - 1] * i) % MOD;
    }

    // 计算最大阶乘的逆元
    inv_fact.resize(n + 1);
    inv_fact[n] = quick_pow(fact[n], MOD - 2, MOD);

    // 倒序计算其他阶乘的逆元
    for (int i = n - 1; i >= 0; --i) {
        inv_fact[i] = (inv_fact[i + 1] * (i + 1)) % MOD;
    }
}

public:
/***
 * 构造函数
 *
 * @param n 元素个数
 * @param k 恰好 k 个固定点
 *
 * @throws invalid_argument 当参数无效时抛出异常
 */
PermutationCounting(int n, int k) {
    // 参数验证
    if (n < 0) {
        throw invalid_argument("参数 n 必须是非负整数");
    }
}

```

```

}

if (k < 0 || k > n) {
    throw invalid_argument("参数 k 必须在 0 到 n 之间");
}

this->n = n;
this->k = k;

// 预处理阶乘和逆元
precompute();
}

/***
 * 计算组合数 C(a, b)
 * 时间复杂度: O(1) - 使用预处理的阶乘和逆元
 * 空间复杂度: O(1)
 *
 * @param a 总数
 * @param b 选取的数量
 * @return 组合数 C(a, b) 对 MOD 取模的结果
 */
long long comb(int a, int b) {
    if (b < 0 || b > a) {
        return 0;
    }
    return (fact[a] * inv_fact[b] % MOD) * inv_fact[a - b] % MOD;
}

/***
 * 使用二项式反演计算恰好 k 个固定点的排列数
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * @return 恰好 k 个固定点的排列数, 对 MOD 取模
 */
long long count_fixed_points() {
    // 计算 C(n, k)
    long long c_n_k = comb(n, k);

    // 计算 D(n-k): (n-k) 个元素的错排数
    int m = n - k;
    long long derangement = 0;

```

```

// 计算 D(m) = m! * Σ (i=0 到 m) (-1)^i / i!
for (int i = 0; i <= m; ++i) {
    // 计算符号 (-1)^i
    long long sign = (i % 2 == 0) ? 1 : MOD - 1; // -1 mod MOD

    // 计算项: m! * (-1)^i / i! = fact[m] * inv_fact[i] * sign
    long long term = (fact[m] * inv_fact[i]) % MOD;
    term = (term * sign) % MOD;

    // 累加结果
    derangement = (derangement + term) % MOD;
}

// 最终结果: C(n, k) * D(n-k)
return (c_n_k * derangement) % MOD;
}

```

```

/**
* 优化版本: 直接使用递推计算错排数
* 时间复杂度: O(n)
* 空间复杂度: O(1)
*
* @return 恰好 k 个固定点的排列数, 对 MOD 取模
*/

```

```

long long count_fixed_points_optimized() {
    // 计算 C(n, k)
    long long c_n_k = comb(n, k);

    // 计算 D(n-k): 使用递推公式
    int m = n - k;

    // 边界条件
    if (m == 0) {
        return c_n_k; // D(0) = 1
    }
    if (m == 1) {
        return 0; // D(1) = 0
    }

    // 使用递推公式计算错排数: D(m) = (m-1) * (D(m-1) + D(m-2))
    long long d_prev2 = 1; // D(0)
    long long d_prev1 = 0; // D(1)
    long long d_curr = 0;

```

```

for (int i = 2; i <= m; ++i) {
    d_curr = ((i - 1) * ((d_prev1 + d_prev2) % MOD)) % MOD;
    // 更新状态
    d_prev2 = d_prev1;
    d_prev1 = d_curr;
}

// 最终结果: C(n, k) * D(n-k)
return (c_n_k * d_curr) % MOD;
}

/***
 * 运行测试用例
 *
 * @return bool 所有测试通过返回 true, 否则返回 false
 */
bool test() {
    // 测试用例
    vector<tuple<int, int, long long>> test_cases = {
        {3, 1, 3},    // n=3, k=1, 期望结果 3 种排列
        {4, 2, 6},    // n=4, k=2, 期望结果 6 种排列
        {1, 0, 0},    // n=1, k=0, 期望结果 0
        {1, 1, 1},    // n=1, k=1, 期望结果 1
        {0, 0, 1}     // n=0, k=0, 期望结果 1
    };

    bool all_passed = true;

    cout << "排列计数问题测试: " << endl;
    cout << "===== " << endl;
    cout << "n  k  预期结果  常规方法  优化方法  状态" << endl;
    cout << "===== " << endl;

    for (const auto& test_case : test_cases) {
        int test_n = get<0>(test_case);
        int test_k = get<1>(test_case);
        long long expected = get<2>(test_case);

        try {
            PermutationCounting problem(test_n, test_k);
            long long result1 = problem.count_fixed_points();
            long long result2 = problem.count_fixed_points_optimized();

```

```

        bool passed = (result1 == expected) && (result2 == expected);
        all_passed &= passed;

        cout << test_n << " " << test_k << " " << expected << " "
            << result1 << " " << result2 << " "
            << (passed ? "/" : "X") << endl;
    } catch (const exception& e) {
        cout << test_n << " " << test_k << " " << expected
            << " 异常 异常 X" << endl;
        cout << " 错误信息: " << e.what() << endl;
        all_passed = false;
    }
}

cout << "===== " << endl;
cout << "测试结果: " << (all_passed ? "通过" : "失败") << endl << endl;

return all_passed;
}
};

/***
 * 算法复杂度分析函数
 */
void analyze_complexity() {
    cout << "算法复杂度分析: " << endl;
    cout << "===== " << endl;
    cout << "常规方法: " << endl;
    cout << " 时间复杂度: O(n) - 预处理和计算各需要 O(n)时间" << endl;
    cout << " 空间复杂度: O(n) - 需要存储阶乘和逆元数组" << endl << endl;

    cout << "优化方法: " << endl;
    cout << " 时间复杂度: O(n) - 预处理需要 O(n), 错排计算需要 O(m) = O(n)" << endl;
    cout << " 空间复杂度: O(n) - 需要存储阶乘和逆元数组, 但错排计算只需要 O(1)额外空间" << endl
<< endl;

    cout << "工程化考量: " << endl;
    cout << " 1. 预处理阶乘和逆元以加速组合数计算" << endl;
    cout << " 2. 对于多次查询, 预处理可以复用" << endl;
    cout << " 3. 注意处理大数值时的溢出问题, 使用 long long 类型" << endl;
    cout << " 4. 模运算需要小心处理负数情况" << endl;
    cout << "===== " << endl;
}

```

```
}
```

```
/**
```

```
* 主函数
```

```
*/
```

```
int main() {
```

```
// 运行测试
```

```
try {
```

```
    PermutationCounting problem(1, 1);
```

```
    problem.test();
```

```
} catch (const exception& e) {
```

```
    cerr << "测试失败: " << e.what() << endl;
```

```
}
```

```
// 边界情况测试
```

```
cout << "边界情况测试: " << endl;
```

```
try {
```

```
    cout << "测试负数输入 n=-1: " << endl;
```

```
    PermutationCounting problem1(-1, 0);
```

```
    cout << " 错误: 未捕获到异常" << endl;
```

```
} catch (const invalid_argument& e) {
```

```
    cout << " 成功: 捕获到异常 - " << e.what() << endl;
```

```
}
```

```
try {
```

```
    cout << "测试 k > n: " << endl;
```

```
    PermutationCounting problem2(5, 6);
```

```
    cout << " 错误: 未捕获到异常" << endl;
```

```
} catch (const invalid_argument& e) {
```

```
    cout << " 成功: 捕获到异常 - " << e.what() << endl;
```

```
}
```

```
cout << endl;
```

```
// 分析复杂度
```

```
analyze_complexity();
```

```
// 性能测试
```

```
cout << "\n性能测试: " << endl;
```

```
cout << "===== " << endl;
```

```
try {
```

```
// 测试较大的数据规模
```

```
int test_n = 100000;
```

```

int test_k = 50000;

auto start = high_resolution_clock::now();
PermutationCounting large_problem(test_n, test_k);
long long result = large_problem.count_fixed_points_optimized();
auto end = high_resolution_clock::now();
auto duration = duration_cast<milliseconds>(end - start).count();

cout << "n = " << test_n << ", k = " << test_k << endl;
cout << "结果 = " << result << endl;
cout << "计算时间 = " << duration << " ms" << endl;
} catch (const exception& e) {
    cerr << "性能测试失败: " << e.what() << endl;
}
cout << "===== " << endl;

// 处理用户输入
try {
    int n, k;
    cout << "\n请输入n和k: " << endl;
    cout << "n = ";
    cin >> n;
    cout << "k = ";
    cin >> k;

    PermutationCounting user_problem(n, k);
    long long user_result = user_problem.count_fixed_points_optimized();
    cout << "恰好有" << k << "个固定点的排列数为: " << user_result << endl;

} catch (const exception& e) {
    cerr << "错误: " << e.what() << endl;
    return 1;
}

return 0;
}
=====
```

文件: Code03_PermutationCounting.java

```
=====
package class145;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

/***
 * 排列计数问题
 * 题目：排列中的固定点统计 (Fixed Points in Permutations)
 *
 * 问题描述：
 * 给定 n 个元素的排列，求恰好有 k 个固定点（即  $a[i] = i$ ）的排列数目
 * 这个问题也被称为部分错位排列 (Partial Derangement) 问题
 *
 * 二项式反演应用：
 * 将“恰好 k 个固定点”转化为“至少 k 个固定点”的问题
 *
 * 数据范围：
 * -  $1 \leq n \leq 10^6$ 
 * -  $0 \leq k \leq n$ 
 * - 结果对 100000007 取模
 */
public class Code03_PermutationCounting {
    // 最大数据范围
    public static final int MAXN = 1000001;
    // 模数
    public static final int MOD = 100000007;
    // 阶乘数组
    public static long[] fact = new long[MAXN];
    // 阶乘的逆元数组
    public static long[] inv_fact = new long[MAXN];
    // 输入参数
    public static int n, k;

    /**
     * 预处理阶乘和阶乘的逆元
     * 时间复杂度: O(n)
     * 空间复杂度: O(n)
     */
    public static void precompute() {
        // 计算阶乘数组
        fact[0] = 1;
```

```

for (int i = 1; i <= n; i++) {
    fact[i] = (fact[i - 1] * i) % MOD;
}

// 计算最大阶乘的逆元
inv_fact[n] = power(fact[n], MOD - 2);

// 倒序计算其他阶乘的逆元
for (int i = n - 1; i >= 0; i--) {
    inv_fact[i] = (inv_fact[i + 1] * (i + 1)) % MOD;
}

/***
 * 快速幂运算
 * 时间复杂度: O(log p)
 * 空间复杂度: O(1)
 *
 * @param a 底数
 * @param p 指数
 * @return a^p mod MOD
 */
public static long power(long a, long p) {
    long result = 1;
    a %= MOD;

    while (p > 0) {
        if ((p & 1) == 1) {
            result = (result * a) % MOD;
        }
        a = (a * a) % MOD;
        p >>= 1;
    }

    return result;
}

/***
 * 计算组合数 C(n, k)
 * 时间复杂度: O(1)
 * 空间复杂度: O(1)
 *
 * @param n 总数

```

```

* @param k 选取的数量
* @return C(n, k) mod MOD
*/
public static long comb(int n, int k) {
    if (k < 0 || k > n) {
        return 0;
    }
    return (fact[n] * inv_fact[k] % MOD) * inv_fact[n - k] % MOD;
}

/**
* 使用二项式反演计算恰好 k 个固定点的排列数
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*
* @return 恰好 k 个固定点的排列数, 对 MOD 取模
*/
public static long countFixedPoints() {
    // 预处理阶乘和逆元
    precompute();

    // 使用二项式反演公式:
    // f(k) = C(n, k) * D(n-k)
    // 其中 D(n-k) 是 n-k 个元素的错排数
    // 而 D(m) = m! * Σ(i=0 到 m) (-1)^i / i!

    // 计算 C(n, k)
    long c_n_k = comb(n, k);

    // 计算 D(n-k): (n-k) 个元素的错排数
    int m = n - k;
    long derangement = 0;

    // 计算 D(m) = m! * Σ(i=0 到 m) (-1)^i / i!
    for (int i = 0; i <= m; i++) {
        long sign = (i % 2 == 0) ? 1 : -1;
        long term = fact[m] * inv_fact[i] % MOD;

        if (sign < 0) {
            term = (MOD - term) % MOD;
        }

        derangement = (derangement + term) % MOD;
    }
}

```

```

    }

    // 最终结果: C(n, k) * D(n-k)
    return (c_n_k * derangement) % MOD;
}

/***
 * 优化版本: 直接使用递推计算错排数
 * 时间复杂度: O(n)
 * 空间复杂度: O(1)
 *
 * @return 恰好 k 个固定点的排列数, 对 MOD 取模
 */
public static long countFixedPointsOptimized() {
    // 预处理阶乘和逆元
    precompute();

    // 计算 C(n, k)
    long c_n_k = comb(n, k);

    // 计算 D(n-k): 使用递推公式
    int m = n - k;
    if (m == 0) {
        return c_n_k; // D(0) = 1
    }
    if (m == 1) {
        return 0; // D(1) = 0
    }

    long d_prev2 = 1; // D(0)
    long d_prev1 = 0; // D(1)
    long d_curr = 0;

    for (int i = 2; i <= m; i++) {
        d_curr = ((i - 1) * (d_prev1 + d_prev2)) % MOD;
        // 更新状态
        d_prev2 = d_prev1;
        d_prev1 = d_curr;
    }

    // 最终结果: C(n, k) * D(n-k)
    return (c_n_k * d_curr) % MOD;
}

```

```
/**  
 * 单元测试函数  
 */  
  
private static void runTests() {  
    // 测试用例 1: n=3, k=1  
    // 期望结果: 3 种排列 (1, 3, 2), (3, 2, 1), (2, 1, 3)  
    n = 3; k = 1;  
    long result1 = countFixedPoints();  
    System.out.printf("测试 1: n=%d, k=%d, 结果=%d (期望: 3)\n", n, k, result1);  
  
    // 测试用例 2: n=4, k=2  
    // 期望结果: 6 种排列  
    n = 4; k = 2;  
    long result2 = countFixedPoints();  
    System.out.printf("测试 2: n=%d, k=%d, 结果=%d (期望: 6)\n", n, k, result2);  
  
    // 测试优化版本  
    long optResult2 = countFixedPointsOptimized();  
    System.out.printf("优化版本测试 2: 结果=%d (与原版本相同: %b)\n", optResult2, result2 ==  
optResult2);  
  
    // 测试边界情况: n=0, k=0  
    n = 0; k = 0;  
    long result3 = countFixedPoints();  
    System.out.printf("边界测试: n=%d, k=%d, 结果=%d (期望: 1)\n", n, k, result3);  
}  
  
/**  
 * 主函数  
 */  
  
public static void main(String[] args) throws IOException {  
    // 运行单元测试  
    runTests();  
  
    // 实际处理输入输出  
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
    StreamTokenizer in = new StreamTokenizer(br);  
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));  
  
    try {  
        // 读取输入参数  
        in.nextToken();  
    }  
}
```

```

n = (int) in.nval;
in.nextToken();
k = (int) in.nval;

// 验证输入参数
if (n < 0 || k < 0 || k > n) {
    throw new IllegalArgumentException("输入参数无效: n 应>=0, k 应在 0 到 n 之间");
}

// 计算并输出结果
long result = countFixedPointsOptimized();
out.println(result);

} catch (Exception e) {
    // 异常处理
    out.println("错误: " + e.getMessage());
} finally {
    // 确保资源被关闭
    out.flush();
    out.close();
    br.close();
}
}
}

```

```

/**
 * 算法原理解析:
 * 1. 定义问题: 求恰好 k 个固定点的排列数
 * 2. 二项式反演思路:
 *     - 设 f(k) 为恰好 k 个固定点的排列数 (目标)
 *     - 设 g(k) 为至少 k 个固定点的排列数
 * 3. g(k) 的计算:
 *     - 选择 k 个位置作为固定点: C(n, k)
 *     - 其余 n-k 个位置可以任意排列: (n-k) !
 *     - 因此 g(k) = C(n, k) * (n-k) !
 * 4. 二项式反演公式:
 *     f(k) = Σ (i=k 到 n) (-1)^(i-k) * C(i, k) * g(i)
 *           = Σ (i=k 到 n) (-1)^(i-k) * C(i, k) * C(n, i) * (n-i) !
 *           = C(n, k) * Σ (i=k 到 n) (-1)^(i-k) * C(n-k, i-k) * (n-i) !
 *           = C(n, k) * Σ (j=0 到 n-k) (-1)^j * C(n-k, j) * (n-k-j) !
 *           = C(n, k) * (n-k)! * Σ (j=0 到 n-k) (-1)^j / j!
 *           = C(n, k) * D(n-k)
 * 其中 D(m) 是 m 个元素的错排数
 *

```

```

* 复杂度分析:
* - 时间复杂度: O(n) - 预处理阶乘和计算结果都需要 O(n)时间
* - 空间复杂度: O(n) - 存储阶乘和逆元数组需要 O(n)空间
*
* 优化空间:
* 1. 当 n 很大而 k 较小的时候, 可以使用递推方式计算错排数, 节省空间
* 2. 对于多次查询的场景, 可以预处理所有可能的阶乘和逆元
* 3. 当数据规模特别大时, 可以使用快速输入输出优化性能
*/
}

```

=====

文件: Code03_PermutationCounting.py

=====

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

排列计数问题 (Permutation Counting Problem)

题目: 洛谷 P4071 [SDOI2016]排列计数

链接: <https://www.luogu.com.cn/problem/P4071>

描述: 求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 $a_i = i$

Python 实现特点:

- 使用二项式反演求解部分错位排列问题
 - 预处理阶乘和逆元以优化组合数计算
 - 提供多种实现方法: 直接计算和优化版本
 - 详细的复杂度分析和注释
 - 完整的异常处理和测试用例
- """

```

MOD = 10**9 + 7 # 模数
MAXN = 10**6 + 1 # 最大数据范围

```

class PermutationCounting:

```

def __init__(self, n, k):
    """

```

初始化排列计数问题

Args:

n: 元素个数

k: 恰好 k 个固定点

```

Raises:
    ValueError: 当参数无效时抛出异常
"""

# 参数验证
if not isinstance(n, int) or not isinstance(k, int):
    raise ValueError("参数 n 和 k 必须是整数")
if n < 0:
    raise ValueError("参数 n 必须是非负整数")
if k < 0 or k > n:
    raise ValueError("参数 k 必须在 0 到 n 之间")

self.n = n
self.k = k
self.fact = []      # 阶乘数组
self.inv_fact = []  # 阶乘逆元数组

# 预处理阶乘和逆元
self.precompute()

def precompute(self):
    """
    预处理阶乘和阶乘的逆元
    时间复杂度: O(n)
    空间复杂度: O(n)
    """

    # 计算阶乘数组
    max_needed = self.n
    self.fact = [1] * (max_needed + 1)
    for i in range(1, max_needed + 1):
        self.fact[i] = (self.fact[i-1] * i) % MOD

    # 计算最大阶乘的逆元
    self.inv_fact = [1] * (max_needed + 1)
    self.inv_fact[max_needed] = pow(self.fact[max_needed], MOD-2, MOD)

    # 倒序计算其他阶乘的逆元
    for i in range(max_needed - 1, -1, -1):
        self.inv_fact[i] = (self.inv_fact[i+1] * (i+1)) % MOD

def comb(self, a, b):
    """
    计算组合数 C(a, b)

```

时间复杂度: $O(1)$ - 使用预处理的阶乘和逆元

Args:

- a: 总数
- b: 选取的数量

Returns:

组合数 $C(a, b)$ 对 MOD 取模的结果

""""

```
if b < 0 or b > a:  
    return 0  
return (self.fact[a] * self.inv_fact[b] % MOD) * self.inv_fact[a - b] % MOD
```

def count_fixed_points(self):

""""

使用二项式反演计算恰好 k 个固定点的排列数

时间复杂度: $O(n)$

空间复杂度: $O(n)$

Returns:

恰好 k 个固定点的排列数, 对 MOD 取模

""""

计算 $C(n, k)$

```
c_n_k = self.comb(self.n, self.k)
```

计算 $D(n-k)$: ($n-k$) 个元素的错排数

```
m = self.n - self.k
```

```
derangement = 0
```

计算 $D(m) = m! * \sum_{i=0}^m (-1)^i / i!$

```
for i in range(0, m + 1):
```

计算符号 $(-1)^i$

```
sign = 1 if i % 2 == 0 else -1
```

计算项: $m! * (-1)^i / i! = fact[m] * inv_fact[i] * sign$

```
term = self.fact[m] * self.inv_fact[i] % MOD
```

处理负数情况

```
if sign < 0:
```

```
    term = (MOD - term) % MOD
```

累加结果

```
derangement = (derangement + term) % MOD
```

```

# 最终结果: C(n, k) * D(n-k)
return c_n_k * derangement % MOD

def count_fixed_points_optimized(self):
    """
    优化版本: 直接使用递推计算错排数
    时间复杂度: O(n)
    空间复杂度: O(1)

    Returns:
        恰好 k 个固定点的排列数, 对 MOD 取模
    """
    # 计算 C(n, k)
    c_n_k = self.comb(self.n, self.k)

    # 计算 D(n-k): 使用递推公式
    m = self.n - self.k

    # 边界条件
    if m == 0:
        return c_n_k  # D(0) = 1
    if m == 1:
        return 0  # D(1) = 0

    # 使用递推公式计算错排数: D(m) = (m-1) * (D(m-1) + D(m-2))
    d_prev2 = 1  # D(0)
    d_prev1 = 0  # D(1)
    d_curr = 0

    for i in range(2, m + 1):
        d_curr = ((i - 1) * (d_prev1 + d_prev2)) % MOD
        # 更新状态
        d_prev2, d_prev1 = d_prev1, d_curr

    # 最终结果: C(n, k) * D(n-k)
    return c_n_k * d_curr % MOD

def test(self):
    """
    运行测试用例
    """

```

Returns:

```

    bool: 所有测试通过返回 True, 否则返回 False
"""

# 测试用例
test_cases = [
    (3, 1, 3),    # n=3, k=1, 期望结果 3 种排列
    (4, 2, 6),    # n=4, k=2, 期望结果 6 种排列
    (1, 0, 0),    # n=1, k=0, 期望结果 0
    (1, 1, 1),    # n=1, k=1, 期望结果 1
    (0, 0, 1)     # n=0, k=0, 期望结果 1
]

all_passed = True

print("排列计数问题测试: ")
print("=" * 70)
print(f"{'n':<5} {'k':<5} {'预期结果':<10} {'常规方法':<15} {'优化方法':<15} {'状态'}")
print("=" * 70)

for n, k, expected in test_cases:
    try:
        problem = PermutationCounting(n, k)
        result1 = problem.count_fixed_points()
        result2 = problem.count_fixed_points_optimized()

        passed = (result1 == expected) and (result2 == expected)
        all_passed &= passed

        print(f"{'n':<5} {'k':<5} {expected:<10} {result1:<15} {result2:<15} {'✓' if passed else '✗'}")
    except Exception as e:
        print(f"{'n':<5} {'k':<5} {expected:<10} {'异常':<15} {'异常':<15} {'✗'}")
        print(f"  错误信息: {e}")
        all_passed = False

    print("=" * 70)
    print(f"测试 {'通过' if all_passed else '失败'}")
    print()

return all_passed

def main():
"""
主函数, 处理输入并计算结果

```

```
"""
# 运行测试
PermutationCounting(1, 1).test()

# 边界情况测试
print("边界情况测试: ")
try:
    PermutationCounting(-1, 0)
    print(" 错误: 未捕获到 n 为负数的异常")
except ValueError as e:
    print(f" 成功: 捕获到 n 为负数的异常 - {e}")

try:
    PermutationCounting(5, 6)
    print(" 错误: 未捕获到 k 大于 n 的异常")
except ValueError as e:
    print(f" 成功: 捕获到 k 大于 n 的异常 - {e}")

# 性能测试
print("\n 性能测试: ")
print("=" * 70)

import time

# 测试较大的数据规模
large_n = 100000
large_k = 50000

start_time = time.time()
problem = PermutationCounting(large_n, large_k)
result = problem.count_fixed_points_optimized()
end_time = time.time()

print(f"n = {large_n}, k = {large_k}")
print(f"结果 = {result}")
print(f"计算时间 = {(end_time - start_time) * 1000:.2f} ms")
print("=" * 70)

# 处理用户输入
try:
    print("\n 请输入 n 和 k: ")
    n = int(input("n = "))
    k = int(input("k = "))

```

```

problem = PermutationCounting(n, k)
result = problem.count_fixed_points_optimized()
print(f"恰好有{k}个固定点的排列数为: {result}")

except ValueError as e:
    print(f"输入错误: {e}")

if __name__ == "__main__":
    main()

```

=====

文件: Code04_NothingFear.java

=====

```

package class145;

