==========
文件夹: class099_DigitDP
==========

[Markdown 文件]
==========
文件: ADDITIONAL_DIGIT_DP_PROBLEMS.md
==========
数位 DP 扩展题目详解

以下是在学习数位 DP 过程中可以练习的更多题目，涵盖各大 OJ 平台，包含完整的题目描述、解题思路和代码实现。

一、LeetCode 题目

1. 233. 数字 1 的个数
- **题目链接**: https://leetcode.cn/problems/number-of-digit-one/
- **题目描述**: 给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。
- **解题思路**:
  1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
  2. 状态设计需要记录：
     - 当前处理位置
     - 是否受上界限制
     - 当前已出现的 1 的个数
  3. 关键点：统计 1 出现的次数而不是数字个数
- **时间复杂度**: O(log n)
- **空间复杂度**: O(log n)
- **相关文件**: Code05_CountDigitOne.java, Code05_CountDigitOne.py

2. 600. 不含连续 1 的非负整数
- **题目链接**: https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/
- **题目描述**: 给定一个正整数 n，返回在 [0, n] 范围内不含连续 1 的非负整数的个数。
- **解题思路**:
  1. 数位 DP 方法：使用数位 DP 框架，逐位确定二进制数字
  2. 状态设计需要记录：
     - 当前处理位置
     - 是否受上界限制
     - 前一位是否为 1
  3. 关键点：当前位不能与前一位同时为 1
- **时间复杂度**: O(log n)
- **空间复杂度**: O(log n)
- **相关文件**: Code06_NonNegativeIntegersWithoutConsecutiveOnes.java, Code16_NumberWithoutConsecutiveOnes.java

3. 1012. 至少有 1 位重复的数字
- **题目链接**: https://leetcode.cn/problems/numbers-with-repeated-digits/
- **题目描述**: 给定正整数 n，返回在 [1, n] 范围内具有至少 1 位重复数字的正整数的个数。
- **解题思路**:
  1. 补集思想：统计没有重复数字的数字个数，然后用总数减去它
  2. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
  3. 状态设计需要记录：
     - 当前处理位置
     - 是否受上界限制
     - 是否已开始填数字
     - 已使用数字的位掩码
  4. 关键点：用位掩码记录已使用的数字
- **时间复杂度**: O(10 * 2^10 * log n)
- **空间复杂度**: O(2^10 * log n)
- **相关文件**: Code04_NumbersWithRepeatedDigits.java, Code17_NumberWithRepeatedDigits.java

二、Codeforces 题目

1. 55D. Beautiful Numbers
- **题目链接**: https://codeforces.com/problemset/problem/55/D
- **题目描述**: 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。给定区间 [1, r]，求其中美丽数字的个数。
- **解题思路**:
  1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
  2. 状态设计需要记录：
     - 当前处理位置
     - 是否受上界限制
     - 是否已开始填数字
     - 当前数字对 LCM(1-9)的余数
     - 已使用数字的 LCM
  3. 关键优化：1-9 的 LCM 是 2520，所有数字的 LCM 都是 2520 的约数
  4. 关键点：一个数能被其所有非零数字整除等价于这个数能被这些数字的最小公倍数(LCM)整除
- **时间复杂度**: O(log r * 2520 * 50)
- **空间复杂度**: O(log r * 2520 * 50)
- **相关文件**: Code13_BeautifulNumbersCF.java, Code13_BeautifulNumbersCF.cpp, Code13_BeautifulNumbersCF.py

2. 628D. Magic Numbers
- **题目链接**: https://codeforces.com/problemset/problem/628/D
- **题目描述**: 定义一个 d-magic number 为满足特定条件的数字，给定区间 [a,b]，求其中 magic number 的个数。
- **解题思路**:

   1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
   2. 状态设计需要记录：
      - 当前处理位置
      - 是否受上界限制
      - 当前数字对 m 的余数
      - 当前位置的奇偶性
   3. 关键点：根据位置奇偶性应用不同约束
- **时间复杂度**: O(log b * m)
- **空间复杂度**: O(log b * m)
- **相关文件**: Code14_MagicNumbersCF.java, Code14_MagicNumbersCF.cpp, Code14_MagicNumbersCF.py

三、AtCoder 题目

1. ABC135 D - Digits Parade
- **题目链接**: https://atcoder.jp/contests/abc135/tasks/abc135_d
- **题目描述**: 给定一个由数字和?组成的字符串，?可以替换成 0-9 的任意数字，求有多少种替换方案使得结果能被 13 整除。
- **解题思路**:
   1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
   2. 状态设计需要记录：
      - 当前处理位置
      - 当前数字对 13 的余数
   3. 关键点：'?'可以替换为 0-9 的任意数字
- **时间复杂度**: O(n * 13)
- **空间复杂度**: O(n * 13)
- **相关文件**: Code15_DigitsParadeABC.java, Code15_DigitsParadeABC.cpp, Code15_DigitsParadeABC.py

四、洛谷题目

1. P4127 [AHOI2009] 同类分布
- **题目链接**: https://www.luogu.com.cn/problem/P4127
- **题目描述**: 给出两个数 a,b，求出 [a,b] 中各位数字之和能整除原数的数的个数。
- **解题思路**:
   1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
   2. 状态设计需要记录：
      - 当前处理位置
      - 是否受上界限制
      - 是否已开始填数字
      - 当前数位和
      - 当前数值对某个数的余数
   3. 关键点：枚举所有可能的数位和，然后对每个数位和进行数位 DP
- **时间复杂度**: O(log b * 162 * 162)

- **空间复杂度**: O(log b * 162 * 162)
- **相关文件**: Code11_SimilarDistribution.java

2. P2602 [ZJOI2010] 数字计数
- **题目链接**: https://www.luogu.com.cn/problem/P2602
- **题目描述**: 给定两个正整数 a 和 b，求在 [a,b] 范围上的所有整数中，每个数码 (digit) 各出现了多少次。
- **解题思路**:
  1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
  2. 对于每个数字 0-9，分别计算其出现次数
  3. 状态设计需要记录：
     - 当前处理位置
     - 是否受上界限制
     - 是否已开始填数字
     - 当前统计的数字
  4. 关键点：分别统计每个数字的出现次数
- **时间复杂度**: O(log b * 10)
- **空间复杂度**: O(log b * 10)
- **相关文件**: Code09_DigitCount.java, Code09_DigitCount.cpp, Code09_DigitCount.py

五、数位 DP 常见题型总结

1. 基础计数问题
- **特征**: 统计满足简单条件的数字个数
- **例题**: 数字 1 的个数、不含连续 1 的数字
- **状态设计**: 位置 + 限制标记 + 简单计数

2. 数位和问题
- **特征**: 与数字的数位和相关
- **例题**: 数位和在指定范围内、数位和整除原数
- **状态设计**: 位置 + 限制标记 + 数位和 + 余数

3. 数字约束问题
- **特征**: 数字必须满足特定模式
- **例题**: 美丽数字、魔法数字
- **状态设计**: 位置 + 限制标记 + 模式状态

4. 字符串匹配问题
- **特征**: 结合字符串匹配算法
- **例题**: 不包含特定子串的好字符串
- **状态设计**: 位置 + 限制标记 + KMP 状态

5. 二进制数位问题

- **特征**: 处理二进制表示
- **例题**: 不含连续 1 的二进制数
- **状态设计**: 位置 + 限制标记 + 前一位状态

6. 通配符问题
- **特征**: 包含通配符的数字
- **例题**: Digits Parade
- **状态设计**: 位置 + 限制标记 + 余数

六、解题技巧总结

1. 状态设计原则
- **最小化状态数**: 只记录必要的约束信息
- **状态压缩**: 使用位运算压缩多个布尔状态
- **模运算优化**: 利用模运算性质减少状态