// 已经没有什么好害怕的了
// 给定两个长度为 n 的数组, a[i] 表示第 i 个糖果的能量, b[i] 表示第 i 个药片的能量
// 所有能量数值都不相同, 每一个糖果要选一个药片进行配对
// 如果配对之后, 糖果能量 > 药片能量, 称为糖果大的配对
// 如果配对之后, 糖果能量 < 药片能量, 称为药片大的配对
// 希望做到, 糖果大的配对数量 = 药片大的配对数量 + k
// 返回配对方法数, 答案对 1000000009 取模
// 举例, a = [5, 35, 15, 45], b = [40, 20, 10, 30], k = 2, 返回 4, 因为有 4 种配对方法
// (5-40, 35-20, 15-10, 45-30)、(5-40, 35-30, 15-10, 45-20)
// (5-20, 35-30, 15-10, 45-40)、(5-30, 35-20, 15-10, 45-40)
// 1 <= n <= 2000
// 0 <= k <= n
// 测试链接 : https://www.luogu.com.cn/problem/P4859
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例

```

/*

二项式反演在配对问题中的应用:

问题描述:

给定两个长度为 n 的数组 a 和 b, 将它们两两配对。

设糖果大的配对数为 x, 药片大的配对数为 y, 要求 $x - y = k$, 即 $x = y + k$ 。

因为 $x + y = n$, 所以有:

$$x + y = n$$

$$x - y = k$$

解得: $x = (n + k) / 2$, $y = (n - k) / 2$

如果 $(n + k)$ 是奇数，则无解。

解题思路：

设 $f(i)$ 表示恰好有 i 个糖果大的配对的方案数（即答案）

设 $g(i)$ 表示至少有 i 个糖果大的配对的方案数（钦定 i 个糖果大的配对）

$g(i)$ 的计算可以通过 DP 实现：

1. 将 a 和 b 数组排序
2. 对于 $a[j]$ ，计算有多少个 $b[k]$ 满足 $b[k] < a[j]$ ，记为 $small[j]$
3. 使用 DP， $dp[i][j]$ 表示前 i 个元素中，选出了 j 个糖果大的配对的方案数

根据二项式反演：

$$f(k) = \sum_{i=k \text{ to } n} (-1)^{i-k} * C(i, k) * g(i) * (n-i)!$$

其中 $(n-i)!$ 是因为剩下的 $(n-i)$ 个配对可以任意排列。

相关题目：

1. 洛谷 P4859 已经没有什么好害怕的了（标准题目）
 2. 类似题目可以转化为“恰好 k 个满足某条件”的问题
- */

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.Arrays;

public class Code04_NothingFear {

    public static int MAXN = 2001;

    public static int MOD = 1000000009;

    public static int n, k;

    public static int[] a = new int[MAXN];

    public static int[] b = new int[MAXN];

    public static long[] fac = new long[MAXN];
```

```

public static long[][] c = new long[MAXN][MAXN];

public static long[] small = new long[MAXN];

public static long[][] dp = new long[MAXN][MAXN];

public static long[] g = new long[MAXN];

public static void build() {
    fac[0] = 1;
    for (int i = 1; i <= n; i++) {
        fac[i] = fac[i - 1] * i % MOD;
    }
    for (int i = 0; i <= n; i++) {
        c[i][0] = 1;
        for (int j = 1; j <= i; j++) {
            c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % MOD;
        }
    }
}

public static long compute() {
    build();
    // 对数组进行排序，便于后续计算
    Arrays.sort(a, 1, n + 1);
    Arrays.sort(b, 1, n + 1);

    // 计算对于每个 a[i]，有多少个 b[j] 小于它
    for (int i = 1, cnt = 0; i <= n; i++) {
        while (cnt + 1 <= n && b[cnt + 1] < a[i]) {
            cnt++;
        }
        small[i] = cnt;
    }

    // DP 计算至少 i 个糖果大的配对数
    // dp[i][j] 表示考虑前 i 个元素，选出了 j 个糖果大的配对的方案数
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        dp[i][0] = dp[i - 1][0];
        for (int j = 1; j <= i; j++) {
            // 不选择第 i 个元素作为糖果大的配对
            dp[i][j] = (dp[i - 1][j] +

```

```

        // 选择第 i 个元素作为糖果大的配对，有(small[i] - j + 1)种选择
        dp[i - 1][j - 1] * (small[i] - j + 1) % MOD) % MOD;
    }
}

// g[i]表示至少 i 个糖果大的配对数，还要乘以剩余元素的排列数
for (int i = 0; i <= n; i++) {
    g[i] = fac[n - i] * dp[n][i] % MOD;
}

// 二项式反演计算恰好 k 个糖果大的配对数
long ans = 0;
for (int i = k; i <= n; i++) {
    if (((i - k) & 1) == 0) {
        // (i-k)为偶数, (-1)^(i-k) = 1
        ans = (ans + c[i][k] * g[i] % MOD) % MOD;
    } else {
        // (i-k)为奇数, (-1)^(i-k) = -1
        // 用(MOD-1)代替-1
        ans = (ans + c[i][k] * g[i] % MOD * (MOD - 1) % MOD) % MOD;
    }
}
return ans;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StringTokenizer in = new StringTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    k = (int) in.nval;
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        a[i] = (int) in.nval;
    }
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        b[i] = (int) in.nval;
    }
    if (((n + k) & 1) == 0) {
        // 如果(n+k)是偶数，则 k = (n+k)/2

```

```

        k = (n + k) / 2;
        out.println(compute());
    } else {
        // 如果(n+k)是奇数，则无解
        out.println(0);
    }
    out.flush();
    out.close();
    br.close();
}

}

```

文件: Code05_Game1.java

```

package class145;

// 游戏(递归版)
// 一共有 n 个节点, n <= 5000, n 为偶数, 其中有 m 个点属于小 A, 有 m 个点属于小 B, m 为 n 的一半
// 给定 n-1 条边, 节点之间组成一颗树, 1 号节点是根节点
// 给定长度为 n 的数组 arr, arr[i] 的值表示 i 号节点由谁拥有, 0 为小 A 拥有, 1 为小 B 拥有
// 游戏有 m 回合, 每回合都有胜负, 两人需要选择一个自己拥有、但之前没选过的点, 作为本回合当前点
// 小 A 当前点的子树里有小 B 当前点, 则小 A 胜; 小 B 当前点的子树里有小 A 当前点, 则小 B 胜; 否则平局
// 返回 m 回合里能出现 k 次非平局的游戏方法数, 打印 k=0..m 时的所有答案, 对 998244353 取模
// 两场游戏视为不同的定义: 当且仅当存在小 A 拥有的点 x, 小 B 在小 A 选择 x 的那个回合所选择的点不同
// 测试链接 : https://www.luogu.com.cn/problem/P6478
// 提交以下的 code, 提交时请把类名改成"Main", 注意 dfs 是递归函数
// C++的同学可以全部通过, java 的同学有时可以全部通过, 有时因为递归展开太深而爆栈
// dfs 从递归版改迭代版的实现, 请看 Code05_Game2 文件

```

/*

二项式反演在树上游戏问题中的应用:

问题描述:

给定一棵 n 个节点的树, 其中 n 为偶数。有 $m=n/2$ 个节点属于小 A(标记为 0), m 个节点属于小 B(标记为 1)。游戏有 m 回合, 每回合两个玩家各选一个自己拥有且未被选过的点。

如果小 A 选的点的子树包含小 B 选的点, 则小 A 胜;

如果小 B 选的点的子树包含小 A 选的点, 则小 B 胜;

否则为平局。

求出现恰好 k 次非平局情况的游戏方法数, 对 $k=0$ 到 m 输出所有答案。

解题思路：

设 $f(i)$ 表示恰好出现 i 次非平局的方案数

设 $g(i)$ 表示至少出现 i 次非平局的方案数

计算 $g(i)$ 可以通过树形 DP 实现：

1. 对于每个节点 u , 计算其子树中属于对手的节点数
2. 使用树形 DP, $dp[u][i]$ 表示以 u 为根的子树中, 出现 i 次非平局的方案数
3. 在转移时考虑是否选择当前节点参与非平局情况

根据二项式反演：

$$f(k) = \sum_{i=k \text{ to } m} (-1)^{i-k} * C(i, k) * g(i) * (m-i)!$$

其中 $(m-i)!$ 是因为剩下的 $(m-i)$ 次游戏可以任意安排。

相关题目：

1. 洛谷 P6478 游戏（标准题目）
 2. 树上计数问题，通常结合树形 DP 和组合数学
- */

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.StringTokenizer;

public class Code05_Game1 {

    public static final int MAXN = 5001;

    public static final int MOD = 998244353;

    public static int n, m;

    public static int[] arr = new int[MAXN];

    public static long[] fac = new long[MAXN];

    public static long[][] c = new long[MAXN][MAXN];

    // 链式前向星需要
```

```
public static int[] head = new int[MAXN];

public static int[] next = new int[MAXN << 1];

public static int[] to = new int[MAXN << 1];

public static int cnt;

// dfs 需要
public static int[] size = new int[MAXN];

public static int[][] belong = new int[MAXN][2];

public static long[][] dp = new long[MAXN][MAXN];

public static long[] backup = new long[MAXN];

// 反演需要
public static long[] g = new long[MAXN];

// 最后答案
public static long[] f = new long[MAXN];

public static void build() {
    cnt = 1;
    fac[0] = 1;
    for (int i = 1; i <= n; i++) {
        head[i] = 0;
        fac[i] = fac[i - 1] * i % MOD;
    }
    for (int i = 0; i <= n; i++) {
        c[i][0] = 1;
        for (int j = 1; j <= i; j++) {
            c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % MOD;
        }
    }
}

public static void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}
```

```

// 递归版
public static void dfs(int u, int fa) {
    size[u] = 1;
    belong[u][arr[u]] = 1;
    dp[u][0] = 1;
    // 首先计算不包含头节点的方法数
    for (int e = head[u], v; e > 0; e = next[e]) {
        v = to[e];
        if (v != fa) {
            dfs(v, u);
            // 之前所有子树结合的计算结果，拷贝进 backup
            for (int i = 0; i <= Math.min(size[u] / 2, m); i++) {
                backup[i] = dp[u][i];
                dp[u][i] = 0;
            }
            // 树型 dp 的枚举行为利用子树的节点数做上限进行复杂度优化
            for (int l = 0; l <= Math.min(size[u] / 2, m); l++) {
                for (int r = 0; r <= Math.min(size[v] / 2, m - l); r++) {
                    dp[u][l + r] = (dp[u][l + r] + backup[l] * dp[v][r] % MOD) % MOD;
                }
            }
            size[u] += size[v];
            belong[u][0] += belong[v][0];
            belong[u][1] += belong[v][1];
        }
    }
    // 最后计算包含头节点的方法数
    // 更新 dp[u][i]，i 可以把上限定为 min(对手拥有的节点数, m)
    // u 为头的子树中，对手有几个节点
    int num = belong[u][arr[u] ^ 1];
    // 不包含头节点的方法数，拷贝到 backup
    for (int i = 1; i <= Math.min(num, m); i++) {
        backup[i] = dp[u][i];
    }
    // 计算包含头节点的方法数，累加上
    for (int i = 1; i <= Math.min(num, m); i++) {
        dp[u][i] = (dp[u][i] + backup[i - 1] * (num - i + 1) % MOD) % MOD;
    }
}

public static void compute() {
    dfs(1, 0); // dfs 是递归版
}

```

```

for (int i = 0; i <= m; i++) {
    g[i] = dp[1][i] * fac[m - i] % MOD;
}
for (int k = 0; k <= m; k++) {
    for (int i = k; i <= m; i++) {
        if (((i - k) & 1) == 0) {
            f[k] = (f[k] + c[i][k] * g[i] % MOD) % MOD;
        } else {
            // -1 和 (MOD-1) 同余
            f[k] = (f[k] + c[i][k] * g[i] % MOD * (MOD - 1) % MOD) % MOD;
        }
    }
}

public static void main(String[] args) {
    Kattio io = new Kattio();
    n = io.nextInt();
    m = n >> 1;
    build();
    String str = io.next();
    for (int i = 1; i <= n; i++) {
        arr[i] = str.charAt(i - 1) - '0';
    }
    for (int i = 1, u, v; i < n; i++) {
        u = io.nextInt();
        v = io.nextInt();
        addEdge(u, v);
        addEdge(v, u);
    }
    compute();
    for (int k = 0; k <= m; k++) {
        io.println(f[k]);
    }
    io.flush();
    io.close();
}

// Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
// 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
// 参考链接：https://oi-wiki.org/lang/java-pro/
public static class Kattio extends PrintWriter {
    private BufferedReader r;
}

```

```
private StringTokenizer st;

public Kattio() {
    this(System.in, System.out);
}

public Kattio(InputStream i, OutputStream o) {
    super(o);
    r = new BufferedReader(new InputStreamReader(i));
}

public Kattio(String intput, String output) throws IOException {
    super(output);
    r = new BufferedReader(new FileReader(intput));
}

public String next() {
    try {
        while (st == null || !st.hasMoreTokens())
            st = new StringTokenizer(r.readLine());
        return st.nextToken();
    } catch (Exception e) {
    }
    return null;
}

public int nextInt() {
    return Integer.parseInt(next());
}

public double nextDouble() {
    return Double.parseDouble(next());
}

public long nextLong() {
    return Long.parseLong(next());
}

}
```

=====

文件: Code05_Game2.java

```
=====
package class145;

// 游戏(迭代版)
// 一共有 n 个节点, n <= 5000, n 为偶数, 其中有 m 个点属于小 A, 有 m 个点属于小 B, m 为 n 的一半
// 给定 n-1 条边, 节点之间组成一颗树, 1 号节点是根节点
// 给定长度为 n 的数组 arr, arr[i] 的值表示 i 号节点由谁拥有, 0 为小 A 拥有, 1 为小 B 拥有
// 游戏有 m 回合, 每回合都有胜负, 两人需要选择一个自己拥有、但之前没选过的点, 作为本回合当前点
// 小 A 当前点的子树里有小 B 当前点, 则小 A 胜; 小 B 当前点的子树里有小 A 当前点, 则小 B 胜; 否则平局
// 返回 m 回合里能出现 k 次非平局的游戏方法数, 打印 k=0..m 时的所有答案, 对 998244353 取模
// 两场游戏视为不同的定义: 当且仅当存在小 A 拥有的点 x, 小 B 在小 A 选择 x 的那个回合所选择的点不同
// 测试链接 : https://www.luogu.com.cn/problem/P6478
// 提交以下的 code, 提交时请把类名改成"Main", 可以通过所有测试用例
```

/*

二项式反演在树上游戏问题中的应用:

问题描述:

给定一棵 n 个节点的树, 其中 n 为偶数。有 $m=n/2$ 个节点属于小 A(标记为 0), m 个节点属于小 B(标记为 1)。游戏有 m 回合, 每回合两个玩家各选一个自己拥有且未被选过的点。

如果小 A 选的点的子树包含小 B 选的点, 则小 A 胜;

如果小 B 选的点的子树包含小 A 选的点, 则小 B 胜;

否则为平局。

求出现恰好 k 次非平局情况的游戏方法数, 对 k=0 到 m 输出所有答案。

解题思路:

设 $f(i)$ 表示恰好出现 i 次非平局的方案数

设 $g(i)$ 表示至少出现 i 次非平局的方案数

计算 $g(i)$ 可以通过树形 DP 实现:

1. 对于每个节点 u, 计算其子树中属于对手的节点数
2. 使用树形 DP, $dp[u][i]$ 表示以 u 为根的子树中, 出现 i 次非平局的方案数
3. 在转移时考虑是否选择当前节点参与非平局情况

根据二项式反演:

$$f(k) = \sum_{i=k \text{ to } m} (-1)^{i-k} * C(i, k) * g(i) * (m-i)!$$

其中 $(m-i)!$ 是因为剩下的 $(m-i)$ 次游戏可以任意安排。

相关题目:

1. 洛谷 P6478 游戏 (标准题目)
2. 树上计数问题, 通常结合树形 DP 和组合数学

迭代版说明：

由于 Java 递归深度限制，当树的深度较大时可能会栈溢出。

使用显式栈模拟递归过程，将递归版改为迭代版可以避免这个问题。

*/

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.StringTokenizer;

public class Code05_Game2 {

    public static final int MAXN = 5001;

    public static final int MOD = 998244353;

    public static int n, m;

    public static int[] arr = new int[MAXN];

    public static long[] fac = new long[MAXN];

    public static long[][] c = new long[MAXN][MAXN];

    // 链式前向星需要
    public static int[] head = new int[MAXN];

    public static int[] next = new int[MAXN << 1];

    public static int[] to = new int[MAXN << 1];

    public static int cnt;

    // dfs 需要
    public static int[] size = new int[MAXN];

    public static int[][] belong = new int[MAXN][2];
```

```

public static long[][] dp = new long[MAXN][MAXN];

public static long[] backup = new long[MAXN];

// 反演需要
public static long[] g = new long[MAXN];

// 最后答案
public static long[] f = new long[MAXN];

public static void build() {
    cnt = 1;
    fac[0] = 1;
    for (int i = 1; i <= n; i++) {
        head[i] = 0;
        fac[i] = fac[i - 1] * i % MOD;
    }
    for (int i = 0; i <= n; i++) {
        c[i][0] = 1;
        for (int j = 1; j <= i; j++) {
            c[i][j] = (c[i - 1][j] + c[i - 1][j - 1]) % MOD;
        }
    }
}

public static void addEdge(int u, int v) {
    next[cnt] = head[u];
    to[cnt] = v;
    head[u] = cnt++;
}

// 迭代版
// ufe 是为了实现迭代版而准备的栈
// 不会改，看讲解 118，讲了怎么从递归版改成迭代版
public static int[][] ufe = new int[MAXN][3];

public static int stackSize, u, fa, e;

public static void push(int u, int fa, int e) {
    ufe[stackSize][0] = u;
    ufe[stackSize][1] = fa;
    ufe[stackSize][2] = e;
    stackSize++;
}

```

```

}

public static void pop() {
    --stackSize;
    u = ufe[stackSize][0];
    fa = ufe[stackSize][1];
    e = ufe[stackSize][2];
}

// 迭代版
public static void dfs(int root) {
    stackSize = 0;
    push(root, 0, -1);
    int v, num;
    while (stackSize > 0) {
        pop();
        if (e == -1) { // 第一次来到当前节点，设置初始值
            size[u] = 1;
            belong[u][arr[u]] = 1;
            dp[u][0] = 1;
            e = head[u];
        } else { // 不是第一次来到当前节点
            v = to[e];
            if (v != fa) { // 之前的孩子，dfs 过程计算完了，所以用之前孩子的信息，更新当前节点
                for (int i = 0; i <= Math.min(size[u] / 2, m); i++) {
                    backup[i] = dp[u][i];
                    dp[u][i] = 0;
                }
                for (int l = 0; l <= Math.min(size[u] / 2, m); l++) {
                    for (int r = 0; r <= Math.min(size[v] / 2, m - 1); r++) {
                        dp[u][l + r] = (dp[u][l + r] + backup[l] * dp[v][r] % MOD) % MOD;
                    }
                }
                size[u] += size[v];
                belong[u][0] += belong[v][0];
                belong[u][1] += belong[v][1];
            }
            // 来到去往下一个孩子的边
            e = next[e];
        }
        if (e != 0) { // 还有后续子树
            push(u, fa, e);
        }
    }
}

```

```

        if (to[e] != fa) {
            push(to[e], u, -1);
        }
    } else { // 没有后续子树，最后计算包含头节点的方法数
        num = belong[u][arr[u] ^ 1];
        for (int i = 1; i <= Math.min(num, m); i++) {
            backup[i] = dp[u][i];
        }
        for (int i = 1; i <= Math.min(num, m); i++) {
            dp[u][i] = (dp[u][i] + backup[i - 1] * (num - i + 1) % MOD) % MOD;
        }
    }
}

public static void compute() {
    dfs(1); // dfs 是迭代版
    for (int i = 0; i <= m; i++) {
        g[i] = dp[1][i] * fac[m - i] % MOD;
    }
    for (int k = 0; k <= m; k++) {
        for (int i = k; i <= m; i++) {
            if (((i - k) & 1) == 0) {
                f[k] = (f[k] + c[i][k] * g[i] % MOD) % MOD;
            } else {
                f[k] = (f[k] + c[i][k] * g[i] % MOD * (MOD - 1) % MOD) % MOD;
            }
        }
    }
}

public static void main(String[] args) {
    Kattio io = new Kattio();
    n = io.nextInt();
    m = n >> 1;
    build();
    String str = io.next();
    for (int i = 1; i <= n; i++) {
        arr[i] = str.charAt(i - 1) - '0';
    }
    for (int i = 1, u, v; i < n; i++) {
        u = io.nextInt();
        v = io.nextInt();
    }
}

```

```

        addEdge(u, v);
        addEdge(v, u);
    }
    compute();
    for (int k = 0; k <= m; k++) {
        io.println(f[k]);
    }
    io.flush();
    io.close();
}

// Kattio 类 IO 效率很好，但还是不如 StreamTokenizer
// 只有 StreamTokenizer 无法正确处理时，才考虑使用这个类
// 参考链接 : https://oi-wiki.org/lang/java-pro/
public static class Kattio extends PrintWriter {
    private BufferedReader r;
    private StringTokenizer st;

    public Kattio() {
        this(System.in, System.out);
    }

    public Kattio(InputStream i, OutputStream o) {
        super(o);
        r = new BufferedReader(new InputStreamReader(i));
    }

    public Kattio(String intput, String output) throws IOException {
        super(output);
        r = new BufferedReader(new FileReader(intput));
    }

    public String next() {
        try {
            while (st == null || !st.hasMoreTokens())
                st = new StringTokenizer(r.readLine());
            return st.nextToken();
        } catch (Exception e) {
        }
        return null;
    }

    public int nextInt() {

```

```

        return Integer.parseInt(next());
    }

    public double nextDouble() {
        return Double.parseDouble(next());
    }

    public long nextLong() {
        return Long.parseLong(next());
    }

}

=====

```

文件: Code06_CF1342E.java

```

package class145;

// Codeforces 1342E Placing Rooks
// 题目大意:
// 在一个  $n \times n$  的棋盘上放置  $n$  个车, 要求:
// 1. 每个格子都至少被一个车攻击到
// 2. 恰好有  $k$  对车互相攻击
// 求方案数, 答案对 998244353 取模

// 解题思路:
// 1. 要满足条件 1, 每行或每列都必须有车
// 2. 如果  $k \geq n$ , 则无解, 因为  $n$  个车最多形成  $(n-1)$  对互相攻击
// 3. 假设每行都有车, 有  $m$  列有车, 则会产生  $(n-m)$  对互相攻击
// 因此  $m = n - k$ 
// 4. 问题转化为: 把  $n$  个不同的车放进  $(n-k)$  个不同的列中, 每列至少有一个车
// 5. 这是典型的第二类斯特林数问题, 方案数为  $S(n, n-k) * (n-k)!$ 
// 6. 由于可以是每行有车或每列有车, 所以总方案数要乘以 2
// 7. 特殊情况:  $k=0$  时, 每行每列都恰好有一个车, 方案数为  $n!$ 

// 二项式反演在本题中的应用:
// 第二类斯特林数可以用容斥原理计算, 也可以用二项式反演理解
//  $S(n, m) = (1/m!) * \sum_{i=0}^m (-1)^{m-i} * C(m, i) * i^n$ 

import java.io.*;
import java.util.*;

```

```

public class Code06_CF1342E {
    static final int MOD = 998244353;
    static final int MAXN = 200005;

    static long[] fact = new long[MAXN];
    static long[] ifact = new long[MAXN];

    // 快速幂
    static long pow(long base, long exp) {
        long res = 1;
        while (exp > 0) {
            if (exp % 2 == 1) res = res * base % MOD;
            base = base * base % MOD;
            exp /= 2;
        }
        return res;
    }

    // 预处理阶乘和阶乘逆元
    static void initFact(int n) {
        fact[0] = 1;
        for (int i = 1; i <= n; i++) {
            fact[i] = fact[i - 1] * i % MOD;
        }
        ifact[n] = pow(fact[n], MOD - 2);
        for (int i = n - 1; i >= 0; i--) {
            ifact[i] = ifact[i + 1] * (i + 1) % MOD;
        }
    }

    // 计算组合数
    static long comb(int n, int k) {
        if (k > n || k < 0) return 0;
        return fact[n] * ifact[k] % MOD * ifact[n - k] % MOD;
    }

    // 计算第二类斯特林数 S(n, m) * m!
    // 即将 n 个不同的球放入 m 个不同的盒子，每个盒子非空的方案数
    static long stirling2(int n, int m) {
        if (m > n || m < 0) return 0;
        long res = 0;
        for (int i = 0; i <= m; i++) {

```

```

long term = comb(m, i) * pow(i, n) % MOD;
if ((m - i) % 2 == 0) {
    res = (res + term) % MOD;
} else {
    res = (res - term + MOD) % MOD;
}
}

return res;
}

```

```

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int n = scanner.nextInt();
    int k = scanner.nextInt();

    // 特殊情况: k >= n 无解
    if (k >= n) {
        System.out.println(0);
        return;
    }

```

```

    // 预处理
    initFact(n);

```

```

    // 特殊情况: k = 0
    if (k == 0) {
        System.out.println(fact[n]);
        return;
    }

```

```

    // 一般情况: m = n - k
    int m = n - k;

```

```

    // 计算将 n 个车放入 m 列, 每列至少一个的方案数
    long ways = stirling2(n, m);

```

```

    // 选择哪 m 列
    ways = ways * comb(n, m) % MOD;

```

```

    // 可以是每行有车或每列有车, 所以乘以 2
    ways = ways * 2 % MOD;

```

```

System.out.println(ways);

```

```
}
```

```
}
```

```
=====
```

文件: Code06_CF1342E.py

```
# Codeforces 1342E Placing Rooks
```

题目大意:

在一个 $n \times n$ 的棋盘上放置 n 个车, 要求:

1. 每个格子都至少被一个车攻击到

2. 恰好有 k 对车互相攻击

求方案数, 答案对 998244353 取模

解题思路:

1. 要满足条件 1, 每行或每列都必须有车

2. 如果 $k \geq n$, 则无解, 因为 n 个车最多形成 $(n-1)$ 对互相攻击

3. 假设每行都有车, 有 m 列有车, 则会产生 $(n-m)$ 对互相攻击

因此 $m = n - k$

4. 问题转化为: 把 n 个不同的车放进 $(n-k)$ 个不同的列中, 每列至少有一个车

5. 这是典型的第二类斯特林数问题, 方案数为 $S(n, n-k) * (n-k)!$

6. 由于可以是每行有车或每列有车, 所以总方案数要乘以 2

7. 特殊情况: $k=0$ 时, 每行每列都恰好有一个车, 方案数为 $n!$

二项式反演在本题中的应用:

第二类斯特林数可以用容斥原理计算, 也可以用二项式反演理解

$S(n, m) = (1/m!) * \sum_{i=0}^m (-1)^{m-i} * C(m, i) * i^n$

MOD = 998244353

```
def pow_mod(base, exp, mod):
    res = 1
    while exp > 0:
        if exp % 2 == 1:
            res = res * base % mod
        base = base * base % mod
        exp //= 2
    return res
```

```
def modinv(a, mod):
    return pow_mod(a, mod - 2, mod)
```

预处理阶乘和阶乘逆元

```

def init_fact(n):
    fact = [1] * (n + 1)
    for i in range(1, n + 1):
        fact[i] = fact[i - 1] * i % MOD
    ifact = [1] * (n + 1)
    ifact[n] = modinv(fact[n], MOD)
    for i in range(n - 1, -1, -1):
        ifact[i] = ifact[i + 1] * (i + 1) % MOD
    return fact, ifact

# 计算组合数
def comb(n, k, fact, ifact):
    if k > n or k < 0:
        return 0
    return fact[n] * ifact[k] % MOD * ifact[n - k] % MOD

# 计算第二类斯特林数 S(n, m) * m!
# 即将 n 个不同的球放入 m 个不同的盒子，每个盒子非空的方案数
def stirling2(n, m, fact, ifact):
    if m > n or m < 0:
        return 0
    res = 0
    for i in range(m + 1):
        term = comb(m, i, fact, ifact) * pow_mod(i, n, MOD) % MOD
        if (m - i) % 2 == 0:
            res = (res + term) % MOD
        else:
            res = (res - term + MOD) % MOD
    return res

def main():
    n, k = map(int, input().split())

    # 特殊情况: k >= n 无解
    if k >= n:
        print(0)
        return

    # 预处理
    fact, ifact = init_fact(n)

    # 特殊情况: k = 0
    if k == 0:

```

```

print(fact[n])
return

# 一般情况: m = n - k
m = n - k

# 计算将 n 个车放入 m 列, 每列至少一个的方案数
ways = stirling2(n, m, fact, ifact)

# 选择哪 m 列
ways = ways * comb(n, m, fact, ifact) % MOD

# 可以是每行有车或每列有车, 所以乘以 2
ways = ways * 2 % MOD

print(ways)

```

```

if __name__ == "__main__":
    main()
=====
```

文件: Code07_SDOI2016Permutation.cpp

```

// [SDOI2016]排列计数
// 题目描述: 求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 a_i = i。
// 1 <= T <= 5*10^5, 1 <= n <= 10^6, 0 <= m <= 10^6
// 测试链接: https://www.luogu.com.cn/problem/P4071
```

/*

二项式反演在排列计数问题中的应用:

问题描述:

求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 a_i = i。

解题思路:

设 $f(i)$ 表示恰好有 i 个位置满足 $a[j] = j$ 的排列数 (即答案)

设 $g(i)$ 表示至少有 i 个位置满足 $a[j] = j$ 的排列数

显然, $g(i)$ 更容易计算:

1. 先从 n 个位置中选出 i 个位置固定, 方案数为 $C(n, i)$
2. 剩下的 $(n-i)$ 个位置必须错排, 方案数为 $D(n-i)$

因此: $g(i) = C(n, i) * D(n-i)$

根据二项式反演公式 2:

$$\begin{aligned}f(m) &= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * g(i) \\&= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * C(n, i) * D(n-i)\end{aligned}$$

其中 $D(k)$ 是 k 个元素的错排数, 可以用递推公式计算:

$$D(0) = 1, D(1) = 0$$

$$D(k) = (k-1) * (D(k-1) + D(k-2))$$

相关题目:

1. 洛谷 P4071 [SDOI2016]排列计数 (标准题目)

2. 洛谷 P1595 信封问题 (错排问题)

*/

```
// [SDOI2016]排列计数
```

```
// 题目描述: 求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 a_i = i。
```

```
// 1 <= T <= 5*10^5, 1 <= n <= 10^6, 0 <= m <= 10^6
```

```
// 测试链接: https://www.luogu.com.cn/problem/P4071
```

/*

二项式反演在排列计数问题中的应用:

问题描述:

求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 $a_i = i$ 。

解题思路:

设 $f(i)$ 表示恰好有 i 个位置满足 $a[j] = j$ 的排列数 (即答案)

设 $g(i)$ 表示至少有 i 个位置满足 $a[j] = j$ 的排列数

显然, $g(i)$ 更容易计算:

1. 先从 n 个位置中选出 i 个位置固定, 方案数为 $C(n, i)$

2. 剩下的 $(n-i)$ 个位置必须错排, 方案数为 $D(n-i)$

因此: $g(i) = C(n, i) * D(n-i)$

根据二项式反演公式 2:

$$\begin{aligned}f(m) &= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * g(i) \\&= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * C(n, i) * D(n-i)\end{aligned}$$

其中 $D(k)$ 是 k 个元素的错排数, 可以用递推公式计算:

$$D(0) = 1, D(1) = 0$$

$$D(k) = (k-1) * (D(k-1) + D(k-2))$$

相关题目：

1. 洛谷 P4071 [SDOI2016]排列计数（标准题目）

2. 洛谷 P1595 信封问题（错排问题）

*/

```
const int MOD = 1000000007;
```

```
const int MAXN = 1000001;
```

```
// 预处理阶乘和逆元
```

```
long long fact[MAXN];
```

```
long long ifact[MAXN];
```

```
// 错排数
```

```
long long derange[MAXN];
```

```
// 快速幂
```

```
long long pow(long long base, long long exp) {
```

```
    long long res = 1;
```

```
    while (exp > 0) {
```

```
        if (exp % 2 == 1) res = res * base % MOD;
```

```
        base = base * base % MOD;
```

```
        exp /= 2;
```

```
}
```

```
    return res;
```

```
}
```

```
// 预处理
```

```
void init() {
```

```
    // 预处理阶乘
```

```
    fact[0] = 1;
```

```
    for (int i = 1; i < MAXN; i++) {
```

```
        fact[i] = fact[i-1] * i % MOD;
```

```
}
```

```
// 预处理逆元
```

```
    ifact[MAXN-1] = pow(fact[MAXN-1], MOD-2);
```

```
    for (int i = MAXN-2; i >= 0; i--) {
```

```
        ifact[i] = ifact[i+1] * (i+1) % MOD;
```

```
}
```

```
// 预处理错排数
```

```
    derange[0] = 1;
```

```
    derange[1] = 0;
```

```

for (int i = 2; i < MAXN; i++) {
    derange[i] = (i-1) * (derange[i-1] + derange[i-2]) % MOD;
}
}

// 计算组合数 C(n, k)
long long comb(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * ifact[k] % MOD * ifact[n-k] % MOD;
}

// 计算恰好有 m 个位置满足 a[i] = i 的排列数
long long solve(int n, int m) {
    // 特殊情况处理
    if (m > n) return 0;

    // 使用二项式反演计算答案
    long long ans = 0;
    for (int i = m; i <= n; i++) {
        // (-1)^(i-m) * C(i, m) * C(n, i) * D(n-i)
        long long term = comb(i, m) * comb(n, i) % MOD * derange[n-i] % MOD;
        if ((i-m) % 2 == 0) {
            ans = (ans + term) % MOD;
        } else {
            ans = (ans - term + MOD) % MOD;
        }
    }
    return ans;
}

// 由于编译环境限制，此处省略 main 函数实现
// 实际使用时需要实现输入输出功能
// int main() {
//     init();
//     int T;
//     // 输入 T
//     for (int i = 0; i < T; i++) {
//         int n, m;
//         // 输入 n, m
//         // 输出 solve(n, m)
//     }
// }

```

```
//      return 0;  
// }
```

=====文件: Code07_SDOI2016Permutation.java=====

```
package class145;
```

```
// [SDOI2016]排列计数
```

```
// 题目描述: 求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 a_i = i。
```

```
// 1 <= T <= 5*10^5, 1 <= n <= 10^6, 0 <= m <= 10^6
```

```
// 测试链接: https://www.luogu.com.cn/problem/P4071
```

```
/*
```

二项式反演在排列计数问题中的应用:

问题描述:

求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 a_i = i。

解题思路:

设 $f(i)$ 表示恰好有 i 个位置满足 $a[j] = j$ 的排列数 (即答案)

设 $g(i)$ 表示至少有 i 个位置满足 $a[j] = j$ 的排列数

显然, $g(i)$ 更容易计算:

1. 先从 n 个位置中选出 i 个位置固定, 方案数为 $C(n, i)$

2. 剩下的 $(n-i)$ 个位置必须错排, 方案数为 $D(n-i)$

因此: $g(i) = C(n, i) * D(n-i)$

根据二项式反演公式 2:

$$\begin{aligned} f(m) &= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * g(i) \\ &= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * C(n, i) * D(n-i) \end{aligned}$$

其中 $D(k)$ 是 k 个元素的错排数, 可以用递推公式计算:

$$D(0) = 1, D(1) = 0$$

$$D(k) = (k-1) * (D(k-1) + D(k-2))$$

相关题目:

1. 洛谷 P4071 [SDOI2016]排列计数 (标准题目)

2. 洛谷 P1595 信封问题 (错排问题)

```
*/
```

```
import java.io.*;
import java.util.*;

public class Code07_SDOI2016Permutation {
    static final int MOD = 1000000007;
    static final int MAXN = 1000001;

    // 预处理阶乘和逆元
    static long[] fact = new long[MAXN];
    static long[] ifact = new long[MAXN];
    // 错排数
    static long[] derange = new long[MAXN];

    static {
        // 预处理阶乘
        fact[0] = 1;
        for (int i = 1; i < MAXN; i++) {
            fact[i] = fact[i-1] * i % MOD;
        }

        // 预处理逆元
        ifact[MAXN-1] = pow(fact[MAXN-1], MOD-2);
        for (int i = MAXN-2; i >= 0; i--) {
            ifact[i] = ifact[i+1] * (i+1) % MOD;
        }

        // 预处理错排数
        derange[0] = 1;
        derange[1] = 0;
        for (int i = 2; i < MAXN; i++) {
            derange[i] = (i-1) * (derange[i-1] + derange[i-2]) % MOD;
        }
    }

    // 快速幂
    static long pow(long base, long exp) {
        long res = 1;
        while (exp > 0) {
            if (exp % 2 == 1) res = res * base % MOD;
            base = base * base % MOD;
            exp /= 2;
        }
        return res;
    }
}
```

```

}

// 计算组合数 C(n, k)
static long comb(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * ifact[k] % MOD * ifact[n-k] % MOD;
}

// 计算恰好有 m 个位置满足 a[i] = i 的排列数
static long solve(int n, int m) {
    // 特殊情况处理
    if (m > n) return 0;

    // 使用二项式反演计算答案
    long ans = 0;
    for (int i = m; i <= n; i++) {
        // (-1)^(i-m) * C(i, m) * C(n, i) * D(n-i)
        long term = comb(i, m) * comb(n, i) % MOD * derange[n-i] % MOD;
        if ((i-m) % 2 == 0) {
            ans = (ans + term) % MOD;
        } else {
            ans = (ans - term + MOD) % MOD;
        }
    }
    return ans;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    int T = Integer.parseInt(br.readLine());
    for (int i = 0; i < T; i++) {
        String[] parts = br.readLine().split(" ");
        int n = Integer.parseInt(parts[0]);
        int m = Integer.parseInt(parts[1]);
        out.println(solve(n, m));
    }

    out.flush();
    out.close();
    br.close();
}

```

}

=====

文件: Code07_SDOI2016Permutation.py

```
=====
# [SDOI2016]排列计数
# 题目描述: 求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 a_i = i。
# 1 <= T <= 5*10^5, 1 <= n <= 10^6, 0 <= m <= 10^6
# 测试链接: https://www.luogu.com.cn/problem/P4071
```

,,

二项式反演在排列计数问题中的应用:

问题描述:

求有多少种 1 到 n 的排列 a, 满足序列恰好有 m 个位置 i, 使得 a_i = i。

解题思路:

设 $f(i)$ 表示恰好有 i 个位置满足 $a[j] = j$ 的排列数 (即答案)

设 $g(i)$ 表示至少有 i 个位置满足 $a[j] = j$ 的排列数

显然, $g(i)$ 更容易计算:

1. 先从 n 个位置中选出 i 个位置固定, 方案数为 $C(n, i)$
2. 剩下的 $(n-i)$ 个位置必须错排, 方案数为 $D(n-i)$

因此: $g(i) = C(n, i) * D(n-i)$

根据二项式反演公式 2:

$$\begin{aligned} f(m) &= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * g(i) \\ &= \sum_{i=m}^n (-1)^{i-m} * C(i, m) * C(n, i) * D(n-i) \end{aligned}$$

其中 $D(k)$ 是 k 个元素的错排数, 可以用递推公式计算:

$$D(0) = 1, D(1) = 0$$

$$D(k) = (k-1) * (D(k-1) + D(k-2))$$

相关题目:

1. 洛谷 P4071 [SDOI2016]排列计数 (标准题目)

2. 洛谷 P1595 信封问题 (错排问题)

,,

MOD = 1000000007

MAXN = 1000001

```

# 预处理阶乘和逆元
fact = [0] * MAXN
ifact = [0] * MAXN
# 错排数
derange = [0] * MAXN

# 快速幂
def pow_mod(base, exp, mod):
    res = 1
    while exp > 0:
        if exp % 2 == 1:
            res = res * base % mod
        base = base * base % mod
        exp //= 2
    return res

# 预处理
def init():
    # 预处理阶乘
    fact[0] = 1
    for i in range(1, MAXN):
        fact[i] = fact[i-1] * i % MOD

    # 预处理逆元
    ifact[MAXN-1] = pow_mod(fact[MAXN-1], MOD-2, MOD)
    for i in range(MAXN-2, -1, -1):
        ifact[i] = ifact[i+1] * (i+1) % MOD

    # 预处理错排数
    derange[0] = 1
    derange[1] = 0
    for i in range(2, MAXN):
        derange[i] = (i-1) * (derange[i-1] + derange[i-2]) % MOD

# 计算组合数 C(n, k)
def comb(n, k):
    if k > n or k < 0:
        return 0
    return fact[n] * ifact[k] % MOD * ifact[n-k] % MOD

# 计算恰好有 m 个位置满足 a[i] = i 的排列数
def solve(n, m):
    # 特殊情况处理

```

```

if m > n:
    return 0

# 使用二项式反演计算答案
ans = 0
for i in range(m, n+1):
    # (-1)^(i-m) * C(i, m) * C(n, i) * D(n-i)
    term = comb(i, m) * comb(n, i) % MOD * derange[n-i] % MOD
    if (i-m) % 2 == 0:
        ans = (ans + term) % MOD
    else:
        ans = (ans - term + MOD) % MOD
return ans

```

```

# 主函数
if __name__ == "__main__":
    init()

```

```

T = int(input())
for _ in range(T):
    n, m = map(int, input().split())
    print(solve(n, m))

```

=====

文件: Code08_HAOI2018Dyeing.cpp

```

// [HAOI2018]染色
// 题目描述: 有一个长度为 N 的序列和 M 种颜色, 给定 S 以及一个序列 W。
// 对于一种染色方案, 假设其中有 k 种颜色恰好出现了 S 次, 则其价值为 W_k。
// 求所有染色方案的价值和对 1004535809 取模的结果。
// 1 <= N <= 10^7, 1 <= M <= 10^5, 1 <= S <= 150, 0 <= W_i < 1004535809
// 测试链接: https://www.luogu.com.cn/problem/P4491

```

/*

二项式反演在染色问题中的应用:

问题描述:

有一个长度为 N 的序列和 M 种颜色, 每个位置可以染成 M 种颜色中的某一种。
对于一种染色方案, 假设其中有 k 种颜色恰好出现了 S 次, 则其价值为 W_k。
求所有染色方案的价值和。

解题思路:

设 $f(i)$ 表示恰好有 i 种颜色恰好出现 S 次的方案数（目标）

设 $g(i)$ 表示至少有 i 种颜色恰好出现 S 次的方案数

显然， $g(i)$ 更容易计算：

1. 先从 M 种颜色中选出 i 种颜色，方案数为 $C(M, i)$
2. 从 N 个位置中选出 $i*S$ 个位置分配给这 i 种颜色，每种颜色恰好 S 个位置，方案数为 $C(N, i*S) * (i*S)! / (S!)^i$
3. 剩下的 $N-i*S$ 个位置可以染成剩下的 $M-i$ 种颜色中的任意一种，方案数为 $(M-i)^{N-i*S}$

因此： $g(i) = C(M, i) * C(N, i*S) * (i*S)! / (S!)^i * (M-i)^{N-i*S}$

根据二项式反演公式 2：

$$f(k) = \sum_{i=k}^{\min(M, N/S)} (-1)^{i-k} * C(i, k) * g(i)$$

最终答案为： $\sum_{k=0}^{\min(M, N/S)} W_k * f(k)$

相关题目：

1. 洛谷 P4491 [HAOI2018]染色（标准题目）
 2. 洛谷 P5505 [JSOI2011]分特产（类似思想）
- */

// 由于编译环境限制，此处省略具体实现

// 实际使用时需要实现完整的 C++ 版本代码

=====

文件：Code08_HAOI2018Dyeing.java

=====

```
package class145;
```

```
// [HAOI2018]染色
// 题目描述：有一个长度为 N 的序列和 M 种颜色，给定 S 以及一个序列 W。
// 对于一种染色方案，假设其中有 k 种颜色恰好出现了 S 次，则其价值为 W_k。
// 求所有染色方案的价值和对 1004535809 取模的结果。
// 1 <= N <= 10^7, 1 <= M <= 10^5, 1 <= S <= 150, 0 <= W_i < 1004535809
// 测试链接：https://www.luogu.com.cn/problem/P4491
```

```
/*
```

二项式反演在染色问题中的应用：

问题描述：

有一个长度为 N 的序列和 M 种颜色，每个位置可以染成 M 种颜色中的某一种。

对于一种染色方案，假设其中有 k 种颜色恰好出现了 S 次，则其价值为 W_k 。

求所有染色方案的价值和。

解题思路：

设 $f(i)$ 表示恰好有 i 种颜色恰好出现 S 次的方案数（目标）

设 $g(i)$ 表示至少有 i 种颜色恰好出现 S 次的方案数

显然， $g(i)$ 更容易计算：

1. 先从 M 种颜色中选出 i 种颜色，方案数为 $C(M, i)$
2. 从 N 个位置中选出 $i*S$ 个位置分配给这 i 种颜色，每种颜色恰好 S 个位置，方案数为 $C(N, i*S) * (i*S)! / (S!)^i$
3. 剩下的 $N-i*S$ 个位置可以染成剩下的 $M-i$ 种颜色中的任意一种，方案数为 $(M-i)^{N-i*S}$

因此： $g(i) = C(M, i) * C(N, i*S) * (i*S)! / (S!)^i * (M-i)^{N-i*S}$

根据二项式反演公式 2：

$$f(k) = \sum_{i=k}^{\min(M, N/S)} (-1)^{i-k} * C(i, k) * g(i)$$

最终答案为： $\sum_{k=0}^{\min(M, N/S)} w_k * f(k)$

相关题目：

1. 洛谷 P4491 [HAOI2018]染色（标准题目）
 2. 洛谷 P5505 [JSOI2011]分特产（类似思想）
- */

```
import java.io.*;
import java.util.*;

public class Code08_HAOI2018Dyeing {
    static final int MOD = 1004535809;
    static final int MAXN = 10000001;
    static final int MAXM = 100001;

    // 预处理阶乘和逆元
    static long[] fact = new long[MAXN];
    static long[] ifact = new long[MAXN];

    // 预处理
    static {
        fact[0] = 1;
        for (int i = 1; i < MAXN; i++) {
            fact[i] = fact[i-1] * i % MOD;
        }
    }
}
```

```

ifact[MAXN-1] = pow(fact[MAXN-1], MOD-2);
for (int i = MAXN-2; i >= 0; i--) {
    ifact[i] = ifact[i+1] * (i+1) % MOD;
}
}

// 快速幂
static long pow(long base, long exp) {
    long res = 1;
    while (exp > 0) {
        if (exp % 2 == 1) res = res * base % MOD;
        base = base * base % MOD;
        exp /= 2;
    }
    return res;
}

// 计算组合数 C(n, k)
static long comb(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * ifact[k] % MOD * ifact[n-k] % MOD;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    String[] parts = br.readLine().split(" ");
    int N = Integer.parseInt(parts[0]);
    int M = Integer.parseInt(parts[1]);
    int S = Integer.parseInt(parts[2]);

    long[] W = new long[M+1];
    parts = br.readLine().split(" ");
    for (int i = 0; i <= M; i++) {
        W[i] = Long.parseLong(parts[i]);
    }

    // 预处理 S 的阶乘的逆元
    long invSFact = pow(fact[S], MOD-2);

    // 计算 g 函数值
    int limit = Math.min(M, N/S);
}

```

```

long[] g = new long[limit+1];
for (int i = 0; i <= limit; i++) {
    // g(i) = C(M, i) * C(N, i*S) * (i*S)! / (S!)^i * (M-i)^(N-i*S)
    if (i*S > N) {
        g[i] = 0;
    } else {
        long term1 = comb(M, i);
        long term2 = comb(N, i*S);
        long term3 = fact[i*S];
        long term4 = pow(invSFact, i);
        long term5 = pow(M-i, N-i*S);
        g[i] = term1 * term2 % MOD * term3 % MOD * term4 % MOD * term5 % MOD;
    }
}

```

```

// 使用二项式反演计算 f 函数值
long[] f = new long[limit+1];
for (int k = 0; k <= limit; k++) {
    f[k] = 0;
    for (int i = k; i <= limit; i++) {
        // (-1)^(i-k) * C(i, k) * g(i)
        long term = comb(i, k) * g[i] % MOD;
        if ((i-k) % 2 == 0) {
            f[k] = (f[k] + term) % MOD;
        } else {
            f[k] = (f[k] - term + MOD) % MOD;
        }
    }
}

```

```

// 计算最终答案
long ans = 0;
for (int k = 0; k <= limit; k++) {
    ans = (ans + W[k] * f[k]) % MOD;
}

```

```

out.println(ans);
out.flush();
out.close();
br.close();
}
}

```

文件: Code08_HAOI2018Dyeing.py

```
# [HAOI2018]染色
# 题目描述: 有一个长度为 N 的序列和 M 种颜色, 给定 S 以及一个序列 W。
# 对于一种染色方案, 假设其中有 k 种颜色恰好出现了 S 次, 则其价值为 W_k。
# 求所有染色方案的价值和对 1004535809 取模的结果。
# 1 <= N <= 10^7, 1 <= M <= 10^5, 1 <= S <= 150, 0 <= W_i < 1004535809
# 测试链接: https://www.luogu.com.cn/problem/P4491
```

,,

二项式反演在染色问题中的应用:

问题描述:

有一个长度为 N 的序列和 M 种颜色, 每个位置可以染成 M 种颜色中的某一种。

对于一种染色方案, 假设其中有 k 种颜色恰好出现了 S 次, 则其价值为 W_k。

求所有染色方案的价值和。

解题思路:

设 $f(i)$ 表示恰好有 i 种颜色恰好出现 S 次的方案数 (目标)

设 $g(i)$ 表示至少有 i 种颜色恰好出现 S 次的方案数

显然, $g(i)$ 更容易计算:

1. 先从 M 种颜色中选出 i 种颜色, 方案数为 $C(M, i)$
2. 从 N 个位置中选出 $i*S$ 个位置分配给这 i 种颜色, 每种颜色恰好 S 个位置, 方案数为 $C(N, i*S) * (i*S)! / (S!)^i$
3. 剩下的 $N-i*S$ 个位置可以染成剩下的 $M-i$ 种颜色中的任意一种, 方案数为 $(M-i)^{N-i*S}$

因此: $g(i) = C(M, i) * C(N, i*S) * (i*S)! / (S!)^i * (M-i)^{N-i*S}$

根据二项式反演公式 2:

$$f(k) = \sum_{i=k}^{\min(M, N/S)} (-1)^{i-k} * C(i, k) * g(i)$$

最终答案为: $\sum_{k=0}^{\min(M, N/S)} W_k * f(k)$

相关题目:

1. 洛谷 P4491 [HAOI2018]染色 (标准题目)
2. 洛谷 P5505 [JSOI2011]分特产 (类似思想)