2. 记忆化优化
- **缓存键设计**: 合理设计记忆化缓存键
- **状态去重**: 识别并合并相同状态
- **提前剪枝**: 在不可能满足条件时提前返回

3. 边界处理
- **前导零处理**: 正确处理前导零情况
- **上下界处理**: 正确处理区间边界
- **特殊输入**: 处理 n=0, n=1 等特殊情况

4. 性能优化
- **数学性质利用**: 发现并利用数学规律
- **状态转移优化**: 优化状态转移过程
- **空间优化**: 使用滚动数组等技术优化空间

七、代码实现规范

1. 文件命名规范
- 使用统一的命名格式：Code{编号}_{题目描述}.{语言}
- 编号从 01 开始顺序排列
- 题目描述使用驼峰命名法

2. 代码结构规范
- 包含完整的题目描述和链接
- 详细的解题思路和时间空间复杂度分析
- 完整的测试用例和性能测试
- 清晰的注释和文档

3. 跨语言实现
- 每种题目提供 Java、C++、Python 三种实现
- 保持算法逻辑的一致性
- 适应各语言特性的优化

八、学习路径建议

1. 初级阶段
- 先掌握基础数位 DP 框架
- 练习简单的计数问题
- 理解状态设计和记忆化原理

2. 中级阶段
- 学习复杂的状态设计
- 练习数位和、模运算相关问题
- 掌握状态压缩技巧

3. 高级阶段
- 学习结合字符串匹配的数位 DP
- 练习二进制数位问题
- 掌握性能优化技巧

4. 实战阶段
- 参加各大 OJ 平台的数位 DP 比赛
- 总结各类题型的解题模式
- 分享解题经验和技巧

九、扩展资源

1. 在线评测平台
- LeetCode: 丰富的数位 DP 题目
- Codeforces: 高质量的数位 DP 比赛题
- AtCoder: 日本编程比赛的数位 DP 题目
- 洛谷: 中文社区的优秀题目

2. 学习资料
- 算法竞赛入门经典：数位 DP 章节
- 挑战程序设计竞赛：动态规划部分
- 各大博客的技术文章

3. 社区讨论
- Stack Overflow: 技术问题解答

- GitHub: 开源代码参考
- 各大技术论坛：经验交流

==========

文件: ADDITIONAL_PROBLEMS.md
==========
数位 DP 扩展题目列表

以下是在学习数位 DP 过程中可以练习的更多题目，涵盖各大 OJ 平台。

一、LeetCode 题目

1. 233. 数字 1 的个数
- **题目链接**: https://leetcode.cn/problems/number-of-digit-one/
- **题目描述**: 给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。
- **相关文件**: Code05_CountDigitOne.java, Code05_CountDigitOne.py

2. 600. 不含连续 1 的非负整数
- **题目链接**: https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/
- **题目描述**: 给定一个正整数 n，返回在 [0, n] 范围内不含连续 1 的非负整数的个数。
- **相关文件**: Code06_NonNegativeIntegersWithoutConsecutiveOnes.java, Code06_NonNegativeIntegersWithoutConsecutiveOnes.py

3. 902. 最大为 N 的数字组合
- **题目链接**: https://leetcode.cn/problems/numbers-at-most-n-given-digit-set/
- **题目描述**: 给定一个按 非递减顺序 排列的数字数组 digits 。你可以用任意次数 digits[i] 来写的数字。返回可以生成的小于或等于给定整数 n 的正整数的个数。
- **相关文件**: Code02_NumbersAtMostGivenDigitSet.java

4. 1012. 至少有 1 位重复的数字
- **题目链接**: https://leetcode.cn/problems/numbers-with-repeated-digits/
- **题目描述**: 给定正整数 n，返回在 [1, n] 范围内具有至少 1 位重复数字的正整数的个数。
- **相关文件**: Code04_NumbersWithRepeatedDigits.java

5. 357. 统计各位数字都不同的数字个数
- **题目链接**: https://leetcode.cn/problems/count-numbers-with-unique-digits/
- **题目描述**: 给你一个整数 n，统计并返回各位数字都不同的数字 x 的个数，其中 0 <= x < 10^n。
- **相关文件**: Code01_CountNumbersWithUniqueDigits.java

6. 2719. 统计整数数目
- **题目链接**: https://leetcode.cn/problems/count-of-integers/
- **题目描述**: 给你两个数字字符串 num1 和 num2 ，以及两个整数 max_sum 和 min_sum 。如果一个整数

x 满足以下条件，我们称它是一个好整数：num1 <= x <= num2；min_sum <= digit_sum(x) <= max_sum。请你返回好整数的数目。答案可能很大，请你返回答案对 10^9 + 7 取余后的结果。
- **相关文件**: Code03_CountOfIntegers.java

7. 2827. 范围中美丽整数的数目
- **题目链接**: https://leetcode.cn/problems/number-of-beautiful-integers-in-the-range/
- **题目描述**: 给你两个正整数：low 和 high 。如果一个整数满足以下条件，我们称它为美丽整数：1. 偶数数位的数目与奇数数位的数目相同；2. 这个整数能被 k 整除。请你返回范围 [low, high] 中美丽整数的数目。
- **相关文件**: Code07_NumberOfBeautifulIntegersInTheRange.java, Code07_NumberOfBeautifulIntegersInTheRange.py, Code07_NumberOfBeautifulIntegersInTheRange.cpp

8. 1397. 找到所有好字符串
- **题目链接**: https://leetcode.cn/problems/find-all-good-strings/
- **题目描述**: 给你两个长度为 n 的字符串 s1 和 s2，以及一个字符串 evil。请你返回好字符串的数目。好字符串的定义是：它的长度为 n，字典序大于等于 s1，字典序小于等于 s2，且不包含 evil 为子字符串。
- **相关文件**: Code08_FindAllGoodStrings.java, Code08_FindAllGoodStrings.py, Code08_FindAllGoodStrings.cpp

二、洛谷题目

1. P2602 [ZJOI2010] 数字计数
- **题目链接**: https://www.luogu.com.cn/problem/P2602
- **题目描述**: 给定两个正整数 a 和 b，求在 [a,b] 范围上的所有整数中，每个数码 (digit) 各出现了多少次。
- **相关文件**: Code09_DigitCount.java, Code09_DigitCount.py, Code09_DigitCount.cpp

2. P3414 SAC#1 - 组合数
- **题目链接**: https://www.luogu.com.cn/problem/P3414
- **题目描述**: 求 C(n,0) + C(n,1) + ... + C(n,n) 的值，其中 C(n,m) 表示组合数。

3. P4127 [AHOI2009] 同类分布
- **题目链接**: https://www.luogu.com.cn/problem/P4127
- **题目描述**: 给出两个数 a,b，求出 [a,b] 中各位数字之和能整除原数的数的个数。

4. P4124 [CQOI2016] 手机号码
- **题目链接**: https://www.luogu.com.cn/problem/P4124
- **题目描述**: 给定一个区间 [L,R]，统计这个区间内满足特定条件的手机号码个数。条件包括：不能出现 4；不能出现连续 3 个相同的数字；后四位必须是相同数字；后五位必须是顺子（例如 12345）。

5. P4317 花神的数论题
- **题目链接**: https://www.luogu.com.cn/problem/P4317
- **题目描述**: 定义 sum(i) 表示 i 的二进制表示中 1 的个数。给出一个正整数 N，求 sum(1) 到 sum(N)

的乘积。

三、牛客网题目

1. 数位 dp - 数位小孩
- **题目链接**: https://ac.nowcoder.com/acm/problem/17023
- **题目描述**: 给出一个区间 [l,r]，求这个区间内有多少个数字满足特定条件。