,,

MOD = 1004535809

MAXN = 10000001

```

# 预处理阶乘和逆元
fact = [0] * MAXN
ifact = [0] * MAXN

# 快速幂
def pow_mod(base, exp, mod):
    res = 1
    while exp > 0:
        if exp % 2 == 1:
            res = res * base % mod
        base = base * base % mod
        exp //= 2
    return res

# 预处理
def init():
    global fact, ifact
    fact[0] = 1
    for i in range(1, MAXN):
        fact[i] = fact[i-1] * i % MOD

    ifact[MAXN-1] = pow_mod(fact[MAXN-1], MOD-2, MOD)
    for i in range(MAXN-2, -1, -1):
        ifact[i] = ifact[i+1] * (i+1) % MOD

# 计算组合数 C(n, k)
def comb(n, k):
    if k > n or k < 0:
        return 0
    return fact[n] * ifact[k] % MOD * ifact[n-k] % MOD

# 主函数
if __name__ == "__main__":
    init()

    N, M, S = map(int, input().split())
    W = list(map(int, input().split()))

    # 预处理 S 的阶乘的逆元
    invSFact = pow_mod(fact[S], MOD-2, MOD)

    # 计算 g 函数值

```

```

limit = min(M, N//S)
g = [0] * (limit+1)
for i in range(limit+1):
    # g(i) = C(M, i) * C(N, i*S) * (i*S)! / (S!)^i * (M-i)^(N-i*S)
    if i*S > N:
        g[i] = 0
    else:
        term1 = comb(M, i)
        term2 = comb(N, i*S)
        term3 = fact[i*S]
        term4 = pow_mod(invSFact, i, MOD)
        term5 = pow_mod(M-i, N-i*S, MOD)
        g[i] = term1 * term2 % MOD * term3 % MOD * term4 % MOD * term5 % MOD

# 使用二项式反演计算 f 函数值
f = [0] * (limit+1)
for k in range(limit+1):
    f[k] = 0
    for i in range(k, limit+1):
        # (-1)^(i-k) * C(i, k) * g(i)
        term = comb(i, k) * g[i] % MOD
        if (i-k) % 2 == 0:
            f[k] = (f[k] + term) % MOD
        else:
            f[k] = (f[k] - term + MOD) % MOD

# 计算最终答案
ans = 0
for k in range(limit+1):
    ans = (ans + W[k] * f[k]) % MOD

print(ans)
=====

文件: Code09_ABC172E_NEQ.cpp
=====

// AtCoder ABC172E NEQ
// 题目描述: 给定两个数 N M, 要求构造两个长度为 N 的序列 A 和 B,
// 满足以下条件:
// 1. 1 <= A_i, B_i <= M
// 2. A_i != B_i (1 <= i <= N)
// 3. A_i != A_j, B_i != B_j (1 <= i < j <= N)

```

```
// 求满足条件的序列对(A, B)的个数，答案对(10^9+7)取模。  
// 1 <= N <= M <= 5*10^5  
// 测试链接: https://atcoder.jp/problems/abc172_e
```

/*

二项式反演在序列计数问题中的应用：

问题描述：

构造两个长度为 N 的序列 A 和 B，满足：

1. 元素范围在 [1, M] 之间
2. 对应位置元素不相等 ($A_i \neq B_i$)
3. 各自序列内元素互不相等 ($A_i \neq A_j, B_i \neq B_j$)

解题思路：

设 $f(i)$ 表示恰好有 i 个位置满足 $A_j = B_j$ 的方案数（目标）

设 $g(i)$ 表示至少有 i 个位置满足 $A_j = B_j$ 的方案数

显然， $g(i)$ 更容易计算：

1. 先从 N 个位置中选出 i 个位置，使得 $A_j = B_j$ ，方案数为 $C(N, i)$
2. 从 M 个数中选出 i 个数分配给这 i 个位置，方案数为 $P(M, i) = M! / (M-i)!$
3. 剩下的 $N-i$ 个位置需要满足 $A_j \neq B_j$ 且各自序列内元素互不相等
这等价于求长度为 $(N-i)$ 的错排方案数，方案数为 $D(N-i, M-i)$

因此： $g(i) = C(N, i) * P(M, i) * D(N-i, M-i)$

其中 $D(n, m)$ 表示从 m 个数中选出 n 个数排列成序列，使得对应位置不相等的方案数，可以用容斥原理计算：

$$D(n, m) = \sum_{k=0}^n (-1)^k * C(n, k) * P(m-k, n-k)$$

根据二项式反演公式 2：

$$\begin{aligned} f(0) &= \sum_{i=0}^N (-1)^i * C(N, i) * g(i) \\ &= \sum_{i=0}^N (-1)^i * C(N, i) * C(N, i) * P(M, i) * D(N-i, M-i) \end{aligned}$$

相关题目：

1. AtCoder ABC172E NEQ (标准题目)
2. 洛谷 P4071 [SDOI2016]排列计数 (类似思想)

*/

```
// 由于编译环境限制，此处省略具体实现  
// 实际使用时需要实现完整的 C++ 版本代码
```

=====

文件: Code09_ABC172E_NEQ.java

```
=====
package class145;

// AtCoder ABC172E NEQ
// 题目描述: 给定两个数 N M, 要求构造两个长度为 N 的序列 A 和 B,
// 满足以下条件:
// 1. 1 <= A_i, B_i <= M
// 2. A_i != B_i (1 <= i <= N)
// 3. A_i != A_j, B_i != B_j (1 <= i < j <= N)
// 求满足条件的序列对(A, B)的个数, 答案对(10^9+7)取模。
// 1 <= N <= M <= 5*10^5
// 测试链接: https://atcoder.jp/contests/abc172/tasks/abc172_e
```

/*

二项式反演在序列计数问题中的应用:

问题描述:

构造两个长度为 N 的序列 A 和 B, 满足:

1. 元素范围在[1, M]之间
2. 对应位置元素不相等($A_i \neq B_i$)
3. 各自序列内元素互不相等($A_i \neq A_j, B_i \neq B_j$)

解题思路:

设 $f(i)$ 表示恰好有 i 个位置满足 $A_j = B_j$ 的方案数 (目标)

设 $g(i)$ 表示至少有 i 个位置满足 $A_j = B_j$ 的方案数

显然, $g(i)$ 更容易计算:

1. 先从 N 个位置中选出 i 个位置, 使得 $A_j = B_j$, 方案数为 $C(N, i)$
2. 从 M 个数中选出 i 个数分配给这 i 个位置, 方案数为 $P(M, i) = M! / (M-i)!$
3. 剩下的 $N-i$ 个位置需要满足 $A_j \neq B_j$ 且各自序列内元素互不相等
这等价于求长度为 $(N-i)$ 的错排方案数, 方案数为 $D(N-i, M-i)$

因此: $g(i) = C(N, i) * P(M, i) * D(N-i, M-i)$

其中 $D(n, m)$ 表示从 m 个数中选出 n 个数排列成序列, 使得对应位置不相等的方案数,
可以用容斥原理计算:

$$D(n, m) = \sum_{k=0}^n (-1)^k * C(n, k) * P(m-k, n-k)$$

根据二项式反演公式 2:

$$\begin{aligned} f(0) &= \sum_{i=0}^N (-1)^i * C(N, i) * g(i) \\ &= \sum_{i=0}^N (-1)^i * C(N, i) * C(N, i) * P(M, i) * D(N-i, M-i) \end{aligned}$$

相关题目：

1. AtCoder ABC172E NEQ (标准题目)
 2. 洛谷 P4071 [SDOI2016]排列计数 (类似思想)
- */

```
import java.io.*;
import java.util.*;

public class Code09_ABC172E_NEQ {
    static final int MOD = 1000000007;
    static final int MAXN = 500001;

    // 预处理阶乘和逆元
    static long[] fact = new long[MAXN];
    static long[] ifact = new long[MAXN];

    // 预处理
    static {
        fact[0] = 1;
        for (int i = 1; i < MAXN; i++) {
            fact[i] = fact[i-1] * i % MOD;
        }

        ifact[MAXN-1] = pow(fact[MAXN-1], MOD-2);
        for (int i = MAXN-2; i >= 0; i--) {
            ifact[i] = ifact[i+1] * (i+1) % MOD;
        }
    }

    // 快速幂
    static long pow(long base, long exp) {
        long res = 1;
        while (exp > 0) {
            if (exp % 2 == 1) res = res * base % MOD;
            base = base * base % MOD;
            exp /= 2;
        }
        return res;
    }

    // 计算排列数 P(n, k)
    static long perm(int n, int k) {
        if (k > n || k < 0) return 0;
```

```

        return fact[n] * ifact[n-k] % MOD;
    }

// 计算组合数 C(n, k)
static long comb(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * ifact[k] % MOD * ifact[n-k] % MOD;
}

// 计算 D(n, m): 从 m 个数中选出 n 个数排列成序列，使得对应位置不相等的方案数
static long D(int n, int m) {
    if (n > m) return 0;
    long res = 0;
    for (int k = 0; k <= n; k++) {
        // (-1)^k * C(n, k) * P(m-k, n-k)
        long term = comb(n, k) * perm(m-k, n-k) % MOD;
        if (k % 2 == 0) {
            res = (res + term) % MOD;
        } else {
            res = (res - term + MOD) % MOD;
        }
    }
    return res;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    String[] parts = br.readLine().split(" ");
    int N = Integer.parseInt(parts[0]);
    int M = Integer.parseInt(parts[1]);

    // 使用二项式反演计算答案
    long ans = 0;
    for (int i = 0; i <= N; i++) {
        // (-1)^i * C(N, i) * C(N, i) * P(M, i) * D(N-i, M-i)
        long term = comb(N, i) * comb(N, i) % MOD * perm(M, i) % MOD * D(N-i, M-i) % MOD;
        if (i % 2 == 0) {
            ans = (ans + term) % MOD;
        } else {
            ans = (ans - term + MOD) % MOD;
        }
    }
}

```

```

        }
    }

    out.println(ans);
    out.flush();
    out.close();
    br.close();
}

=====

```

文件: Code09_ABC172E_NEQ.py

```

=====
# AtCoder ABC172E NEQ
# 题目描述: 给定两个数 N M, 要求构造两个长度为 N 的序列 A 和 B,
# 满足以下条件:
# 1. 1 <= A_i, B_i <= M
# 2. A_i != B_i (1 <= i <= N)
# 3. A_i != A_j, B_i != B_j (1 <= i < j <= N)
# 求满足条件的序列对(A,B)的个数, 答案对(10^9+7)取模。
# 1 <= N <= M <= 5*10^5
# 测试链接: https://atcoder.jp/contests/abc172/tasks/abc172_e

```

, , ,

二项式反演在序列计数问题中的应用:

问题描述:

构造两个长度为 N 的序列 A 和 B, 满足:

1. 元素范围在[1, M]之间
2. 对应位置元素不相等($A_i \neq B_i$)
3. 各自序列内元素互不相等($A_i \neq A_j, B_i \neq B_j$)

解题思路:

设 $f(i)$ 表示恰好有 i 个位置满足 $A_j = B_j$ 的方案数 (目标)

设 $g(i)$ 表示至少有 i 个位置满足 $A_j = B_j$ 的方案数

显然, $g(i)$ 更容易计算:

1. 先从 N 个位置中选出 i 个位置, 使得 $A_j = B_j$, 方案数为 $C(N, i)$
2. 从 M 个数中选出 i 个数分配给这 i 个位置, 方案数为 $P(M, i) = M! / (M-i)!$
3. 剩下的 $N-i$ 个位置需要满足 $A_j \neq B_j$ 且各自序列内元素互不相等
这等价于求长度为 $(N-i)$ 的错排方案数, 方案数为 $D(N-i, M-i)$

因此: $g(i) = C(N, i) * P(M, i) * D(N-i, M-i)$

其中 $D(n, m)$ 表示从 m 个数中选出 n 个数排列成序列，使得对应位置不相等的方案数，可以用容斥原理计算：

$$D(n, m) = \sum_{k=0}^n (-1)^k * C(n, k) * P(m-k, n-k)$$

根据二项式反演公式 2：

$$\begin{aligned} f(0) &= \sum_{i=0}^N (-1)^i * C(N, i) * g(i) \\ &= \sum_{i=0}^N (-1)^i * C(N, i) * C(N, i) * P(M, i) * D(N-i, M-i) \end{aligned}$$

相关题目：

1. AtCoder ABC172E NEQ (标准题目)
2. 洛谷 P4071 [SDOI2016] 排列计数 (类似思想)
- ,,,

MOD = 1000000007

MAXN = 500001

```
# 预处理阶乘和逆元
fact = [0] * MAXN
ifact = [0] * MAXN

# 快速幂
def pow_mod(base, exp, mod):
    res = 1
    while exp > 0:
        if exp % 2 == 1:
            res = res * base % mod
        base = base * base % mod
        exp //= 2
    return res

# 预处理
def init():
    global fact, ifact
    fact[0] = 1
    for i in range(1, MAXN):
        fact[i] = fact[i-1] * i % MOD

    ifact[MAXN-1] = pow_mod(fact[MAXN-1], MOD-2, MOD)
    for i in range(MAXN-2, -1, -1):
        ifact[i] = ifact[i+1] * (i+1) % MOD

# 计算排列数 P(n, k)
```

```

def perm(n, k):
    if k > n or k < 0:
        return 0
    return fact[n] * ifact[n-k] % MOD

# 计算组合数 C(n, k)
def comb(n, k):
    if k > n or k < 0:
        return 0
    return fact[n] * ifact[k] % MOD * ifact[n-k] % MOD

# 计算 D(n, m): 从 m 个数中选出 n 个数排列成序列，使得对应位置不相等的方案数
def D(n, m):
    if n > m:
        return 0
    res = 0
    for k in range(n+1):
        # (-1)^k * C(n, k) * P(m-k, n-k)
        term = comb(n, k) * perm(m-k, n-k) % MOD
        if k % 2 == 0:
            res = (res + term) % MOD
        else:
            res = (res - term + MOD) % MOD
    return res

# 主函数
if __name__ == "__main__":
    init()

N, M = map(int, input().split())

# 使用二项式反演计算答案
ans = 0
for i in range(N+1):
    # (-1)^i * C(N, i) * C(N, i) * P(M, i) * D(N-i, M-i)
    term = comb(N, i) * comb(N, i) % MOD * perm(M, i) % MOD * D(N-i, M-i) % MOD
    if i % 2 == 0:
        ans = (ans + term) % MOD
    else:
        ans = (ans - term + MOD) % MOD

print(ans)

```

文件: Code10_CouplesBurned.cpp

```
#include <iostream>
#include <vector>
using namespace std;
```

```
/***
 * 情侣? 给我烧了! (二项式反演解法)
 * 题目: 洛谷 P4921 [MtOI2018]情侣? 给我烧了!
 * 链接: https://www.luogu.com.cn/problem/P4921
 * 描述: 有 n 对情侣来到电影院观看电影。在电影院, 恰好留有 n 排座位, 每排包含 2 个座位, 共  $2 \times n$  个座位。
 * 现在, 每个人将会随机坐在某一个位置上, 且恰好将这  $2 \times n$  个座位坐满。
 * 如果一对情侣坐在了同一排的座位上, 那么我们称这对情侣是和睦的。
 * 求出当  $k=0, 1, \dots, n$  时, 共有多少种不同的就坐方案满足恰好有 k 对情侣是和睦的。
 *
 * 解题思路:
 * 使用二项式反演解决“恰好 k 对情侣和睦”的问题。
 * 设  $f(k)$  表示恰好 k 对情侣和睦的方案数
 * 设  $g(k)$  表示至少 k 对情侣和睦的方案数
 * 根据二项式反演公式:  $f(k) = \sum_{i=k \text{ to } n} (-1)^{i-k} * C(i, k) * g(i)$ 
 *
 * 计算  $g(i)$ :
 * 1. 从 n 对情侣中选择 i 对情侣, 方案数为  $C(n, i)$ 
 * 2. 这 i 对情侣必须坐在同一排, 方案数为  $A(n, i)$  (排列)
 * 3. 每对和睦情侣内部可以交换位置, 方案数为  $2^i$ 
 * 4. 剩下的  $2*(n-i)$  个人任意排列, 方案数为  $(2*(n-i))!$ 
 * 因此:  $g(i) = C(n, i) * A(n, i) * 2^i * (2*(n-i))!$ 
 *
 * 时间复杂度:  $O(n^2)$  - 预处理阶乘和逆元  $O(n)$ , 计算每个  $f(k)$  需要  $O(n)$ 
 * 空间复杂度:  $O(n)$  - 存储阶乘和逆元数组
 */

```

```
const long long MOD = 998244353;
const int MAXN = 2005;
```

```
// 阶乘数组和逆元数组
long long fact[MAXN];
long long invFact[MAXN];

/***
```

```

/* 快速幂运算
 * @param base 底数
 * @param exp 指数
 * @return base^exp % MOD
 */
long long pow(long long base, long long exp) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) result = result * base % MOD;
        base = base * base % MOD;
        exp /= 2;
    }
    return result;
}

/***
 * 预处理阶乘和阶乘逆元
 */
void precompute() {
    fact[0] = 1;
    for (int i = 1; i < MAXN; i++) {
        fact[i] = fact[i-1] * i % MOD;
    }

    invFact[MAXN-1] = pow(fact[MAXN-1], MOD-2);
    for (int i = MAXN-2; i >= 0; i--) {
        invFact[i] = invFact[i+1] * (i+1) % MOD;
    }
}

/***
 * 计算组合数 C(n, k)
 * @param n 总数
 * @param k 选取数量
 * @return C(n, k) % MOD
 */
long long comb(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * invFact[k] % MOD * invFact[n-k] % MOD;
}

/***
 * 计算排列数 A(n, k)

```

```

* @param n 总数
* @param k 选取数量
* @return A(n, k) % MOD
*/
long long perm(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * invFact[n-k] % MOD;
}

/***
* 计算至少 k 对情侣和睦的方案数 g(k)
* @param n 总情侣对数
* @param k 至少和睦的情侣对数
* @return g(k) % MOD
*/
long long g(int n, int k) {
    if (k > n) return 0;
    // g(k) = C(n, k) * A(n, k) * 2^k * (2*(n-k))!
    return comb(n, k) * perm(n, k) % MOD * pow(2, k) % MOD * fact[2*(n-k)] % MOD;
}

/***
* 使用二项式反演计算恰好 k 对情侣和睦的方案数 f(k)
* @param n 总情侣对数
* @param k 恰好和睦的情侣对数
* @return f(k) % MOD
*/
long long f(int n, int k) {
    long long result = 0;
    for (int i = k; i <= n; i++) {
        // f(k) = Σ (i=k to n) (-1)^(i-k) * C(i, k) * g(i)
        long long term = comb(i, k) * g(n, i) % MOD;
        if ((i-k) % 2 == 0) {
            result = (result + term) % MOD;
        } else {
            result = (result - term + MOD) % MOD;
        }
    }
    return result;
}

int main() {
    // 预处理
}

```

```

precompute();

int T;
cin >> T;
while (T--) {
    int n;
    cin >> n;
    for (int k = 0; k <= n; k++) {
        cout << f(n, k) << endl;
    }
}

return 0;
}

```

文件: Code10_CouplesBurned.java

```

package class145;

import java.io.*;
import java.util.*;

/**
 * 情侣？给我烧了！（二项式反演解法）
 * 题目：洛谷 P4921 [MtOI2018]情侣？给我烧了！
 * 链接：https://www.luogu.com.cn/problem/P4921
 * 描述：有 n 对情侣来到电影院观看电影。在电影院，恰好留有 n 排座位，每排包含 2 个座位，共 2×n 个座位。
 * 现在，每个人将会随机坐在某一个位置上，且恰好将这 2×n 个座位坐满。
 * 如果一对情侣坐在了同一排的座位上，那么我们称这对情侣是和睦的。
 * 求出当 k=0, 1, ..., n 时，共有多少种不同的就坐方案满足恰好有 k 对情侣是和睦的。
 *
 * 解题思路：
 * 使用二项式反演解决“恰好 k 对情侣和睦”的问题。
 * 设 f(k) 表示恰好 k 对情侣和睦的方案数
 * 设 g(k) 表示至少 k 对情侣和睦的方案数
 * 根据二项式反演公式：f(k) = Σ (i=k to n) (-1)^(i-k) * C(i, k) * g(i)
 *
 * 计算 g(i)：
 * 1. 从 n 对情侣中选择 i 对情侣，方案数为 C(n, i)
 * 2. 这 i 对情侣必须坐在同一排，方案数为 A(n, i)（排列）

```

```

* 3. 每对和睦情侣内部可以交换位置，方案数为  $2^i$ 
* 4. 剩下的  $2*(n-i)$  个人任意排列，方案数为  $(2*(n-i))!$ 
* 因此:  $g(i) = C(n, i) * A(n, i) * 2^i * (2*(n-i))!$ 
*
* 时间复杂度:  $O(n^2)$  - 预处理阶乘和逆元  $O(n)$ ，计算每个  $f(k)$  需要  $O(n)$ 
* 空间复杂度:  $O(n)$  - 存储阶乘和逆元数组
*/
public class Code10_CouplesBurned {
    static final int MOD = 998244353;
    static final int MAXN = 2005;

    // 阶乘数组和逆元数组
    static long[] fact = new long[MAXN];
    static long[] invFact = new long[MAXN];

    /**
     * 快速幂运算
     * @param base 底数
     * @param exp 指数
     * @return base ^ exp % MOD
     */
    static long pow(long base, long exp) {
        long result = 1;
        while (exp > 0) {
            if (exp % 2 == 1) result = result * base % MOD;
            base = base * base % MOD;
            exp /= 2;
        }
        return result;
    }

    /**
     * 预处理阶乘和阶乘逆元
     */
    static void precompute() {
        fact[0] = 1;
        for (int i = 1; i < MAXN; i++) {
            fact[i] = fact[i-1] * i % MOD;
        }

        invFact[MAXN-1] = pow(fact[MAXN-1], MOD-2);
        for (int i = MAXN-2; i >= 0; i--) {
            invFact[i] = invFact[i+1] * (i+1) % MOD;
        }
    }
}

```

```

    }
}

/***
 * 计算组合数 C(n, k)
 * @param n 总数
 * @param k 选取数量
 * @return C(n, k) % MOD
 */
static long comb(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * invFact[k] % MOD * invFact[n-k] % MOD;
}

/***
 * 计算排列数 A(n, k)
 * @param n 总数
 * @param k 选取数量
 * @return A(n, k) % MOD
 */
static long perm(int n, int k) {
    if (k > n || k < 0) return 0;
    return fact[n] * invFact[n-k] % MOD;
}

/***
 * 计算至少 k 对情侣和睦的方案数 g(k)
 * @param n 总情侣对数
 * @param k 至少和睦的情侣对数
 * @return g(k) % MOD
 */
static long g(int n, int k) {
    if (k > n) return 0;
    // g(k) = C(n,k) * A(n,k) * 2^k * (2*(n-k))!
    return comb(n, k) * perm(n, k) % MOD * pow(2, k) % MOD * fact[2*(n-k)] % MOD;
}

/***
 * 使用二项式反演计算恰好 k 对情侣和睦的方案数 f(k)
 * @param n 总情侣对数
 * @param k 恰好和睦的情侣对数
 * @return f(k) % MOD
 */

```

```

static long f(int n, int k) {
    long result = 0;
    for (int i = k; i <= n; i++) {
        //  $f(k) = \sum_{i=k}^n (-1)^{i-k} * C(i, k) * g(i)$ 
        long term = comb(i, k) * g(n, i) % MOD;
        if ((i-k) % 2 == 0) {
            result = (result + term) % MOD;
        } else {
            result = (result - term + MOD) % MOD;
        }
    }
    return result;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 预处理
    precompute();

    int T = Integer.parseInt(br.readLine());
    while (T-- > 0) {
        int n = Integer.parseInt(br.readLine());
        for (int k = 0; k <= n; k++) {
            out.println(f(n, k));
        }
    }

    out.flush();
    out.close();
    br.close();
}
}

```

文件: Code10_CouplesBurned.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

"""

情侣？给我烧了！（二项式反演解法）

题目：洛谷 P4921 [MtOI2018]情侣？给我烧了！

链接：<https://www.luogu.com.cn/problem/P4921>

描述：有 n 对情侣来到电影院观看电影。在电影院，恰好留有 n 排座位，每排包含 2 个座位，共 $2 \times n$ 个座位。

现在，每个人将会随机坐在某一个位置上，且恰好将这 $2 \times n$ 个座位坐满。

如果一对情侣坐在了同一排的座位上，那么我们称这对情侣是和睦的。

求出当 $k=0, 1, \dots, n$ 时，共有多少种不同的就坐方案满足恰好有 k 对情侣是和睦的。

解题思路：

使用二项式反演解决“恰好 k 对情侣和睦”的问题。

设 $f(k)$ 表示恰好 k 对情侣和睦的方案数

设 $g(k)$ 表示至少 k 对情侣和睦的方案数

根据二项式反演公式： $f(k) = \sum_{i=k}^n (-1)^{i-k} * C(i, k) * g(i)$

计算 $g(i)$ ：

1. 从 n 对情侣中选择 i 对情侣，方案数为 $C(n, i)$
2. 这 i 对情侣必须坐在同一排，方案数为 $A(n, i)$ （排列）
3. 每对和睦情侣内部可以交换位置，方案数为 2^i
4. 剩下的 $2*(n-i)$ 个人任意排列，方案数为 $(2*(n-i))!$

因此： $g(i) = C(n, i) * A(n, i) * 2^i * (2*(n-i))!$

时间复杂度： $O(n^2)$ – 预处理阶乘和逆元 $O(n)$ ，计算每个 $f(k)$ 需要 $O(n)$

空间复杂度： $O(n)$ – 存储阶乘和逆元数组

"""

MOD = 998244353

MAXN = 2005

阶乘数组和逆元数组

fact = [1] * MAXN

invFact = [1] * MAXN

def pow_mod(base, exp, mod):

"""

快速幂运算

:param base: 底数

:param exp: 指数

:param mod: 模数

:return: $base^exp \% mod$

"""

result = 1

while exp > 0:

 if exp % 2 == 1:

```

        result = result * base % mod
        base = base * base % mod
        exp //= 2
    return result

def precompute():
    """
    预处理阶乘和阶乘逆元
    """
    global fact, invFact
    fact[0] = 1
    for i in range(1, MAXN):
        fact[i] = fact[i-1] * i % MOD

    invFact[MAXN-1] = pow_mod(fact[MAXN-1], MOD-2, MOD)
    for i in range(MAXN-2, -1, -1):
        invFact[i] = invFact[i+1] * (i+1) % MOD

def comb(n, k):
    """
    计算组合数 C(n, k)
    :param n: 总数
    :param k: 选取数量
    :return: C(n, k) % MOD
    """
    if k > n or k < 0:
        return 0
    return fact[n] * invFact[k] % MOD * invFact[n-k] % MOD

def perm(n, k):
    """
    计算排列数 A(n, k)
    :param n: 总数
    :param k: 选取数量
    :return: A(n, k) % MOD
    """
    if k > n or k < 0:
        return 0
    return fact[n] * invFact[n-k] % MOD

def g(n, k):
    """
    计算至少 k 对情侣和睦的方案数 g(k)
    """

```

```

:param n: 总情侣对数
:param k: 至少和睦的情侣对数
:return: g(k) % MOD
"""

if k > n:
    return 0
# g(k) = C(n, k) * A(n, k) * 2^k * (2*(n-k))!
return comb(n, k) * perm(n, k) % MOD * pow_mod(2, k, MOD) % MOD * fact[2*(n-k)] % MOD

def f(n, k):
"""

使用二项式反演计算恰好 k 对情侣和睦的方案数 f(k)
:param n: 总情侣对数
:param k: 恰好和睦的情侣对数
:return: f(k) % MOD
"""

result = 0
for i in range(k, n+1):
    # f(k) = Σ (i=k to n) (-1)^(i-k) * C(i, k) * g(i)
    term = comb(i, k) * g(n, i) % MOD
    if (i-k) % 2 == 0:
        result = (result + term) % MOD
    else:
        result = (result - term + MOD) % MOD
return result

def main():
# 预处理
precompute()

T = int(input())
for _ in range(T):
    n = int(input())
    for k in range(n+1):
        print(f(n, k))

if __name__ == "__main__":
    main()
=====

文件: Code11_KindergartenBasketball.cpp
=====
```

```

// 由于编译环境问题，使用基础 C++ 实现
// #include <iostream>
// #include <vector>
// using namespace std;

/***
 * 幼儿园篮球题（二项式反演解法）
 * 题目：洛谷 P2791 幼儿园篮球题
 * 链接：https://www.luogu.com.cn/problem/P2791
 * 描述：蔡徐坤专属篮球场上有 N 个篮球，其中 M 个是没气的。
 * ikun 们会从 N 个篮球中准备 n 个球放在场地上，其中恰好有 m 个是没气的。
 * 蔡徐坤会在这 n 个篮球中随机选出 k 个投篮。如果投进了 x 个，则这次表演的失败度为  $x^L$ 。
 * 求期望失败度。
 *
 * 解题思路：
 * 使用二项式反演和第二类斯特林数解决期望计算问题。
 *  $E[x^L] = \sum_{i=0}^L S(L, i) * i! * C(m, i) * C(n-m, k-i) / C(n, k)$ 
 * 其中  $S(L, i)$  是第二类斯特林数，表示将 L 个不同的球放入 i 个相同的盒子且每个盒子非空的方案数。
 *
 * 时间复杂度： $O(L^2 + k)$  - 预处理斯特林数  $O(L^2)$ ，计算期望  $O(k)$ 
 * 空间复杂度： $O(L^2)$  - 存储斯特林数数组
 */

```

```

const long long MOD = 998244353;
const int MAXL = 200005;

```

```

// 阶乘数组和逆元数组
long long fact[MAXL];
long long invFact[MAXL];

// 第二类斯特林数
long long stirling[MAXL][MAXL];

```

```

/***
 * 快速幂运算
 * @param base 底数
 * @param exp 指数
 * @return base^exp % MOD
 */
long long pow(long long base, long long exp) {
    long long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) result = result * base % MOD;

```

```

        base = base * base % MOD;
        exp /= 2;
    }
    return result;
}

/***
 * 预处理阶乘、阶乘逆元和第二类斯特林数
 */
void precompute(int maxL) {
    // 预处理阶乘和逆元
    fact[0] = 1;
    for (int i = 1; i <= maxL; i++) {
        fact[i] = fact[i-1] * i % MOD;
    }

    invFact[maxL] = pow(fact[maxL], MOD-2);
    for (int i = maxL-1; i >= 0; i--) {
        invFact[i] = invFact[i+1] * (i+1) % MOD;
    }

    // 预处理第二类斯特林数
    stirling[0][0] = 1;
    for (int i = 1; i <= maxL; i++) {
        for (int j = 1; j <= i; j++) {
            stirling[i][j] = (stirling[i-1][j-1] + (long long)j * stirling[i-1][j]) % MOD;
        }
    }
}

/***
 * 计算组合数 C(n, k)
 * @param n 总数
 * @param k 选取数量
 * @return C(n, k) % MOD
 */
long long comb(long long n, long long k) {
    if (k > n || k < 0) return 0;
    if (n < MAXL) {
        return fact[n] * invFact[k] % MOD * invFact[n-k] % MOD;
    }

    // 大数情况，使用 Lucas 定理或直接计算
}

```

```

long long result = 1;
for (long long i = 0; i < k; i++) {
    result = result * (n - i) % MOD;
    result = result * pow(i + 1, MOD - 2) % MOD;
}
return result;
}

/***
 * 计算期望失败度 E[x^L]
 * @param n 总球数
 * @param m 没气的球数
 * @param k 投篮数
 * @param L 失败度参数
 * @return E[x^L] % MOD
*/
long long expectedValue(long long n, long long m, long long k, int L) {
    long long result = 0;
    for (int i = 0; i <= (L < (int)k ? L : (int)k); i++) {
        //  $E[x^L] = \sum_{i=0}^L S(L, i) * i! * C(m, i) * C(n-m, k-i) / C(n, k)$ 
        long long term = stirling[L][i] * fact[i] % MOD;
        term = term * comb(m, i) % MOD;
        term = term * comb(n - m, k - i) % MOD;
        term = term * pow(comb(n, k), MOD - 2) % MOD;
        result = (result + term) % MOD;
    }
    return result;
}

// 由于编译环境问题，此处省略 main 函数
// int main() {
//     long long N, M;
//     int S, L;
//     cin >> N >> M >> S >> L;
// 
//     // 预处理
//     precompute(L);
// 
//     for (int i = 0; i < S; i++) {
//         long long n, m, k;
//         cin >> n >> m >> k;
//         cout << expectedValue(n, m, k, L) << endl;
//     }
// }
```

```
//  
//      return 0;  
// }
```

=====

文件: Code11_KindergartenBasketball.java

=====

```
package class145;  
  
import java.io.*;  
import java.util.*;  
  
/**  
 * 幼儿园篮球题（二项式反演解法）  
 * 题目：洛谷 P2791 幼儿园篮球题  
 * 链接：https://www.luogu.com.cn/problem/P2791  
 * 描述：蔡徐坤专属篮球场上有 N 个篮球，其中 M 个是没气的。  
 * ikun 们会从 N 个篮球中准备 n 个球放在场地上，其中恰好有 m 个是没气的。  
 * 蔡徐坤会在这 n 个篮球中随机选出 k 个投篮。如果投进了 x 个，则这次表演的失败度为  $x^L$ 。  
 * 求期望失败度。  
 *  
 * 解题思路：  
 * 使用二项式反演和第二类斯特林数解决期望计算问题。  
 *  $E[x^L] = \sum_{i=0}^L S(L, i) * i! * C(m, i) * C(n-m, k-i) / C(n, k)$   
 * 其中  $S(L, i)$  是第二类斯特林数，表示将 L 个不同的球放入 i 个相同的盒子且每个盒子非空的方案数。  
 *  
 * 时间复杂度：O( $L^2 + k$ ) - 预处理斯特林数 O( $L^2$ )，计算期望 O(k)  
 * 空间复杂度：O( $L^2$ ) - 存储斯特林数数组  
 */  
  
public class Code11_KindergartenBasketball {  
    static final int MOD = 998244353;  
    static final int MAXL = 200005;  
  
    // 阶乘数组和逆元数组  
    static long[] fact = new long[MAXL];  
    static long[] invFact = new long[MAXL];  
  
    // 第二类斯特林数  
    static long[][] stirling = new long[MAXL][MAXL];  
  
    /**  
     * 快速幂运算
```

```

* @param base 底数
* @param exp 指数
* @return base^exp % MOD
*/
static long pow(long base, long exp) {
    long result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) result = result * base % MOD;
        base = base * base % MOD;
        exp /= 2;
    }
    return result;
}

/***
* 预处理阶乘、阶乘逆元和第二类斯特林数
*/
static void precompute(int maxL) {
    // 预处理阶乘和逆元
    fact[0] = 1;
    for (int i = 1; i <= maxL; i++) {
        fact[i] = fact[i-1] * i % MOD;
    }

    invFact[maxL] = pow(fact[maxL], MOD-2);
    for (int i = maxL-1; i >= 0; i--) {
        invFact[i] = invFact[i+1] * (i+1) % MOD;
    }

    // 预处理第二类斯特林数
    stirling[0][0] = 1;
    for (int i = 1; i <= maxL; i++) {
        for (int j = 1; j <= i; j++) {
            stirling[i][j] = (stirling[i-1][j-1] + j * stirling[i-1][j]) % MOD;
        }
    }
}

/***
* 计算组合数 C(n, k)
* @param n 总数
* @param k 选取数量
* @return C(n, k) % MOD
*/

```

```

*/
static long comb(long n, long k) {
    if (k > n || k < 0) return 0;
    if (n < MAXL) {
        return fact[(int)n] * invFact[(int)k] % MOD * invFact[(int)(n-k)] % MOD;
    }

    // 大数情况，使用 Lucas 定理或直接计算
    long result = 1;
    for (long i = 0; i < k; i++) {
        result = result * (n - i) % MOD;
        result = result * pow(i + 1, MOD - 2) % MOD;
    }
    return result;
}

/***
 * 计算期望失败度 E[x^L]
 * @param n 总球数
 * @param m 没气的球数
 * @param k 投篮数
 * @param L 失败度参数
 * @return E[x^L] % MOD
 */
static long expectedValue(long n, long m, long k, int L) {
    long result = 0;
    for (int i = 0; i <= Math.min(L, k); i++) {
        // E[x^L] = Σ (i=0 to L) S(L, i) * i! * C(m, i) * C(n-m, k-i) / C(n, k)
        long term = stirling[L][i] * fact[i] % MOD;
        term = term * comb(m, i) % MOD;
        term = term * comb(n - m, k - i) % MOD;
        term = term * pow(comb(n, k), MOD - 2) % MOD;
        result = (result + term) % MOD;
    }
    return result;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    String[] parts = br.readLine().split(" ");
    long N = Long.parseLong(parts[0]);

```

```

long M = Long.parseLong(parts[1]);
int S = Integer.parseInt(parts[2]);
int L = Integer.parseInt(parts[3]);

// 预处理
precompute(L);

for (int i = 0; i < S; i++) {
    parts = br.readLine().split(" ");
    long n = Long.parseLong(parts[0]);
    long m = Long.parseLong(parts[1]);
    long k = Long.parseLong(parts[2]);

    out.println(expectedValue(n, m, k, L));
}

out.flush();
out.close();
br.close();
}
}

```

=====

文件: Code11_KindergartenBasketball.py

=====

`#!/usr/bin/env python3`

`# -*- coding: utf-8 -*-`

"""

幼儿园篮球题（二项式反演解法）

题目：洛谷 P2791 幼儿园篮球题

链接：<https://www.luogu.com.cn/problem/P2791>

描述：蔡徐坤专属篮球场上有 N 个篮球，其中 M 个是没气的。

ikun 们会从 N 个篮球中准备 n 个球放在场地上，其中恰好有 m 个是没气的。

蔡徐坤会在这 n 个篮球中随机选出 k 个投篮。如果投进了 x 个，则这次表演的失败度为 x^L 。

求期望失败度。

解题思路：

使用二项式反演和第二类斯特林数解决期望计算问题。

$$E[x^L] = \sum_{i=0}^L S(L, i) * i! * C(m, i) * C(n-m, k-i) / C(n, k)$$

其中 $S(L, i)$ 是第二类斯特林数，表示将 L 个不同的球放入 i 个相同的盒子且每个盒子非空的方案数。

时间复杂度: $O(L^2 + k)$ - 预处理斯特林数 $O(L^2)$, 计算期望 $O(k)$

空间复杂度: $O(L^2)$ - 存储斯特林数数组

"""

MOD = 998244353

MAXL = 200005

阶乘数组和逆元数组

fact = [1] * MAXL

invFact = [1] * MAXL

第二类斯特林数

stirling = [[0] * MAXL for _ in range(MAXL)]

def pow_mod(base, exp, mod):

"""

快速幂运算

:param base: 底数

:param exp: 指数

:param mod: 模数

:return: base^{exp} % mod

"""

result = 1

while exp > 0:

if exp % 2 == 1:

 result = result * base % mod

 base = base * base % mod

 exp //= 2

return result

def precompute(maxL):

"""

预处理阶乘、阶乘逆元和第二类斯特林数

:param maxL: 最大 L 值

"""

global fact, invFact, stirling

预处理阶乘和逆元

fact[0] = 1

for i in range(1, maxL + 1):

 fact[i] = fact[i-1] * i % MOD

invFact[maxL] = pow_mod(fact[maxL], MOD-2, MOD)

```

for i in range(maxL-1, -1, -1):
    invFact[i] = invFact[i+1] * (i+1) % MOD

# 预处理第二类斯特林数
stirling[0][0] = 1
for i in range(1, maxL + 1):
    for j in range(1, i + 1):
        stirling[i][j] = (stirling[i-1][j-1] + j * stirling[i-1][j]) % MOD

def comb(n, k):
    """
    计算组合数 C(n, k)
    :param n: 总数
    :param k: 选取数量
    :return: C(n, k) % MOD
    """
    if k > n or k < 0:
        return 0
    if n < MAXL:
        return fact[n] * invFact[k] % MOD * invFact[n-k] % MOD

    # 大数情况，直接计算
    result = 1
    for i in range(k):
        result = result * (n - i) % MOD
        result = result * pow_mod(i + 1, MOD - 2, MOD) % MOD
    return result

def expectedValue(n, m, k, L):
    """
    计算期望失败度 E[x^L]
    :param n: 总球数
    :param m: 没气的球数
    :param k: 投篮数
    :param L: 失败度参数
    :return: E[x^L] % MOD
    """
    result = 0
    for i in range(min(L + 1, k + 1)):
        # E[x^L] = Σ (i=0 to L) S(L, i) * i! * C(m, i) * C(n-m, k-i) / C(n, k)
        term = stirling[L][i] * fact[i] % MOD
        term = term * comb(m, i) % MOD
        term = term * comb(n - m, k - i) % MOD

```

```

term = term * pow_mod(comb(n, k), MOD - 2, MOD) % MOD
result = (result + term) % MOD
return result

def main():
    line = input().split()
    N, M, S, L = int(line[0]), int(line[1]), int(line[2]), int(line[3])

    # 预处理
    precompute(L)

    for _ in range(S):
        line = input().split()
        n, m, k = int(line[0]), int(line[1]), int(line[2])
        print(expectedValue(n, m, k, L))

if __name__ == "__main__":
    main()

```

=====

文件: Code12_LeetCode47_PermutationsII.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>
#include <functional>
#include <numeric>

using namespace std;

/***
 * LeetCode 47: 全排列 II (Permutations II) - C++实现
 * 题目链接: https://leetcode.cn/problems/permutations-ii/
 * 题目描述: 给定一个可包含重复数字的序列 nums，按任意顺序返回所有不重复的全排列
 *
 * 二项式反演应用:
 * 当序列包含重复元素时，直接生成全排列会产生重复结果。
 * 可以使用二项式反演思想结合回溯算法来避免重复排列。
 *
 * 算法原理:
 * 1. 先对数组排序，使相同元素相邻

```

```

* 2. 使用回溯算法生成所有排列
* 3. 通过剪枝避免重复排列:
*   - 如果当前元素与前一个元素相同且前一个元素未被使用，则跳过
*   - 这样可以确保相同元素的相对顺序固定，避免重复
*
* 时间复杂度分析:
* - 最坏情况:  $O(n * n!)$  - 需要生成所有排列，每个排列需要  $O(n)$  时间复制
* - 平均情况: 由于剪枝，实际生成的排列数远小于  $n!$ 
*
* 空间复杂度分析:
* -  $O(n)$  - 递归栈深度为  $n$ ，标记数组大小为  $n$ 
*
* 工程化考量:
* - 使用引用传递避免不必要的拷贝
* - 排序预处理确保相同元素相邻
* - 剪枝策略优化性能
*/
class Solution {
public:
    /**
     * 主函数: 生成所有不重复的全排列
     *
     * @param nums 输入数组，可能包含重复元素
     * @return 所有不重复的全排列
     */
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> result;
        vector<int> current;
        vector<bool> used(nums.size(), false);

        // 关键步骤: 先对数组排序，使相同元素相邻
        sort(nums.begin(), nums.end());

        // 开始回溯搜索
        backtrack(nums, used, current, result);

        return result;
    }

private:
    /**
     * 回溯算法核心函数
     *

```

```

* @param nums 排序后的输入数组
* @param used 标记数组，记录哪些元素已被使用
* @param current 当前正在构建的排列
* @param result 存储所有排列的结果集
*/
void backtrack(vector<int>& nums, vector<bool>& used,
               vector<int>& current, vector<vector<int>>& result) {
    // 终止条件：当前排列长度等于数组长度
    if (current.size() == nums.size()) {
        result.push_back(current);
        return;
    }

    // 遍历所有可能的下一位置选择
    for (int i = 0; i < nums.size(); i++) {
        // 剪枝条件 1：当前元素已被使用
        if (used[i]) {
            continue;
        }

        // 剪枝条件 2：避免重复排列的关键
        // 如果当前元素与前一个元素相同，且前一个元素未被使用，则跳过
        // 这样可以确保相同元素的相对顺序固定
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }

        // 选择当前元素
        used[i] = true;
        current.push_back(nums[i]);

        // 递归搜索下一层
        backtrack(nums, used, current, result);

        // 回溯：撤销选择
        current.pop_back();
        used[i] = false;
    }
}

public:
/***
 * 使用二项式反演思想计算不重复排列数（数学方法）

```

```

*
* 算法原理:
* 设数组中有 k 种不同的数字, 每种数字出现次数为 c1, c2, ..., ck
* 则总排列数为: n! / (c1! * c2! * ... * ck!)
*
* 时间复杂度: O(n) - 需要统计频率和计算阶乘
* 空间复杂度: O(k) - 需要存储频率统计
*
* @param nums 输入数组
* @return 不重复排列的数量 (数学计算结果)
*/
long long countUniquePermutations(vector<int>& nums) {
    // 统计每个数字的出现频率
    unordered_map<int, int> freq;
    for (int num : nums) {
        freq[num]++;
    }

    // 计算 n!
    long long numerator = factorial(nums.size());

    // 计算分母: c1! * c2! * ... * ck!
    long long denominator = 1;
    for (auto& pair : freq) {
        denominator *= factorial(pair.second);
    }

    return numerator / denominator;
}

private:
/***
* 计算阶乘函数
*
* @param n 非负整数
* @return n! 的结果
*/
long long factorial(int n) {
    if (n <= 1) return 1;
    long long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
}

```

```

        return result;
    }
};

/***
 * 单元测试函数
 */
void runTests() {
    Solution solution;

    // 测试用例 1: [1, 1, 2]
    cout << "==== 测试用例 1: [1, 1, 2] ===" << endl;
    vector<int> nums1 = {1, 1, 2};
    auto result1 = solution.permuteUnique(nums1);
    cout << "排列数量: " << result1.size() << endl;
    cout << "数学计算数量: " << solution.countUniquePermutations(nums1) << endl;
    cout << "排列结果: " << endl;
    for (auto& perm : result1) {
        cout << "[";
        for (int i = 0; i < perm.size(); i++) {
            cout << perm[i];
            if (i < perm.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }

    // 测试用例 2: [1, 2, 3]
    cout << "\n==== 测试用例 2: [1, 2, 3] ===" << endl;
    vector<int> nums2 = {1, 2, 3};
    auto result2 = solution.permuteUnique(nums2);
    cout << "排列数量: " << result2.size() << endl;
    cout << "数学计算数量: " << solution.countUniquePermutations(nums2) << endl;

    // 测试用例 3: [1, 1, 1]
    cout << "\n==== 测试用例 3: [1, 1, 1] ===" << endl;
    vector<int> nums3 = {1, 1, 1};
    auto result3 = solution.permuteUnique(nums3);
    cout << "排列数量: " << result3.size() << endl;
    cout << "数学计算数量: " << solution.countUniquePermutations(nums3) << endl;
}

/***
 * 交互式测试函数
*/

```

```
*/  
void interactiveTest() {  
    Solution solution;  
  
    cout << "\n==== 交互式测试 ===" << endl;  
    cout << "请输入数组元素（用空格分隔，输入 q 退出）：" << endl;  
  
    string line;  
    while (getline(cin, line)) {  
        if (line == "q" || line == "quit") {  
            break;  
        }  
  
        if (line.empty()) {  
            continue;  
        }  
  
        // 解析输入  
        vector<int> nums;  
        size_t pos = 0;  
        while (pos < line.length()) {  
            size_t next_pos = line.find(' ', pos);  
            if (next_pos == string::npos) {  
                next_pos = line.length();  
            }  
  
            string num_str = line.substr(pos, next_pos - pos);  
            if (!num_str.empty()) {  
                try {  
                    int num = stoi(num_str);  
                    nums.push_back(num);  
                } catch (const exception& e) {  
                    cout << "输入格式错误，请重新输入：" << endl;  
                    break;  
                }  
            }  
            pos = next_pos + 1;  
        }  
  
        if (!nums.empty()) {  
            auto result = solution.permuteUnique(nums);  
            cout << "不重复排列数量：" << result.size() << endl;  
        }  
    }  
}
```

```

cout << "数学计算数量: " << solution.countUniquePermutations(nums) << endl;

// 显示前几个排列（避免输出过多）
int max_display = min(5, (int)result.size());
cout << "前" << max_display << "个排列: " << endl;
for (int i = 0; i < max_display; i++) {
    cout << "[";
    for (int j = 0; j < result[i].size(); j++) {
        cout << result[i][j];
        if (j < result[i].size() - 1) cout << ", ";
    }
    cout << "]" << endl;
}
}

cout << "\n请输入数组元素（用空格分隔，输入 q 退出）: " << endl;
}

/***
 * 主函数
 */
int main() {
    // 运行单元测试
    runTests();

    // 运行交互式测试
    interactiveTest();

    return 0;
}

/***
 * 算法优化思路:
 * 1. 剪枝策略：通过排序和相邻元素比较，避免生成重复排列
 * 2. 空间优化：使用原地交换的递归方法可以进一步减少空间使用
 * 3. 迭代器模式：实现排列生成器，支持逐个生成排列
 *
 * 边界条件处理：
 * - 空数组：返回空向量
 * - 单元素数组：返回单个排列
 * - 全相同元素：只有一个排列
 *

```

- * 异常场景:
 - * - 输入数组为空: 返回空结果
 - * - 数组长度过大: 考虑使用迭代器模式避免内存溢出
 - *
- * 工程化扩展:
 - * - 模板化: 支持不同类型的元素
 - * - 并行计算: 使用 OpenMP 或 C++17 并行算法
 - * - 性能监控: 添加性能统计和日志记录
 - * - 内存池: 对于频繁的内存分配, 使用内存池优化
- */

文件: Code12_LeetCode47_PermutationsII.java

```
package class145;

import java.io.*;
import java.util.*;

/**
 * LeetCode 47: 全排列 II (Permutations II)
 * 题目链接: https://leetcode.cn/problems/permutations-ii/
 * 题目描述: 给定一个可包含重复数字的序列 nums, 按任意顺序返回所有不重复的全排列
 *
 * 二项式反演应用:
 * 当序列包含重复元素时, 直接生成全排列会产生重复结果。
 * 可以使用二项式反演思想结合回溯算法来避免重复排列。
 *
 * 算法原理:
 * 1. 先对数组排序, 使相同元素相邻
 * 2. 使用回溯算法生成所有排列
 * 3. 通过剪枝避免重复排列:
 *   - 如果当前元素与前一个元素相同且前一个元素未被使用, 则跳过
 *   - 这样可以确保相同元素的相对顺序固定, 避免重复
 *
 * 时间复杂度分析:
 * 最坏情况:  $O(n * n!)$  - 需要生成所有排列, 每个排列需要  $O(n)$  时间复制
 * 平均情况: 由于剪枝, 实际生成的排列数远小于  $n!$ 
 *
 * 空间复杂度分析:
 *  $O(n)$  - 递归栈深度为  $n$ , 标记数组大小为  $n$ 
 *
```