四、Codeforces 题目

1. 55D. Beautiful numbers
- **题目链接**: https://codeforces.com/problemset/problem/55/D
- **题目描述**: 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。给定区间 [l,r]，求其中美丽数字的个数。

2. 628D. Magic Numbers
- **题目链接**: https://codeforces.com/problemset/problem/628/D
- **题目描述**: 定义一个 magic number 为满足特定条件的数字，给定区间 [a,b]，求其中 magic number 的个数。

3. 464C. Substitutes in Number
- **题目链接**: https://codeforces.com/problemset/problem/464/C
- **题目描述**: 给定一个初始数字字符串和一系列替换规则，求经过替换后得到的数字对某个数取模的结果。

五、AtCoder 题目

1. ABC135 D - Digits Parade
- **题目链接**: https://atcoder.jp/contests/abc135/tasks/abc135_d
- **题目描述**: 给定一个由数字和?组成的字符串，?可以替换成 0-9 的任意数字，求有多少种替换方案使得结果能被 13 整除。

2. ABC102 D - Equal Cut
- **题目链接**: https://atcoder.jp/contests/abc102/tasks/arc100_b
- **题目描述**: 给定一个数组，将其分为 4 段，使得 4 段和的最大值与最小值差值最小。

六、HackerRank 题目

1. Digit DP - Special Numbers
- **题目链接**: https://www.hackerrank.com/contests/hourrank-21/challenges
- **题目描述**: 给定一个区间 [l,r]，统计满足特定条件的数字个数。

七、面试高频题

1. 统计区间内特定数字出现次数
- **题目描述**: 给定区间 [l,r] 和数字 d，统计 d 在区间内所有数字中出现的次数。

2. 统计满足数位和条件的数字
- **题目描述**: 给定区间 [l,r] 和数位和范围 [min_sum, max_sum]，统计满足条件的数字个数。

3. 统计不含特定模式的数字
- **题目描述**: 给定区间 [l,r]，统计不含特定数字模式（如连续相同数字、连续递增等）的数字个数。

数位 DP 常见题型总结

1. 基础计数问题
- 统计某个数字在区间内出现次数
- 统计满足特定数位和的数字个数
- 统计不含连续相同数字的数字个数

2. 约束条件问题
- 数字必须由特定集合中的数字组成
- 数字必须满足特定的整除性质
- 数字必须不包含某些子串

3. 最优化问题
- 在满足条件的数字中找最大/最小值
- 在满足条件的数字中找具有特定性质的数字

4. 复合问题
- 结合字符串匹配的数位 DP
- 结合图论的数位 DP
- 结合博弈论的数位 DP

数位 DP 解题技巧

1. 状态设计
- 确定当前位置
- 确定已有的约束条件（如已出现的数字个数、数位和等）
- 确定是否达到上限（isLimit）
- 确定是否已经开始填数字（isNum，处理前导零）

2. 转移方程
- 枚举当前位可以填入的数字
- 根据填入的数字更新状态
- 递归处理下一位

3. 边界条件
- 处理到末尾时的返回值
- 处理前导零的情况
- 处理上限约束的情况

4. 优化技巧
- 记忆化搜索避免重复计算
- 状态压缩减少空间复杂度
- 提前剪枝减少搜索空间

==========

文件: COMPREHENSIVE_GUIDE.md
==========
数位 DP 完全掌握指南

一、算法核心思想

1.1 基本概念
数位 DP（Digit Dynamic Programming）是一种专门用于解决与数字的数位相关问题的动态规划技术。它通过逐位确定数字，结合记忆化搜索来高效统计满足特定条件的数字个数或计算相关属性。

1.2 适用问题特征
- **数字统计问题**：统计区间内满足特定条件的数字个数
- **数字属性计算**：计算数字的某种属性总和或最值
- **数字构造问题**：构造满足特定条件的数字
- **数字模式匹配**：数字需要满足特定的模式或约束