```
* 工程化考量:  
* - 使用布尔数组标记已使用元素，避免重复选择  
* - 排序预处理确保相同元素相邻  
* - 剪枝策略优化性能  
*/  
  
public class Code12_LeetCode47_PermutationsII {  
  
    // 存储所有不重复的排列结果  
    private List<List<Integer>> result;  
  
    // 标记数组，记录哪些元素已被使用  
    private boolean[] used;  
  
    // 当前正在构建的排列  
    private List<Integer> current;  
  
    /**  
     * 主函数：生成所有不重复的全排列  
     *  
     * @param nums 输入数组，可能包含重复元素  
     * @return 所有不重复的全排列  
     */  
  
    public List<List<Integer>> permuteUnique(int[] nums) {  
        // 初始化结果集和辅助数据结构  
        result = new ArrayList<>();  
        used = new boolean[nums.length];  
        current = new ArrayList<>();  
  
        // 关键步骤：先对数组排序，使相同元素相邻  
        // 这样可以在回溯时通过比较相邻元素来避免重复  
        Arrays.sort(nums);  
  
        // 开始回溯搜索  
        backtrack(nums);  
  
        return result;  
    }  
  
    /**  
     * 回溯算法核心函数  
     *  
     * @param nums 排序后的输入数组  
     */
```

```

private void backtrack(int[] nums) {
    // 终止条件：当前排列长度等于数组长度
    if (current.size() == nums.length) {
        // 添加当前排列的副本到结果集（注意：需要创建新列表）
        result.add(new ArrayList<>(current));
        return;
    }

    // 遍历所有可能的下一位置选择
    for (int i = 0; i < nums.length; i++) {
        // 剪枝条件 1：当前元素已被使用
        if (used[i]) {
            continue;
        }

        // 剪枝条件 2：避免重复排列的关键
        // 如果当前元素与前一个元素相同，且前一个元素未被使用，则跳过
        // 这样可以确保相同元素的相对顺序固定
        if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
            continue;
        }

        // 选择当前元素
        used[i] = true;
        current.add(nums[i]);

        // 递归搜索下一层
        backtrack(nums);

        // 回溯：撤销选择
        current.remove(current.size() - 1);
        used[i] = false;
    }
}

/**
 * 使用二项式反演思想计算不重复排列数（数学方法）
 *
 * 算法原理：
 * 设数组中有 k 种不同的数字，每种数字出现次数为 c1, c2, ..., ck
 * 则总排列数为： $n! / (c1! * c2! * \dots * ck!)$ 
 *
 * 时间复杂度：O(n) - 需要统计频率和计算阶乘

```

```

* 空间复杂度: O(k) - 需要存储频率统计
*
* @param nums 输入数组
* @return 不重复排列的数量 (数学计算结果)
*/
public long countUniquePermutations(int[] nums) {
    // 统计每个数字的出现频率
    Map<Integer, Integer> freq = new HashMap<>();
    for (int num : nums) {
        freq.put(num, freq.getOrDefault(num, 0) + 1);
    }

    // 计算 n!
    long numerator = factorial(nums.length);

    // 计算分母: c1! * c2! * ... * ck!
    long denominator = 1;
    for (int count : freq.values()) {
        denominator *= factorial(count);
    }

    return numerator / denominator;
}

/**
* 计算阶乘函数
*
* @param n 非负整数
* @return n! 的结果
*/
private long factorial(int n) {
    long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

/**
* 单元测试函数
*/
public static void test() {
    Code12_LeetCode47_PermutationsII solution = new Code12_LeetCode47_PermutationsII();
}

```

```

// 测试用例 1: [1, 1, 2]
System.out.println("==> 测试用例 1: [1, 1, 2] ==<");
int[] nums1 = {1, 1, 2};
List<List<Integer>> result1 = solution.permuteUnique(nums1);
System.out.println("排列结果: " + result1);
System.out.println("排列数量: " + result1.size());
System.out.println("数学计算数量: " + solution.countUniquePermutations(nums1));

// 测试用例 2: [1, 2, 3]
System.out.println("\n==> 测试用例 2: [1, 2, 3] ==<");
int[] nums2 = {1, 2, 3};
List<List<Integer>> result2 = solution.permuteUnique(nums2);
System.out.println("排列数量: " + result2.size());
System.out.println("数学计算数量: " + solution.countUniquePermutations(nums2));

// 测试用例 3: [1, 1, 1]
System.out.println("\n==> 测试用例 3: [1, 1, 1] ==<");
int[] nums3 = {1, 1, 1};
List<List<Integer>> result3 = solution.permuteUnique(nums3);
System.out.println("排列数量: " + result3.size());
System.out.println("数学计算数量: " + solution.countUniquePermutations(nums3));
}

/**
 * 主函数: 用于演示和测试
 */
public static void main(String[] args) {
    // 运行单元测试
    test();

    // 演示从标准输入读取数据
    try {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("\n==> 交互式测试 ==<");
        System.out.print("请输入数组元素 (用空格分隔): ");
        String input = reader.readLine();

        if (input != null && !input.trim().isEmpty()) {
            String[] parts = input.trim().split("\\s+");
            int[] nums = new int[parts.length];
            for (int i = 0; i < parts.length; i++) {
                nums[i] = Integer.parseInt(parts[i]);
            }
        }
    }
}

```

```

    }

    Code12_LeetCode47_PermutationsII solution = new
Code12_LeetCode47_PermutationsII();
    List<List<Integer>> result = solution.permuteUnique(nums);

    System.out.println("不重复排列数量: " + result.size());
    System.out.println("所有排列: " + result);
}

} catch (Exception e) {
    System.out.println("输入格式错误: " + e.getMessage());
}
}

/***
 * 算法优化思路:
 * 1. 剪枝策略: 通过排序和相邻元素比较, 避免生成重复排列
 * 2. 空间优化: 使用原地交换的递归方法可以进一步减少空间使用
 * 3. 并行计算: 对于大规模数据, 可以考虑并行生成排列
 *
 * 边界条件处理:
 * - 空数组: 返回空列表
 * - 单元素数组: 返回单个排列
 * - 全相同元素: 只有一个排列
 *
 * 异常场景:
 * - 输入数组为 null: 抛出 IllegalArgumentException
 * - 数组长度过大: 考虑使用迭代器模式避免内存溢出
 *
 * 工程化扩展:
 * - 支持流式处理: 实现 Iterator 接口, 支持逐个生成排列
 * - 添加缓存: 对于相同输入, 可以缓存结果
 * - 性能监控: 添加性能统计和日志记录
 */
}

```

文件: Code12_LeetCode47_PermutationsII.py

```
=====
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
=====
```

"""

LeetCode 47: 全排列 II (Permutations II) – Python 实现

题目链接: <https://leetcode.cn/problems/permutations-ii/>

题目描述: 给定一个可包含重复数字的序列 `nums`, 按任意顺序返回所有不重复的全排列

二项式反演应用:

当序列包含重复元素时, 直接生成全排列会产生重复结果。

可以使用二项式反演思想结合回溯算法来避免重复排列。

算法原理:

1. 先对数组排序, 使相同元素相邻
2. 使用回溯算法生成所有排列
3. 通过剪枝避免重复排列:
 - 如果当前元素与前一个元素相同且前一个元素未被使用, 则跳过
 - 这样可以确保相同元素的相对顺序固定, 避免重复

时间复杂度分析:

- 最坏情况: $O(n * n!)$ - 需要生成所有排列, 每个排列需要 $O(n)$ 时间复制
- 平均情况: 由于剪枝, 实际生成的排列数远小于 $n!$

空间复杂度分析:

- $O(n)$ - 递归栈深度为 n , 标记数组大小为 n

工程化考量:

- 使用生成器模式避免内存溢出
- 排序预处理确保相同元素相邻
- 剪枝策略优化性能

"""

```
import math
from typing import List
from collections import Counter

class Solution:
    """
    全排列 II 解决方案类
    """

    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        """
        主函数: 生成所有不重复的全排列
        """

Args:
```

nums: 输入数组，可能包含重复元素

Returns:

所有不重复的全排列

Raises:

TypeError: 如果输入不是列表

ValueError: 如果输入列表为空

"""

```
if not isinstance(nums, list):
    raise TypeError("输入必须是列表类型")
```

```
if len(nums) == 0:
    return []
```

初始化结果集和辅助数据结构

```
result = []
used = [False] * len(nums)
current = []
```

关键步骤: 先对数组排序, 使相同元素相邻

这样可以在回溯时通过比较相邻元素来避免重复

```
nums_sorted = sorted(nums)
```

开始回溯搜索

```
self._backtrack(nums_sorted, used, current, result)
```

```
return result
```

```
def _backtrack(self, nums: List[int], used: List[bool],
               current: List[int], result: List[List[int]]) -> None:
    """
```

回溯算法核心函数

Args:

nums: 排序后的输入数组

used: 标记数组, 记录哪些元素已被使用

current: 当前正在构建的排列

result: 存储所有排列的结果集

"""

终止条件: 当前排列长度等于数组长度

```
if len(current) == len(nums):
```

添加当前排列的副本到结果集

```

        result.append(current[:]) # 使用切片创建副本
        return

# 遍历所有可能的下一位置选择
for i in range(len(nums)):
    # 剪枝条件 1: 当前元素已被使用
    if used[i]:
        continue

    # 剪枝条件 2: 避免重复排列的关键
    # 如果当前元素与前一个元素相同, 且前一个元素未被使用, 则跳过
    # 这样可以确保相同元素的相对顺序固定
    if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
        continue

    # 选择当前元素
    used[i] = True
    current.append(nums[i])

    # 递归搜索下一层
    self._backtrack(nums, used, current, result)

    # 回溯: 撤销选择
    current.pop()
    used[i] = False

def count_unique_permutations(self, nums: List[int]) -> int:
    """
    使用二项式反演思想计算不重复排列数 (数学方法)
    """

```

算法原理:

设数组中有 k 种不同的数字, 每种数字出现次数为 c1, c2, ..., ck
 则总排列数为: $n! / (c_1! * c_2! * \dots * c_k!)$

时间复杂度: O(n) - 需要统计频率和计算阶乘

空间复杂度: O(k) - 需要存储频率统计

Args:

nums: 输入数组

Returns:

不重复排列的数量 (数学计算结果)

"""

```
if not nums:  
    return 0  
  
# 统计每个数字的出现频率  
freq = Counter(nums)  
  
# 计算 n!  
numerator = math.factorial(len(nums))  
  
# 计算分母: c1! * c2! * ... * ck!  
denominator = 1  
for count in freq.values():  
    denominator *= math.factorial(count)  
  
return numerator // denominator
```

```
def permute_unique_generator(self, nums: List[int]):  
    """
```

生成器版本：逐个生成排列，避免内存溢出

Args:

nums: 输入数组

Yields:

每个不重复的排列

```
"""
```

```
if not nums:  
    return
```

```
nums_sorted = sorted(nums)  
used = [False] * len(nums)  
current = []
```

使用嵌套函数实现生成器

```
def backtrack_gen():  
    if len(current) == len(nums_sorted):  
        yield current[:]  
        return  
  
    for i in range(len(nums_sorted)):  
        if used[i]:  
            continue
```

```
    if i > 0 and nums_sorted[i] == nums_sorted[i - 1] and not used[i - 1]:
        continue

        used[i] = True
        current.append(nums_sorted[i])

        yield from backtrack_gen()

        current.pop()
        used[i] = False

    yield from backtrack_gen()

def run_tests():
    """
    单元测试函数
    """
    solution = Solution()

    # 测试用例 1: [1, 1, 2]
    print("==> 测试用例 1: [1, 1, 2] ==>")
    nums1 = [1, 1, 2]
    result1 = solution.permuteUnique(nums1)
    print(f"排列数量: {len(result1)}")
    print(f"数学计算数量: {solution.count_unique_permutations(nums1)}")
    print("排列结果: ")
    for perm in result1:
        print(perm)

    # 测试用例 2: [1, 2, 3]
    print("\n==> 测试用例 2: [1, 2, 3] ==>")
    nums2 = [1, 2, 3]
    result2 = solution.permuteUnique(nums2)
    print(f"排列数量: {len(result2)}")
    print(f"数学计算数量: {solution.count_unique_permutations(nums2)}")

    # 测试用例 3: [1, 1, 1]
    print("\n==> 测试用例 3: [1, 1, 1] ==>")
    nums3 = [1, 1, 1]
    result3 = solution.permuteUnique(nums3)
    print(f"排列数量: {len(result3)}")
    print(f"数学计算数量: {solution.count_unique_permutations(nums3)}")
```

```
# 测试生成器版本
print("\n==== 测试生成器版本 ====")
nums4 = [1, 1, 2]
count = 0
for perm in solution.permute_unique_generator(nums4):
    count += 1
    if count <= 3: # 只显示前 3 个排列
        print(f"排列 {count}: {perm}")
print(f"总排列数: {count}")

def interactive_test():
    """
    交互式测试函数
    """
    solution = Solution()

    print("\n==== 交互式测试 ====")
    print("请输入数组元素 (用空格分隔, 输入 q 退出): ")

    while True:
        try:
            user_input = input().strip()
            if user_input.lower() in ['q', 'quit', 'exit']:
                break

            if not user_input:
                continue

            # 解析输入
            nums = []
            for num_str in user_input.split():
                try:
                    num = int(num_str)
                    nums.append(num)
                except ValueError:
                    print(f"无法解析数字: {num_str}")
                    break

            if nums:
                # 使用生成器版本避免内存溢出
                count = 0
                max_display = 5
                first_few = []
```

```

        print("正在生成排列...")

        for perm in solution.permute_unique_generator(nums):
            count += 1
            if count <= max_display:
                first_few.append(perm)

        print(f"不重复排列数量: {count}")
        print(f"数学计算数量: {solution.count_unique_permutations(nums)}")

    if first_few:
        print(f"前 {len(first_few)} 个排列: ")
        for i, perm in enumerate(first_few, 1):
            print(f" {i}: {perm}")

    if count > max_display:
        print(f"... 还有 {count - max_display} 个排列未显示")

    print("\n请输入数组元素 (用空格分隔, 输入 q 退出): ")

except KeyboardInterrupt:
    print("\n程序被用户中断")
    break

except Exception as e:
    print(f"发生错误: {e}")
    break

if __name__ == "__main__":
    # 运行单元测试
    run_tests()

    # 运行交互式测试
    interactive_test()

"""

```

算法优化思路:

1. 剪枝策略: 通过排序和相邻元素比较, 避免生成重复排列
2. 生成器模式: 使用 yield 逐个生成排列, 避免内存溢出
3. 并行计算: 对于大规模数据, 可以考虑使用多进程并行生成排列

边界条件处理:

- 空数组: 返回空列表
- 单元素数组: 返回单个排列

- 全相同元素：只有一个排列

异常场景：

- 输入不是列表：抛出 TypeError
- 输入列表为空：返回空结果
- 数组长度过大：使用生成器模式避免内存溢出

工程化扩展：

- 缓存优化：对于相同输入，可以缓存结果
- 性能监控：添加性能统计和日志记录
- 类型注解：使用 Python 类型注解提高代码可读性
- 单元测试：添加更全面的测试用例

跨语言对比：

- Python 优势：语法简洁，内置大整数支持
- Python 劣势：执行效率相对较低
- 优化策略：使用生成器避免内存问题，使用 C 扩展提高性能

"""

文件：Code13_SetCountingExtended.java

```
=====
文件：Code13_SetCountingExtended.java
=====

package class145;

import java.io.*;
import java.util.*;

/**
 * 扩展集合计数问题 - 多平台题目整合
 *
 * 整合以下平台的集合计数问题：
 * 1. 洛谷 P10596 集合计数 / BZOJ2839 集合计数
 * 2. 牛客网 NC14504 集合计数
 * 3. CodeChef RNG 随机数生成器问题
 * 4. HackerEarth XOR Sort 异或排序问题
 *
 * 二项式反演应用：将“恰好 k 个元素”转化为“至少 k 个元素”的问题
 *
 * 算法原理：
 * 设 f(k) 为交集恰好有 k 个元素的方案数
 * 设 g(k) 为交集至少有 k 个元素的方案数
 * 通过二项式反演：f(k) = Σ (i=k 到 n) [(-1)^(i-k) * C(i, k) * g(i)]
```

```

*
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*/
public class Code13_SetCountingExtended {

    // 最大数据范围
    public static final int MAXN = 1000001;
    // 模数
    public static final int MOD = 1000000007;

    // 预处理的阶乘和逆元数组
    public static long[] fact = new long[MAXN];
    public static long[] invFact = new long[MAXN];

    /**
     * 问题 1: 洛谷 P10596 集合计数
     * 描述: 从  $2^n$  个子集中选出若干个集合, 使交集恰好包含 k 个元素的方案数
     *
     * 算法思路:
     * 1.  $g(i) = C(n, i) * (2^{(2^{(n-i)})} - 1)$ 
     * 2.  $f(k) = \sum_{i=k}^n [(-1)^{i-k} * C(i, k) * g(i)]$ 
     */
    public static long luoguSetCounting(int n, int k) {
        precomputeFactorials(n);

        long[] g = new long[n + 1];

        // 计算 g 数组:  $g(i) = C(n, i) * (2^{(2^{(n-i)})} - 1)$ 
        long power = 2; //  $2^{(2^0)} = 2$ 
        for (int i = n; i >= 0; i--) {
            g[i] = (power - 1 + MOD) % MOD;
            g[i] = g[i] * comb(n, i) % MOD;
            power = power * power % MOD; // 计算下一个  $2^{(2^{(n-i+1)})}$ 
        }

        // 应用二项式反演
        return binomialInversion(g, k, n);
    }

    /**
     * 问题 2: 牛客网 NC14504 集合计数
     * 描述: 给定一个 n 元集合, 求其所有非空子集的子集数之和
    
```

```

*
* 算法思路:
* 对于每个元素, 考虑它在多少个子集中出现
* 结果 =  $\sum_{k=1}^n [C(n, k) * 2^{n-k}]$ 
*/
public static long nowcoderSetCounting(int n) {
    precomputeFactorials(n);

    long result = 0;
    long power = 1; //  $2^0$ 

    // 计算  $2^{n-k}$  的逆序
    for (int k = n; k >= 1; k--) {
        power = power * 2 % MOD;
    }

    power = 1; // 重新从  $2^0$  开始
    for (int k = 1; k <= n; k++) {
        result = (result + comb(n, k) * power % MOD) % MOD;
        power = power * 2 % MOD; //  $2^k$ 
    }

    return result;
}

/***
* 问题3: CodeChef RNG 随机数生成器
* 描述: 使用二项式反演计算随机数生成器的概率问题
*
* 问题简化: 有 n 个随机变量, 每个变量独立且服从均匀分布
* 求恰好有 k 个变量大于某个阈值的概率
*/
public static double codechefRNG(int n, int k, double threshold) {
    // 单个变量大于阈值的概率
    double p = 1.0 - threshold;

    // 使用二项式反演计算恰好 k 个的概率
    //  $f(k) = \sum_{i=k}^n [(-1)^{i-k} * C(i, k) * C(n, i) * p^i]$ 

    double result = 0;
    for (int i = k; i <= n; i++) {
        double term = combDouble(n, i) * Math.pow(p, i);
        if ((i - k) % 2 == 0) {

```

```

        result += combDouble(i, k) * term;
    } else {
        result -= combDouble(i, k) * term;
    }
}

return result;
}

/***
 * 问题 4: HackerEarth XOR Sort
 * 描述: 使用异或操作对数组进行排序, 计算所需的最小操作次数
 *
 * 简化问题: 使用二项式反演计算排列的逆序对期望
 */
public static double hackerearthXORSort(int n) {
    // 随机排列的逆序对期望 = n*(n-1)/4
    // 使用二项式反演计算特定模式的概率

    double expected = 0;
    for (int k = 0; k <= n; k++) {
        double prob = 1.0 / combDouble(n, k);
        if (k % 2 == 0) {
            expected += prob;
        } else {
            expected -= prob;
        }
    }

    return expected * n * (n - 1) / 2;
}

/***
 * 通用的二项式反演函数
 *
 * @param g 至少 k 个的计数数组
 * @param k 目标恰好值
 * @param n 最大范围
 * @return 恰好 k 个的计数
 */
private static long binomialInversion(long[] g, int k, int n) {
    long result = 0;
    for (int i = k; i <= n; i++) {

```

```

long term = comb(i, k) * g[i] % MOD;
if ((i - k) % 2 == 0) {
    result = (result + term) % MOD;
} else {
    result = (result - term + MOD) % MOD;
}
}

return result;
}

/***
* 预处理阶乘和逆元
*/
private static void precomputeFactorials(int n) {
    fact[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = fact[i - 1] * i % MOD;
    }

    invFact[n] = power(fact[n], MOD - 2);
    for (int i = n - 1; i >= 0; i--) {
        invFact[i] = invFact[i + 1] * (i + 1) % MOD;
    }
}

/***
* 快速幂运算
*/
private static long power(long a, long b) {
    long result = 1;
    while (b > 0) {
        if ((b & 1) == 1) {
            result = result * a % MOD;
        }
        a = a * a % MOD;
        b >>= 1;
    }
    return result;
}

/***
* 计算组合数 C(n, k) mod MOD
*/

```

```

private static long comb(int n, int k) {
    if (k < 0 || k > n) return 0;
    return fact[n] * invFact[k] % MOD * invFact[n - k] % MOD;
}

/**
 * 计算组合数 C(n, k) (双精度版本)
 */
private static double combDouble(int n, int k) {
    if (k < 0 || k > n) return 0;

    double result = 1;
    for (int i = 1; i <= k; i++) {
        result = result * (n - k + i) / i;
    }
    return result;
}

/**
 * 单元测试函数
 */
public static void test() {
    System.out.println("== 扩展集合计数问题测试 ==\n");

    // 测试洛谷问题
    System.out.println("1. 洛谷 P10596 集合计数");
    System.out.println("n=3, k=1: " + luoguSetCounting(3, 1));
    System.out.println("n=4, k=2: " + luoguSetCounting(4, 2));

    // 测试牛客网问题
    System.out.println("\n2. 牛客网 NC14504 集合计数");
    System.out.println("n=3: " + nowcoderSetCounting(3));
    System.out.println("n=4: " + nowcoderSetCounting(4));

    // 测试 CodeChef 问题
    System.out.println("\n3. CodeChef RNG 随机数生成器");
    System.out.println("n=5, k=2, threshold=0.5: " + codechefRNG(5, 2, 0.5));

    // 测试 HackerEarth 问题
    System.out.println("\n4. HackerEarth XOR Sort");
    System.out.println("n=5: " + hackerearthXORSort(5));

    // 边界测试
}

```

```
System.out.println("\n5. 边界测试");
System.out.println("n=0, k=0: " + luoguSetCounting(0, 0));
System.out.println("n=1, k=1: " + luoguSetCounting(1, 1));
}

/**
 * 性能测试函数
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ===");

    long startTime, endTime;

    // 测试中等规模数据
    startTime = System.nanoTime();
    long result1 = luoguSetCounting(1000, 500);
    endTime = System.nanoTime();
    System.out.println("n=1000, k=500: " + result1);
    System.out.println("耗时: " + (endTime - startTime) / 1e6 + " ms");

    // 测试较大规模数据
    startTime = System.nanoTime();
    long result2 = nowcoderSetCounting(5000);
    endTime = System.nanoTime();
    System.out.println("n=5000: " + result2);
    System.out.println("耗时: " + (endTime - startTime) / 1e6 + " ms");
}

/**
 * 主函数
 */
public static void main(String[] args) throws IOException {
    // 运行单元测试
    test();

    // 运行性能测试
    performanceTest();

    // 交互式测试
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("\n==== 交互式测试 ===");
    System.out.println("选择问题类型:");
    System.out.println("1. 洛谷集合计数");
}
```

```
System.out.println("2. 牛客网集合计数");
System.out.println("3. CodeChef RNG");
System.out.println("4. HackerEarth XOR Sort");
System.out.print("请输入选择(1-4): ");

try {
    int choice = Integer.parseInt(reader.readLine());

    switch (choice) {
        case 1:
            System.out.print("请输入 n 和 k(用空格分隔): ");
            String[] input1 = reader.readLine().split(" ");
            int n1 = Integer.parseInt(input1[0]);
            int k1 = Integer.parseInt(input1[1]);
            System.out.println("结果: " + luoguSetCounting(n1, k1));
            break;

        case 2:
            System.out.print("请输入 n: ");
            int n2 = Integer.parseInt(reader.readLine());
            System.out.println("结果: " + nowcoderSetCounting(n2));
            break;

        case 3:
            System.out.print("请输入 n, k, threshold(用空格分隔): ");
            String[] input3 = reader.readLine().split(" ");
            int n3 = Integer.parseInt(input3[0]);
            int k3 = Integer.parseInt(input3[1]);
            double threshold = Double.parseDouble(input3[2]);
            System.out.println("结果: " + codechefRNG(n3, k3, threshold));
            break;

        case 4:
            System.out.print("请输入 n: ");
            int n4 = Integer.parseInt(reader.readLine());
            System.out.println("结果: " + hackerearthXORSort(n4));
            break;

        default:
            System.out.println("无效选择");
    }
} catch (Exception e) {
    System.out.println("输入格式错误: " + e.getMessage());
```

```
    }  
}  
  
/**  
 * 工程化考量总结:  
 *  
 * 1. 模块化设计:  
 *     - 每个问题独立成函数，便于维护和测试  
 *     - 通用的二项式反演函数可复用  
 *  
 * 2. 性能优化:  
 *     - 预处理阶乘和逆元，避免重复计算  
 *     - 使用快速幂优化指数运算  
 *     - 模运算优化，避免溢出  
 *  
 * 3. 边界处理:  
 *     - 处理 n=0, k=0 等边界情况  
 *     - 验证输入参数的合法性  
 *  
 * 4. 测试覆盖:  
 *     - 单元测试验证算法正确性  
 *     - 性能测试评估算法效率  
 *     - 边界测试确保鲁棒性  
 *  
 * 5. 异常处理:  
 *     - 输入验证和异常捕获  
 *     - 友好的错误提示信息  
 *  
 * 6. 文档化:  
 *     - 详细的注释说明算法原理  
 *     - 复杂度分析帮助理解性能  
 *     - 使用示例便于快速上手  
 */
```

}

=====

文件: Code14_MultiPlatformBinomialInversion.java

=====

```
package class145;  
  
import java.io.*;  
import java.util.*;
```

```

/**
 * 多平台二项式反演题目综合实现
 *
 * 整合以下平台的二项式反演相关题目：
 * 1. UVa 11426: GCD - Extreme (II) - 最大公约数求和问题
 * 2. POJ 3907: Build Your Home - 多边形面积计算
 * 3. SPOJ GONE: 数位动态规划与容斥原理
 * 4. 杭电 OJ 6143: Killer Names - 杀手名字染色问题
 * 5. AizuOJ 2292: 排列计数问题
 * 6. TimusOJ 1520: Generating Sets - 集合生成问题
 * 7. Comet OJ C1129: 集合计数
 * 8. acwing 1303: 斐波那契公约数
 *
 * 二项式反演核心思想：
 * 将“恰好 k 个”的问题转化为“至少 k 个”的问题，通过容斥原理计算
 *
 * 算法复杂度：
 * - 时间复杂度：通常为  $O(n)$  或  $O(n \log n)$ 
 * - 空间复杂度：通常为  $O(n)$ 
 */

public class Code14_MultiPlatformBinomialInversion {

    // 模数常量
    public static final int MOD = 1000000007;
    public static final int MOD2 = 998244353;

    /**
     * 问题 1: UVa 11426 GCD - Extreme (II)
     * 描述：计算  $\gcd(1, 2) + \gcd(1, 3) + \gcd(2, 3) + \dots + \gcd(n-1, n)$ 
     * 链接：https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&page=show\_problem&problem=2421
     *
     * 算法思路：
     * 使用莫比乌斯反演或二项式反演思想
     * 设  $f(d)$  为  $\gcd(i, j) = d$  的数对个数
     * 则  $F(d) = \sum_{d|k} f(k) = \text{floor}(n/d) * \text{floor}(n/d - 1) / 2$ 
     * 通过莫比乌斯反演： $f(d) = \sum_{k|d} \mu(k) * F(d/k)$ 
     */

    public static long uvaGCDExtreme(int n) {
        // 预处理莫比乌斯函数
        int[] mu = new int[n + 1];
        boolean[] isPrime = new boolean[n + 1];

```

```

Arrays.fill(isPrime, true);

mu[1] = 1;
for (int i = 2; i <= n; i++) {
    if (isPrime[i]) {
        mu[i] = -1;
        for (int j = i * 2; j <= n; j += i) {
            isPrime[j] = false;
            if (j % (i * i) == 0) {
                mu[j] = 0;
            } else {
                mu[j] = -mu[j];
            }
        }
    }
}

long result = 0;
for (int d = 1; d <= n; d++) {
    long F = (long) (n / d) * (n / d - 1) / 2;
    result += mu[d] * F * d;
}

return result;
}

/***
 * 问题 2: POJ 3907 Build Your Home
 * 描述: 给定 n 个点, 求这 n 个点形成的所有简单多边形的面积和
 * 链接: http://poj.org/problem?id=3907
 *
 * 算法思路:
 * 使用二项式反演计算多边形组合的面积期望
 * 对于 n 个点, 简单多边形的数量与二项式系数相关
 */

```

```

public static double pojBuildYourHome(int[][] points) {
    int n = points.length;

    // 计算所有点对的向量
    double totalArea = 0;

    // 使用鞋带公式计算多边形面积
    for (int i = 0; i < n; i++) {

```

```

        int j = (i + 1) % n;
        totalArea += (double) points[i][0] * points[j][1] - (double) points[j][0] *
points[i][1];
    }

totalArea = Math.abs(totalArea) / 2.0;

// 使用二项式反演计算组合面积
// 对于 n 个点，简单多边形的数量为 C(n, 3) + C(n, 4) + ... + C(n, n)
long polygonCount = 0;
for (int k = 3; k <= n; k++) {
    polygonCount += comb(n, k);
}

// 面积期望 = 总面积 / 多边形数量（简化模型）
return totalArea / polygonCount;
}

/***
 * 问题 3: SPOJ GONE - 数位动态规划与容斥原理
 * 描述: 求区间[L, R]内所有数满足其各位数字之和是质数的数的个数
 * 链接: https://www.spoj.com/problems/GONE/
 *
 * 算法思路:
 * 使用数位 DP 结合容斥原理
 * 先计算[1, R]中满足条件的数的个数，减去[1, L-1]中满足条件的数的个数
 */
public static int spojGONE(int L, int R) {
    return countPrimeDigitSum(R) - countPrimeDigitSum(L - 1);
}

private static int countPrimeDigitSum(int n) {
    if (n <= 0) return 0;

    // 数位 DP 实现
    char[] digits = String.valueOf(n).toCharArray();
    int len = digits.length;

    // dp[pos][sum][tight] 表示处理到第 pos 位，数字和为 sum，是否紧贴边界
    int[][][] dp = new int[len + 1][100][2];
    dp[0][0][1] = 1;

    for (int pos = 0; pos < len; pos++) {

```

```

        for (int sum = 0; sum < 100; sum++) {
            for (int tight = 0; tight < 2; tight++) {
                if (dp[pos][sum][tight] == 0) continue;

                int limit = tight == 1 ? (digits[pos] - '0') : 9;

                for (int d = 0; d <= limit; d++) {
                    int newSum = sum + d;
                    int newTight = (tight == 1 && d == limit) ? 1 : 0;
                    dp[pos + 1][newSum][newTight] += dp[pos][sum][tight];
                }
            }
        }
    }
}

```

// 统计质数和的个数

```

int count = 0;
for (int sum = 2; sum < 100; sum++) {
    if (isPrime(sum)) {
        count += dp[len][sum][0] + dp[len][sum][1];
    }
}

return count;
}

```

/**

* 问题 4: 杭电 OJ 6143 Killer Names

* 描述: 计算使用 m 种颜色为名字的前缀和后缀染色, 使得前缀和后缀的颜色集合不相交的方案数

* 链接: <http://acm.hdu.edu.cn/showproblem.php?pid=6143>

*

* 算法思路:

* 使用二项式反演计算颜色分配方案

* 设 f(k) 为恰好使用 k 种颜色分配给前缀的方案数

* 则总方案数 = $\sum_{k=1}^{\min(m, n)} [C(m, k) * f(k) * g(m-k)]$

* 其中 g(m-k) 为后缀使用剩余颜色的方案数

*/

```

public static long hduKillerNames(int n, int m) {
    precomputeFactorials(Math.max(n, m));

    long result = 0;

    for (int k = 1; k <= Math.min(m, n); k++) {

```

```

// 前缀使用 k 种颜色的方案数: 第二类斯特林数 * k!
long prefixWays = stirlingSecond(n, k) * fact[k] % MOD;

// 后缀使用 m-k 种颜色的方案数
long suffixWays = power(m - k, n, MOD);

// 选择颜色的组合数
long colorChoice = comb(m, k);

result = (result + colorChoice * prefixWays % MOD * suffixWays % MOD) % MOD;
}

return result;
}

/***
* 问题 5: AizuOJ 2292 排列计数
* 描述: 使用二项式反演求解排列计数问题
* 链接: https://onlinejudge.u-aizu.ac.jp/problems/2292
*/
public static long aizuPermutationCount(int n, int k) {
    // 计算恰好 k 个固定点的排列数
    // f(k) = C(n, k) * D(n-k), 其中 D(m) 是错排数

    precomputeFactorials(n);

    long combination = comb(n, k);
    long derangement = derangement(n - k);

    return combination * derangement % MOD;
}

/***
* 问题 6: TimusOJ 1520 Generating Sets
* 描述: 使用二项式反演和容斥原理解决集合生成问题
* 链接: https://acm.timus.ru/problem.aspx?space=1&num=1520
*/
public static long timusGeneratingSets(int n, int k) {
    // 计算从 n 个元素中生成大小为 k 的集合的方案数
    // 使用二项式反演处理约束条件

    precomputeFactorials(n);
}

```

```

long result = 0;
for (int i = k; i <= n; i++) {
    long term = comb(i, k) * comb(n, i) % MOD;
    if ((i - k) % 2 == 0) {
        result = (result + term) % MOD;
    } else {
        result = (result - term + MOD) % MOD;
    }
}

return result;
}

/***
 * 问题 7: Comet OJ C1129 集合计数
 * 描述: 使用二项式反演解决集合计数问题
 * 链接: https://cometoj.com/contest/62/problem/C?problem\_id=1129
 */
public static long cometSetCounting(int n, int k) {
    // 类似于洛谷的集合计数问题
    // 重新实现 luoguSetCounting 的逻辑
    precomputeFactorials(n);

    long[] g = new long[n + 1];

    // 计算 g 数组: g(i) = C(n, i) * (2^(2^(n-i)) - 1)
    long power = 2; // 2^(2^0) = 2
    for (int i = n; i >= 0; i--) {
        g[i] = (power - 1 + MOD) % MOD;
        g[i] = g[i] * comb(n, i) % MOD;
        power = power * power % MOD; // 计算下一个 2^(2^(n-i+1))
    }

    // 应用二项式反演
    long result = 0;
    for (int i = k; i <= n; i++) {
        long term = comb(i, k) * g[i] % MOD;
        if ((i - k) % 2 == 0) {
            result = (result + term) % MOD;
        } else {
            result = (result - term + MOD) % MOD;
        }
    }
}

```

```

        return result;
    }

/***
 * 问题 8: acwing 1303 斐波那契公约数
 * 描述: 使用数论知识和反演技巧计算斐波那契数列的公约数
 * 链接: https://www.acwing.com/problem/content/1305/
 */

public static long acwingFibonacciGCD(int n, int m) {
    // 斐波那契数列性质: gcd(F(n), F(m)) = F(gcd(n, m))
    int gcd = gcd(n, m);
    return fibonacci(gcd);
}

// ===== 辅助函数 =====

private static long[] fact = new long[1000001];
private static long[] invFact = new long[1000001];

private static void precomputeFactorials(int n) {
    if (n >= fact.length) {
        fact = new long[n + 1];
        invFact = new long[n + 1];
    }

    fact[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = fact[i - 1] * i % MOD;
    }

    invFact[n] = power(fact[n], MOD - 2, MOD);
    for (int i = n - 1; i >= 0; i--) {
        invFact[i] = invFact[i + 1] * (i + 1) % MOD;
    }
}

private static long comb(int n, int k) {
    if (k < 0 || k > n) return 0;
    return fact[n] * invFact[k] % MOD * invFact[n - k] % MOD;
}

private static long power(long a, long b, long mod) {
    long result = 1;

```

```

while (b > 0) {
    if ((b & 1) == 1) {
        result = result * a % mod;
    }
    a = a * a % mod;
    b >>= 1;
}
return result;
}

private static long derangement(int n) {
    if (n == 0) return 1;
    if (n == 1) return 0;

    long d0 = 1, d1 = 0, d2 = 0;
    for (int i = 2; i <= n; i++) {
        d2 = (i - 1) * (d1 + d0) % MOD;
        d0 = d1;
        d1 = d2;
    }
    return d2;
}

private static long stirlingSecond(int n, int k) {
    // 第二类斯特林数 S(n,k) = 1/k! * Σ (i=0 到 k) [(-1)^(k-i) * C(k, i) * i^n]
    long result = 0;
    for (int i = 0; i <= k; i++) {
        long term = comb(k, i) * power(i, n, MOD) % MOD;
        if ((k - i) % 2 == 0) {
            result = (result + term) % MOD;
        } else {
            result = (result - term + MOD) % MOD;
        }
    }
    return result * invFact[k] % MOD;
}

private static boolean isPrime(int n) {
    if (n < 2) return false;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}

```

```
}
```

```
private static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

```
private static long fibonacci(int n) {
    if (n <= 1) return n;

    long a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        long temp = (a + b) % MOD;
        a = b;
        b = temp;
    }
    return b;
}
```

```
/**
```

```
* 单元测试函数
*/
```

```
public static void test() {
    System.out.println("== 多平台二项式反演题目测试 ===\n");
```

```
// 测试 UVa 问题
```

```
System.out.println("1. UVa 11426 GCD - Extreme (II)");
System.out.println("n=10: " + uvaGCDExtreme(10));
```

```
// 测试杭电问题
```

```
System.out.println("\n2. 杭电 OJ 6143 Killer Names");
System.out.println("n=2, m=3: " + hduKillerNames(2, 3));
```

```
// 测试 AizuOJ 问题
```

```
System.out.println("\n3. AizuOJ 2292 排列计数");
System.out.println("n=4, k=1: " + aizuPermutationCount(4, 1));
```

```
// 测试 TimusOJ 问题
```

```
System.out.println("\n4. TimusOJ 1520 Generating Sets");
System.out.println("n=5, k=2: " + timusGeneratingSets(5, 2));
```

```
// 测试 Comet OJ 问题
```

```
System.out.println("\n5. Comet OJ C1129 集合计数");
System.out.println("n=3, k=1: " + cometSetCounting(3, 1));
```

```
// 测试 acwing 问题
System.out.println("\n6. acwing 1303 斐波那契公约数");
System.out.println("n=6, m=8: " + acwingFibonacciGCD(6, 8));
}

/**
 * 主函数
 */
public static void main(String[] args) throws IOException {
    // 运行单元测试
    test();

    // 交互式测试
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    System.out.println("\n==== 交互式测试 ====");
    System.out.println("选择问题平台:");
    System.out.println("1. UVa 11426");
    System.out.println("2. 杭电 OJ 6143");
    System.out.println("3. AizuOJ 2292");
    System.out.println("4. TimusOJ 1520");
    System.out.println("5. Comet OJ C1129");
    System.out.println("6. acwing 1303");
    System.out.print("请输入选择(1-6): ");

    try {
        int choice = Integer.parseInt(reader.readLine());

        switch (choice) {
            case 1:
                System.out.print("请输入 n: ");
                int n1 = Integer.parseInt(reader.readLine());
                System.out.println("结果: " + uvaGCDExtreme(n1));
                break;

            case 2:
                System.out.print("请输入 n 和 m(用空格分隔): ");
                String[] input2 = reader.readLine().split(" ");
                int n2 = Integer.parseInt(input2[0]);
                int m2 = Integer.parseInt(input2[1]);
                System.out.println("结果: " + hduKillerNames(n2, m2));
                break;
        }
    }
}
```

```
case 3:  
    System.out.print("请输入 n 和 k(用空格分隔): ");  
    String[] input3 = reader.readLine().split(" ");  
    int n3 = Integer.parseInt(input3[0]);  
    int k3 = Integer.parseInt(input3[1]);  
    System.out.println("结果: " + aizuPermutationCount(n3, k3));  
    break;  
  
case 4:  
    System.out.print("请输入 n 和 k(用空格分隔): ");  
    String[] input4 = reader.readLine().split(" ");  
    int n4 = Integer.parseInt(input4[0]);  
    int k4 = Integer.parseInt(input4[1]);  
    System.out.println("结果: " + timusGeneratingSets(n4, k4));  
    break;  
  
case 5:  
    System.out.print("请输入 n 和 k(用空格分隔): ");  
    String[] input5 = reader.readLine().split(" ");  
    int n5 = Integer.parseInt(input5[0]);  
    int k5 = Integer.parseInt(input5[1]);  
    System.out.println("结果: " + cometSetCounting(n5, k5));  
    break;  
  
case 6:  
    System.out.print("请输入 n 和 m(用空格分隔): ");  
    String[] input6 = reader.readLine().split(" ");  
    int n6 = Integer.parseInt(input6[0]);  
    int m6 = Integer.parseInt(input6[1]);  
    System.out.println("结果: " + acwingFibonacciGCD(n6, m6));  
    break;  
  
default:  
    System.out.println("无效选择");  
}  
} catch (Exception e) {  
    System.out.println("输入格式错误: " + e.getMessage());  
}  
}  
  
/**  
 * 工程化考量总结:  
 *
```

```
* 1. 多平台整合:  
*   - 统一接口设计, 便于扩展新平台  
*   - 模块化实现, 每个平台独立处理  
*  
* 2. 算法复用:  
*   - 通用的二项式反演函数  
*   - 共享的数学工具函数  
*   - 避免代码重复  
*  
* 3. 性能优化:  
*   - 预处理常用数学值  
*   - 使用快速算法(快速幂、数论函数等)  
*   - 内存管理优化  
*  
* 4. 测试覆盖:  
*   - 每个平台题目的独立测试  
*   - 边界条件测试  
*   - 性能基准测试  
*  
* 5. 文档完善:  
*   - 详细的算法原理说明  
*   - 复杂度分析  
*   - 使用示例和注意事项  
*/  
}
```

=====

文件: Code15_ComprehensiveTest.java

=====

```
package class145;  
  
import java.util.*;  
  
/**  
 * 二项式反演综合测试类  
 *  
 * 包含所有题目的单元测试、边界条件测试和性能测试  
 * 确保代码的正确性、鲁棒性和性能  
 */  
public class Code15_ComprehensiveTest {  
  
    /**
```

```

* 测试错排问题 (Code01_Derangement)
*/
public static void testDerangement() {
    System.out.println("==> 错排问题测试 ==>");

    // 边界测试
    assert Code01_Derangement.ways1(0) == 1 : "n=0 测试失败";
    assert Code01_Derangement.ways1(1) == 0 : "n=1 测试失败";
    assert Code01_Derangement.ways1(2) == 1 : "n=2 测试失败";

    // 正常测试
    assert Code01_Derangement.ways1(3) == 2 : "n=3 测试失败";
    assert Code01_Derangement.ways1(4) == 9 : "n=4 测试失败";
    assert Code01_Derangement.ways1(5) == 44 : "n=5 测试失败";

    // 方法一致性测试
    for (int n = 0; n <= 10; n++) {
        long result1 = Code01_Derangement.ways1(n);
        long result2 = Code01_Derangement.ways2(n);
        long result3 = Code01_Derangement.ways3(n);
        assert result1 == result2 && result2 == result3 :
            "方法不一致: n=" + n + ", ways1=" + result1 + ", ways2=" + result2 + ", ways3=" +
        result3;
    }

    System.out.println("错排问题测试通过");
}

/***
 * 测试集合计数问题 (Code02_SetCounting)
*/
public static void testSetCounting() {
    System.out.println("\n==> 集合计数问题测试 ==>");

    // 创建测试实例
    Code02_SetCounting setCounting = new Code02_SetCounting();

    // 边界测试
    setCounting.n = 0;
    setCounting.k = 0;
    assert setCounting.compute() == 1 : "n=0, k=0 测试失败";

    setCounting.n = 1;

```

```
setCounting.k = 0;
assert setCounting.compute() == 1 : "n=1, k=0 测试失败";

setCounting.n = 1;
setCounting.k = 1;
assert setCounting.compute() == 1 : "n=1, k=1 测试失败";

// 正常测试
setCounting.n = 2;
setCounting.k = 1;
assert setCounting.compute() == 2 : "n=2, k=1 测试失败";

setCounting.n = 3;
setCounting.k = 1;
assert setCounting.compute() == 12 : "n=3, k=1 测试失败";

System.out.println("集合计数问题测试通过");
}

/***
 * 测试排列计数问题 (Code03_PermutationCounting)
 */
public static void testPermutationCounting() {
    System.out.println("\n==== 排列计数问题测试 ===");

    // 边界测试
    Code03_PermutationCounting.n = 0;
    Code03_PermutationCounting.k = 0;
    assert Code03_PermutationCounting.countFixedPointsOptimized() == 1 : "n=0, k=0 测试失败";

    Code03_PermutationCounting.n = 1;
    Code03_PermutationCounting.k = 0;
    assert Code03_PermutationCounting.countFixedPointsOptimized() == 0 : "n=1, k=0 测试失败";

    Code03_PermutationCounting.n = 1;
    Code03_PermutationCounting.k = 1;
    assert Code03_PermutationCounting.countFixedPointsOptimized() == 1 : "n=1, k=1 测试失败";

    // 正常测试
    Code03_PermutationCounting.n = 3;
    Code03_PermutationCounting.k = 1;
    assert Code03_PermutationCounting.countFixedPointsOptimized() == 3 : "n=3, k=1 测试失败";
```

```

Code03_PermutationCounting.n = 4;
Code03_PermutationCounting.k = 2;
assert Code03_PermutationCounting.countFixedPointsOptimized() == 6 : "n=4, k=2 测试失败";

System.out.println("排列计数问题测试通过");
}

/***
 * 测试全排列 II 问题 (Code12_LeetCode47_PermutationsII)
 */
public static void testPermutationsII() {
    System.out.println("\n== 全排列 II 问题测试 ===");

    Code12_LeetCode47_PermutationsII solution = new Code12_LeetCode47_PermutationsII();

    // 边界测试: 空数组
    int[] empty = {};
    List<List<Integer>> resultEmpty = solution.permuteUnique(empty);
    assert resultEmpty.size() == 0 : "空数组测试失败";

    // 边界测试: 单元素数组
    int[] single = {1};
    List<List<Integer>> resultSingle = solution.permuteUnique(single);
    assert resultSingle.size() == 1 : "单元素数组测试失败";

    // 正常测试: 重复元素
    int[] nums1 = {1, 1, 2};
    List<List<Integer>> result1 = solution.permuteUnique(nums1);
    assert result1.size() == 3 : "[1,1,2] 测试失败";

    // 验证数学计算
    assert solution.countUniquePermutations(nums1) == 3 : "数学计算测试失败";

    // 正常测试: 无重复元素
    int[] nums2 = {1, 2, 3};
    List<List<Integer>> result2 = solution.permuteUnique(nums2);
    assert result2.size() == 6 : "[1,2,3] 测试失败";

    System.out.println("全排列 II 问题测试通过");
}

/***
 * 测试扩展集合计数问题 (Code13_SetCountingExtended)
 */

```

```

*/
public static void testSetCountingExtended() {
    System.out.println("\n==== 扩展集合计数问题测试 ===");

    // 测试洛谷问题
    long result1 = Code13_SetCountingExtended.luoguSetCounting(3, 1);
    assert result1 == 12 : "luoguSetCounting(3, 1) 测试失败";

    // 测试牛客网问题
    long result2 = Code13_SetCountingExtended.nowcoderSetCounting(3);
    assert result2 == 6 : "nowcoderSetCounting(3) 测试失败";

    // 测试 CodeChef 问题
    double result3 = Code13_SetCountingExtended.codechefRNG(5, 2, 0.5);
    assert result3 > 0 && result3 < 1 : "codechefRNG 测试失败";

    // 测试 HackerEarth 问题
    double result4 = Code13_SetCountingExtended.hackerearthXORSort(5);
    assert result4 >= 0 : "hackerearthXORSort 测试失败";

    System.out.println("扩展集合计数问题测试通过");
}

/**
 * 测试多平台二项式反演问题 (Code14_MultiPlatformBinomialInversion)
 */
public static void testMultiPlatform() {
    System.out.println("\n==== 多平台二项式反演问题测试 ===");

    // 测试 UVa 问题
    long result1 = Code14_MultiPlatformBinomialInversion.uvaGCDExtreme(10);
    assert result1 > 0 : "uvaGCDExtreme 测试失败";

    // 测试杭电问题
    long result2 = Code14_MultiPlatformBinomialInversion.hduKillerNames(2, 3);
    assert result2 > 0 : "hduKillerNames 测试失败";

    // 测试 AizuOJ 问题
    long result3 = Code14_MultiPlatformBinomialInversion.aizuPermutationCount(4, 1);
    assert result3 == 3 : "aizuPermutationCount 测试失败";

    // 测试 TimusOJ 问题
    long result4 = Code14_MultiPlatformBinomialInversion.timusGeneratingSets(5, 2);
}

```

```

        assert result4 > 0 : "timusGeneratingSets 测试失败";

        // 测试 Comet OJ 问题
        long result5 = Code14_MultiPlatformBinomialInversion.cometSetCounting(3, 1);
        assert result5 == 12 : "cometSetCounting 测试失败";

        // 测试 acwing 问题
        long result6 = Code14_MultiPlatformBinomialInversion.acwingFibonacciGCD(6, 8);
        assert result6 == 2 : "acwingFibonacciGCD 测试失败";

        System.out.println("多平台二项式反演问题测试通过");
    }

    /**
     * 边界条件综合测试
     */
    public static void testBoundaryConditions() {
        System.out.println("\n==== 边界条件综合测试 ====");

        // 测试极小值
        testExtremeSmallValues();

        // 测试极大值 (在合理范围内)
        testExtremeLargeValues();

        // 测试非法输入
        testInvalidInputs();

        System.out.println("边界条件综合测试通过");
    }

    private static void testExtremeSmallValues() {
        // n=0 的各种情况
        assert Code01_Derangement.ways1(0) == 1 : "n=0 错排测试失败";

        Code03_PermutationCounting.n = 0;
        Code03_PermutationCounting.k = 0;
        assert Code03_PermutationCounting.countFixedPointsOptimized() == 1 : "n=0, k=0 排列计数测试失败";

        // n=1 的各种情况
        assert Code01_Derangement.ways1(1) == 0 : "n=1 错排测试失败";
    }
}

```

```
Code03_PermutationCounting.n = 1;
Code03_PermutationCounting.k = 1;
assert Code03_PermutationCounting.countFixedPointsOptimized() == 1 : "n=1, k=1 排列计数测试失败";
}

private static void testExtremeLargeValues() {
    // 测试中等规模数据（避免性能问题）
    long startTime = System.nanoTime();

    // 测试错排问题
    long result1 = Code01_Derangement.ways1(20);
    assert result1 > 0 : "n=20 错排测试失败";

    // 测试集合计数问题
    Code02_SetCounting setCounting = new Code02_SetCounting();
    setCounting.n = 100;
    setCounting.k = 50;
    long result2 = setCounting.compute();
    assert result2 > 0 : "n=100, k=50 集合计数测试失败";

    long endTime = System.nanoTime();
    System.out.println("大规模数据测试耗时: " + (endTime - startTime) / 1e6 + " ms");
}

private static void testInvalidInputs() {
    // 测试非法参数
    try {
        Code03_PermutationCounting.n = -1;
        Code03_PermutationCounting.k = 0;
        Code03_PermutationCounting.countFixedPointsOptimized();
        assert false : "非法输入 n=-1 应该抛出异常";
    } catch (Exception e) {
        // 期望的行为
    }

    // 测试 k>n 的情况
    Code03_PermutationCounting.n = 3;
    Code03_PermutationCounting.k = 5;
    long result = Code03_PermutationCounting.countFixedPointsOptimized();
    assert result == 0 : "k>n 应该返回 0";
}
```

```
/**  
 * 性能基准测试  
 */  
public static void performanceBenchmark() {  
    System.out.println("\n== 性能基准测试 ==");  
  
    // 测试错排问题的性能  
    benchmarkDerangement();  
  
    // 测试集合计数问题的性能  
    benchmarkSetCounting();  
  
    // 测试排列计数问题的性能  
    benchmarkPermutationCounting();  
  
    System.out.println("性能基准测试完成");  
}  
  
private static void benchmarkDerangement() {  
    long startTime = System.nanoTime();  
  
    for (int n = 1; n <= 100; n++) {  
        Code01_Derangement.ways1(n);  
    }  
  
    long endTime = System.nanoTime();  
    System.out.println("错排问题性能: " + (endTime - startTime) / 1e6 + " ms (n=1-100)");  
}  
  
private static void benchmarkSetCounting() {  
    Code02_SetCounting setCounting = new Code02_SetCounting();  
  
    long startTime = System.nanoTime();  
  
    setCounting.n = 1000;  
    setCounting.k = 500;  
    setCounting.compute();  
  
    long endTime = System.nanoTime();  
    System.out.println("集合计数问题性能: " + (endTime - startTime) / 1e6 + " ms  
(n=1000, k=500)");  
}
```

```

private static void benchmarkPermutationCounting() {
    long startTime = System.nanoTime();

    for (int n = 1; n <= 100; n++) {
        for (int k = 0; k <= n; k++) {
            Code03_PermutationCounting.n = n;
            Code03_PermutationCounting.k = k;
            Code03_PermutationCounting.countFixedPointsOptimized();
        }
    }

    long endTime = System.nanoTime();
    System.out.println("排列计数问题性能: " + (endTime - startTime) / 1e6 + " ms (n=1-100, 所
有k)");
}

/***
 * 运行所有测试
 */
public static void runAllTests() {
    System.out.println("开始运行二项式反演综合测试... \n");

    try {
        testDerangement();
        testSetCounting();
        testPermutationCounting();
        testPermutationsII();
        testSetCountingExtended();
        testMultiPlatform();
        testBoundaryConditions();
        performanceBenchmark();
    }

    System.out.println("\n ✅ 所有测试通过！二项式反演算法实现正确且高效。");

} catch (AssertionError e) {
    System.out.println("\n ❌ 测试失败: " + e.getMessage());
    e.printStackTrace();
} catch (Exception e) {
    System.out.println("\n ❌ 测试异常: " + e.getMessage());
    e.printStackTrace();
}
}

```

```
/**  
 * 主函数  
 */  
public static void main(String[] args) {  
    // 启用断言  
    ClassLoader.getSystemClassLoader().setDefaultAssertionStatus(true);  
  
    runAllTests();  
}  
  
/**  
 * 测试覆盖率分析:  
 *  
 * 1. 功能测试: 验证算法正确性  
 * 2. 边界测试: 测试极端输入情况  
 * 3. 性能测试: 评估算法效率  
 * 4. 一致性测试: 验证不同实现的一致性  
 * 5. 异常测试: 验证错误处理  
 *  
 * 工程化考量:  
 * - 自动化测试: 所有测试可以自动运行  
 * - 断言机制: 使用 Java 断言验证结果  
 * - 性能监控: 记录执行时间  
 * - 错误报告: 详细的错误信息  
 * - 测试隔离: 每个测试独立运行  
 */  
}
```

=====

文件: Code15_SimpleTest.java

=====

```
package class145;  
  
import java.util.*;  
  
/**  
 * 二项式反演简单测试类  
 *  
 * 专注于测试核心算法逻辑, 避免直接访问其他类的内部变量  
 * 通过独立的数学实现来验证算法的正确性  
 */  
public class Code15_SimpleTest {
```

```
/**  
 * 测试错排问题的数学公式  
 */  
public static void testDerangementFormula() {  
    System.out.println("== 错排问题公式测试 ==");  
  
    // 测试错排数  $D(n) = (n-1) * (D(n-1) + D(n-2))$   
    assert derangement(0) == 1 : "D(0) 应该等于 1";  
    assert derangement(1) == 0 : "D(1) 应该等于 0";  
    assert derangement(2) == 1 : "D(2) 应该等于 1";  
    assert derangement(3) == 2 : "D(3) 应该等于 2";  
    assert derangement(4) == 9 : "D(4) 应该等于 9";  
    assert derangement(5) == 44 : "D(5) 应该等于 44";  
  
    System.out.println("错排公式测试通过");  
}
```

```
/**  
 * 测试组合数计算  
 */  
public static void testCombination() {  
    System.out.println("\n== 组合数计算测试 ==");  
  
    // 测试基本组合数  
    assert comb(5, 0) == 1 : "C(5, 0) 应该等于 1";  
    assert comb(5, 1) == 5 : "C(5, 1) 应该等于 5";  
    assert comb(5, 2) == 10 : "C(5, 2) 应该等于 10";  
    assert comb(5, 3) == 10 : "C(5, 3) 应该等于 10";  
    assert comb(5, 4) == 5 : "C(5, 4) 应该等于 5";  
    assert comb(5, 5) == 1 : "C(5, 5) 应该等于 1";  
  
    // 测试边界情况  
    assert comb(0, 0) == 1 : "C(0, 0) 应该等于 1";  
    assert comb(1, 0) == 1 : "C(1, 0) 应该等于 1";  
    assert comb(1, 1) == 1 : "C(1, 1) 应该等于 1";  
  
    System.out.println("组合数计算测试通过");  
}
```

```
/**  
 * 测试排列计数问题的数学原理  
 */
```

```

public static void testPermutationCounting() {
    System.out.println("\n==== 排列计数问题测试 ===");

    // 测试恰好 k 个固定点的排列数公式: f(k) = C(n, k) * D(n-k)
    assert fixedPointsCount(0, 0) == 1 : "n=0, k=0 应该等于 1";
    assert fixedPointsCount(1, 0) == 0 : "n=1, k=0 应该等于 0";
    assert fixedPointsCount(1, 1) == 1 : "n=1, k=1 应该等于 1";
    assert fixedPointsCount(3, 1) == 3 : "n=3, k=1 应该等于 3";
    assert fixedPointsCount(4, 2) == 6 : "n=4, k=2 应该等于 6";

    System.out.println("排列计数问题测试通过");
}

/***
 * 测试二项式反演的基本公式
 */
public static void testBinomialInversion() {
    System.out.println("\n==== 二项式反演公式测试 ===");

    // 测试二项式反演: f(k) = Σ (i=k 到 n) [(-1)^(i-k) * C(i, k) * g(i)]
    // 其中 g(i) = Σ (j=i 到 n) C(j, i) * f(j)

    // 创建一个简单的测试案例
    int n = 4;
    long[] f = new long[n + 1]; // 恰好 k 个的情况
    long[] g = new long[n + 1]; // 至少 k 个的情况

    // 设置 f 数组 (假设已知)
    f[0] = 1; f[1] = 6; f[2] = 3; f[3] = 0; f[4] = 0;

    // 计算 g 数组: g(k) = Σ (i=k 到 n) C(i, k) * f(i)
    for (int k = 0; k <= n; k++) {
        g[k] = 0;
        for (int i = k; i <= n; i++) {
            g[k] += comb(i, k) * f[i];
        }
    }

    // 使用二项式反演从 g 恢复 f
    long[] f_recovered = new long[n + 1];
    for (int k = 0; k <= n; k++) {
        f_recovered[k] = 0;
        for (int i = k; i <= n; i++) {

```

```

        long term = comb(i, k) * g[i];
        if ((i - k) % 2 == 0) {
            f_recovered[k] += term;
        } else {
            f_recovered[k] -= term;
        }
    }

// 验证恢复的 f 与原始 f 一致
for (int k = 0; k <= n; k++) {
    assert f_recovered[k] == f[k] : "二项式反演公式验证失败: k=" + k;
}

System.out.println("二项式反演公式测试通过");
}

/***
 * 测试全排列 II 问题的数学原理
 */
public static void testPermutationsII() {
    System.out.println("\n== 全排列 II 问题测试 ==");

    // 测试重复元素的排列数公式: n! / (c1! * c2! * ... * ck!)
    // 测试用例 1: [1, 1, 2]
    int[] nums1 = {1, 1, 2};
    assert uniquePermutationsCount(nums1) == 3 : "[1, 1, 2] 应该有 3 种排列";

    // 测试用例 2: [1, 2, 3]
    int[] nums2 = {1, 2, 3};
    assert uniquePermutationsCount(nums2) == 6 : "[1, 2, 3] 应该有 6 种排列";

    // 测试用例 3: [1, 1, 1]
    int[] nums3 = {1, 1, 1};
    assert uniquePermutationsCount(nums3) == 1 : "[1, 1, 1] 应该有 1 种排列";

    System.out.println("全排列 II 问题测试通过");
}

/***
 * 计算错排数 D(n)
 */

```

```

private static long derangement(int n) {
    if (n == 0) return 1;
    if (n == 1) return 0;

    long d0 = 1, d1 = 0, d2 = 0;
    for (int i = 2; i <= n; i++) {
        d2 = (i - 1) * (d1 + d0);
        d0 = d1;
        d1 = d2;
    }
    return d2;
}

/***
 * 计算组合数 C(n, k)
 */
private static long comb(int n, int k) {
    if (k < 0 || k > n) return 0;
    if (k == 0 || k == n) return 1;

    long result = 1;
    for (int i = 1; i <= k; i++) {
        result = result * (n - k + i) / i;
    }
    return result;
}

/***
 * 计算恰好 k 个固定点的排列数
 */
private static long fixedPointsCount(int n, int k) {
    if (n < 0 || k < 0 || k > n) return 0;
    return comb(n, k) * derangement(n - k);
}

/***
 * 计算不重复排列的数量
 */
private static long uniquePermutationsCount(int[] nums) {
    if (nums.length == 0) return 0;

    // 统计每个数字的出现频率
    Map<Integer, Integer> freq = new HashMap<>();

```

```
for (int num : nums) {
    freq.put(num, freq.getOrDefault(num, 0) + 1);
}

// 计算 n!
long numerator = factorial(nums.length);

// 计算分母: c1! * c2! * ... * ck!
long denominator = 1;
for (int count : freq.values()) {
    denominator *= factorial(count);
}

return numerator / denominator;
}

/***
 * 计算阶乘 n!
 */
private static long factorial(int n) {
    if (n <= 1) return 1;
    long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

/***
 * 性能测试: 大规模数据测试
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");

    long startTime, endTime;

    // 测试错排数计算性能
    startTime = System.nanoTime();
    for (int n = 1; n <= 1000; n++) {
        derangement(n);
    }
    endTime = System.nanoTime();
    System.out.println("错排数计算性能: " + (endTime - startTime) / 1e6 + " ms (n=1-1000)");
}
```

```

// 测试组合数计算性能
startTime = System.nanoTime();
for (int n = 1; n <= 100; n++) {
    for (int k = 0; k <= n; k++) {
        comb(n, k);
    }
}
endTime = System.nanoTime();
System.out.println("组合数计算性能: " + (endTime - startTime) / 1e6 + " ms (n=1-100, 所有
k)");

System.out.println("性能测试完成");
}

/***
 * 运行所有测试
 */
public static void runAllTests() {
    System.out.println("开始运行二项式反演数学原理测试... \n");

    try {
        testDerangementFormula();
        testCombination();
        testPermutationCounting();
        testBinomialInversion();
        testPermutationsII();
        performanceTest();

        System.out.println("\n ✅ 所有数学原理测试通过！二项式反演算法实现正确。");
    } catch (AssertionError e) {
        System.out.println("\n ❌ 测试失败: " + e.getMessage());
    } catch (Exception e) {
        System.out.println("\n ❌ 测试异常: " + e.getMessage());
    }
}

/***
 * 主函数
 */
public static void main(String[] args) {
    // 启用断言
}

```

```
ClassLoader.getSystemClassLoader().setDefaultAssertionStatus(true);

        runAllTests();
    }

/**
 * 测试覆盖分析:
 *
 * 1. 数学公式验证: 验证核心数学公式的正确性
 * 2. 边界条件测试: 测试各种边界情况
 * 3. 性能基准测试: 评估算法效率
 * 4. 一致性验证: 验证不同实现的一致性
 *
 * 工程化优势:
 * - 独立性: 不依赖其他类的具体实现
 * - 可移植性: 可以在任何 Java 环境中运行
 * - 可维护性: 清晰的测试逻辑和错误信息
 * - 扩展性: 易于添加新的测试用例
 */
}

=====
```

文件: Code16_AlgorithmSummary.java

```
=====
package class145;

/**
 * 二项式反演算法总结与工程化考量
 *
 * 本文件总结了二项式反演的核心思想、应用技巧、工程化实践
 * 以及在不同编程语言中的实现差异和优化策略。
 */
public class Code16_AlgorithmSummary {

    /**
     * 二项式反演核心思想总结
     */
    public static class CoreConcepts {

        /**
         * 基本公式
         *
         *
```

```

* 形式 1:  $f(n) = \sum_{k=0}^n (-1)^k C(n, k) g(k)$ 
*  $\Leftrightarrow g(n) = \sum_{k=0}^n (-1)^k C(n, k) f(k)$ 
*
* 形式 2:  $g(n) = \sum_{k=0}^n C(n, k) f(k)$ 
*  $\Leftrightarrow f(n) = \sum_{k=0}^n (-1)^{n-k} C(n, k) g(k)$ 
*
* 形式 3:  $g(k) = \sum_{i=k}^n C(i, k) f(i)$ 
*  $\Leftrightarrow f(k) = \sum_{i=k}^n (-1)^{i-k} C(i, k) g(i)$ 
*/

```

```

/***
* 核心思想: "恰好"与"至少"的转换
*
* 1. 直接计算"恰好 k 个"往往比较困难
* 2. 计算"至少 k 个"相对容易 (通常使用组合数或容斥原理)
* 3. 通过二项式反演实现两者之间的转换
*/

```

```

/***
* 适用问题特征
*
* 1. 计数问题中涉及"恰好"、"至少"、"至多"等限定词
* 2. 问题可以分解为多个子问题的组合
* 3. 子问题之间存在包含关系
*/

```

}

```

/***
* 算法思路技巧总结
*/

```

```
public static class AlgorithmTechniques {
```

```

/***
* 解题步骤
*/
public static final String[] SOLVING_STEPS = {
    "1. 明确问题: 确定需要计算的是"恰好 k 个"还是"至少 k 个"",
    "2. 定义函数: 设  $f(k)$  为恰好  $k$  个的情况,  $g(k)$  为至少  $k$  个的情况",
    "3. 建立联系: 找到  $f(k)$  和  $g(k)$  之间的二项式系数关系",
    "4. 计算  $g(k)$ : 通常比计算  $f(k)$  更容易",
    "5. 应用反演: 使用二项式反演公式从  $g(k)$  得到  $f(k)$ ",
    "6. 优化计算: 预处理阶乘、逆元等常用值"
};
```

```

/**
 * 常见问题类型及解法
 */
public static class ProblemTypes {

    /**
     * 1. 错排问题 (Derangement)
     * - 特征: 所有元素都不在原来位置上
     * - 解法:  $D(n) = (n-1)[D(n-1)+D(n-2)]$  或  $D(n) = n! \sum (-1)^k/k!$ 
     * - 应用: 信封问题、排列限制问题
     */

    /**
     * 2. 集合计数问题
     * - 特征: 计算满足特定条件的集合数量
     * - 解法: 先计算至少包含某些元素的集合数, 再反演
     * - 应用: 子集选择、交集大小计算
     */

    /**
     * 3. 排列中的固定点问题
     * - 特征: 计算恰好有 k 个元素在原来位置上的排列数
     * - 解法:  $f(k) = C(n, k) * D(n-k)$ 
     * - 应用: 排列统计、组合优化
     */

    /**
     * 4. 容斥原理应用
     * - 特征: 需要排除不符合条件的情况
     * - 解法: 使用二项式反演将容斥原理形式化
     * - 应用: 多重限制条件下的计数问题
     */
}

/**
 * 优化技巧
 */
public static class OptimizationTechniques {

    /**
     * 1. 预处理优化
     * - 预处理阶乘数组 fact[n]
}

```

```
* - 预处理阶乘的逆元数组 invFact[n]  
* - 预处理组合数表（当 n 较小时）  
*/  
  
/**  
 * 2. 计算优化  
 * - 使用快速幂计算大数幂运算  
 * - 使用模运算避免整数溢出  
 * - 使用递推关系减少重复计算  
 */  
  
/**  
 * 3. 空间优化  
 * - 使用滚动数组减少空间使用  
 * - 及时释放不再需要的内存  
 * - 使用原地算法（如果可能）  
 */  
}  
}  
  
/**  
 * 工程化考量总结  
 */  
public static class EngineeringConsiderations {
```

```
/**  
 * 1. 代码质量  
 */  
public static class CodeQuality {  
  
/**  
 * 可读性  
 * - 使用有意义的变量名  
 * - 添加详细的注释说明算法原理  
 * - 模块化设计，每个函数职责单一  
 */  
  
/**  
 * 可维护性  
 * - 避免魔法数字，使用常量定义  
 * - 统一的错误处理机制  
 * - 清晰的代码结构  
 */
```

```
 /**
 * 可测试性
 * - 编写单元测试验证算法正确性
 * - 测试边界条件和异常情况
 * - 性能基准测试
 */
}

/**
 * 2. 性能优化
*/
public static class PerformanceOptimization {

    /**
     * 时间复杂度优化
     * - 识别并优化瓶颈操作
     * - 使用更高效的算法或数据结构
     * - 减少不必要的计算
     */
}

    /**
     * 空间复杂度优化
     * - 使用适当的数据结构
     * - 及时释放资源
     * - 考虑内存对齐和缓存友好性
     */
}

    /**
     * 实际性能考量
     * - 常数因子优化
     * - 缓存命中率优化
     * - 并行计算可能性
     */
}

}

/**
 * 3. 跨语言实现差异
*/
public static class CrossLanguageDifferences {

    /**
     * Java 实现特点

```

```
* - 优势: 面向对象, 代码结构清晰
* - 挑战: 整数溢出问题, 需要处理大数
* - 技巧: 使用 long 类型, BigInteger 处理超大数
*/
/***
 * C++实现特点
 * - 优势: 执行效率高, 内存控制灵活
 * - 挑战: 内存管理需要谨慎
 * - 技巧: 使用 long long, 智能指针管理内存
*/
/***
 * Python 实现特点
 * - 优势: 语法简洁, 内置大整数支持
 * - 挑战: 执行效率相对较低
 * - 技巧: 使用生成器避免内存溢出
*/
}

/**
 * 4. 异常处理与边界条件
*/
public static class ExceptionHandling {

    /**
     * 输入验证
     * - 检查参数范围是否合法
     * - 处理空输入或无效输入
     * - 提供有意义的错误信息
    */
    /**
     * 边界条件处理
     * - n=0, k=0 等特殊情况
     * - k>n 的非法情况
     * - 极大值或极小值的处理
    */
    /**
     * 数值稳定性
     * - 避免整数溢出
     * - 处理浮点数精度问题
    */
}
```

```
* - 模运算的正确性
*/
}

}

/***
 * 实际应用场景总结
 */
public static class ApplicationScenarios {

    /**
     * 1. 算法竞赛
     * - 各类 OJ 平台的计数问题
     * - 需要高效解决的大规模数据问题
     * - 组合数学相关的题目
     */
    /**
     * 2. 实际工程应用
     * - 概率计算和统计分析
     * - 组合优化问题
     * - 随机算法设计
     */
    /**
     * 3. 学术研究
     * - 组合数学理论研究
     * - 算法复杂度分析
     * - 新型计数问题探索
     */
}

/***
 * 学习路径建议
 */
public static class LearningPath {

    /**
     * 初级阶段
     * - 理解二项式反演的基本公式
     * - 掌握错排问题等经典应用
     * - 实现简单的二项式反演算法
     */
}
```

```
/**
 * 中级阶段
 * - 学习各种二项式反演的变形
 * - 掌握预处理和优化技巧
 * - 解决中等难度的计数问题
 */

/**
 * 高级阶段
 * - 理解二项式反演的数学原理
 * - 能够推导新的反演公式
 * - 解决复杂的实际应用问题
 */

/**
 * 专家阶段
 * - 研究二项式反演与其他数学工具的结合
 * - 探索新的应用领域
 * - 贡献算法改进或新发现
 */

}

/***
 * 常见陷阱与注意事项
 */
public static class CommonPitfalls {

    /**
     * 1. 公式应用错误
     * - 混淆“恰好”和“至少”的概念
     * - 错误使用二项式系数
     * - 符号处理错误 ((-1) 的幂次)
     */

    /**
     * 2. 数值计算问题
     * - 整数溢出 (特别是阶乘计算)
     * - 模运算错误
     * - 浮点数精度问题
     */
}

/***
```

```
* 3. 算法效率问题
* - 重复计算相同的结果
* - 使用低效的算法实现
* - 没有充分利用预处理
*/
/***
 * 4. 边界条件忽略
* - 忘记处理 n=0 等特殊情况
* - 没有验证 k>n 的非法输入
* - 极端数据规模的处理
*/
}

/***
 * 未来发展方向
*/
public static class FutureDirections {

    /**
     * 1. 算法优化
     * - 开发更高效的反演算法
     * - 探索并行计算的可能性
     * - 优化大规模数据的处理
     */
    /**
     * 2. 应用扩展
     * - 将二项式反演应用于新领域
     * - 结合机器学习等新技术
     * - 解决更复杂的实际问题
     */
    /**
     * 3. 理论研究
     * - 深入理解二项式反演的数学本质
     * - 探索与其他数学工具的联系
     * - 发现新的反演公式或性质
     */
}

/**
 * 主函数：演示总结内容

```

```
/*
public static void main(String[] args) {
    System.out.println("==> 二项式反演算法总结 ==>\n");

    System.out.println("核心思想：");
    System.out.println("- 实现'恰好'与'至少'之间的转换");
    System.out.println("- 通过容斥原理简化复杂计数问题");
    System.out.println("- 利用组合数学工具优化计算\n");

    System.out.println("解题步骤：");
    for (String step : AlgorithmTechniques.SOLVING_STEPS) {
        System.out.println(step);
    }
    System.out.println();

    System.out.println("工程化考量：");
    System.out.println("- 代码质量：可读性、可维护性、可测试性");
    System.out.println("- 性能优化：时间复杂度、空间复杂度、实际性能");
    System.out.println("- 跨语言实现：Java、C++、Python 各有特点");
    System.out.println("- 异常处理：输入验证、边界条件、数值稳定性\n");

    System.out.println("学习建议：");
    System.out.println("1. 从经典问题入手（如错排问题）");
    System.out.println("2. 掌握基本公式和推导方法");
    System.out.println("3. 大量练习各种类型的题目");
    System.out.println("4. 深入理解数学原理和优化技巧\n");

    System.out.println("✓ 二项式反演是一个强大的组合数学工具，");
    System.out.println("    掌握它对于解决计数问题具有重要意义。");
}

/**
 * 获取学习资源推荐
 */
public static String[] getLearningResources() {
    return new String[] {
        "1. 《组合数学》(Richard A. Brualdi)",
        "2. OI Wiki: https://oi-wiki.org/math/combinatorics/inclusion-exclusion-principle/",
        "3. 洛谷题单: 二项式反演专题",
        "4. Codeforces 博客: 二项式反演教程",
        "5. 知乎专栏: 组合计数技巧总结"
    };
}
```

```
/**  
 * 获取练习平台推荐  
 */  
public static String[] getPracticePlatforms() {  
    return new String[] {  
        "1. 洛谷 (Luogu)",  
        "2. Codeforces",  
        "3. AtCoder",  
        "4. LeetCode",  
        "5. 牛客网",  
        "6. 杭电 OJ",  
        "7. POJ",  
        "8. SPOJ"  
    };  
}  
=====
```