1.3 核心思想
将数字看作字符串，从高位到低位逐位确定，通过状态记录已有的约束信息，利用记忆化避免重复计算。

二、标准模板框架

2.1 基本模板结构
```java
// 数位 DP 标准模板
int dfs(int pos, int state, boolean isLimit, boolean isNum) {
    // 1. 递归终止条件
    if (pos == n) {
        return check(state) ? 1 : 0;
    }
```

```
    // 2. 记忆化搜索
    if (!isLimit && isNum && memo[pos][state] != -1) {
        return memo[pos][state];
    }

    // 3. 确定可选范围
    int up = isLimit ? digits[pos] : 9;
    int ans = 0;

    // 4. 处理前导零
    if (!isNum) {
        ans += dfs(pos + 1, state, false, false);
    }

    // 5. 枚举当前位选择
    for (int d = isNum ? 0 : 1; d <= up; d++) {
        if (valid(d, state)) {
            ans += dfs(pos + 1, updateState(state, d),
                       isLimit && d == up, true);
        }
    }

    // 6. 记忆化存储
    if (!isLimit && isNum) {
        memo[pos][state] = ans;
    }

    return ans;
}
```

2.2 关键参数说明
- **pos**：当前处理到的数位位置
- **state**：记录已有的约束状态信息
- **isLimit**：是否受到给定数字上界的限制
- **isNum**：是否已经开始填数字（用于处理前导零）

三、状态设计技巧

3.1 基础状态设计
问题类型	状态参数	说明
数字计数	计数变量	记录已出现的数字个数

| 数位和 | 和变量 | 记录当前数位和 |
| 数字约束 | 位掩码 | 记录已使用的数字 |

3.2 复合状态设计
对于复杂问题，需要设计多维度状态：
```java
// 复杂状态示例：美丽数字问题
int dfs(int pos, int oddCount, int evenCount, int remainder,
        boolean isLimit, boolean isNum) {
    // 状态包含：位置、奇数位计数、偶数位计数、余数、限制标记、数字标记
}
```

3.3 状态压缩技巧
- **位运算压缩**：使用位掩码压缩多个布尔状态
- **模运算优化**：利用模运算性质减少状态数
- **哈希映射**：对于稀疏状态使用哈希表

四、经典题型详解

4.1 数字统计问题
例题：统计数字 1 的出现次数
```java
// 状态：位置、1 的计数、限制标记
int dfs(int pos, int count, boolean isLimit) {
    if (pos == n) return count;
    // ... 状态转移
}
```

4.2 数位和问题
例题：统计数位和在指定范围内的数字
```java
// 状态：位置、数位和、限制标记
int dfs(int pos, int sum, boolean isLimit) {
    if (pos == n) return (sum >= min && sum <= max) ? 1 : 0;
    // ... 状态转移
}
```

4.3 数字约束问题
例题：统计不含重复数字的数字
```java
```

```
// 状态：位置、数字使用掩码、限制标记、数字标记
int dfs(int pos, int mask, boolean isLimit, boolean isNum) {
    if (pos == n) return isNum ? 1 : 0;
    // ... 状态转移
}
```

4.4 模运算问题
例题：统计能被特定数整除的数字
```java
// 状态：位置、余数、限制标记
int dfs(int pos, int remainder, boolean isLimit) {
    if (pos == n) return remainder == 0 ? 1 : 0;
    // ... 状态转移
}
```

五、优化技巧大全

5.1 记忆化优化
- **缓存键设计**：合理设计记忆化缓存键
- **状态去重**：识别并合并相同状态
- **懒记忆化**：只记忆化不受限制的状态

5.2 数学优化
- **组合数学**：利用排列组合公式替代 DP
- **数论性质**：利用模运算、GCD 等性质
- **递推关系**：发现并利用递推关系

5.3 算法优化
- **提前剪枝**：在不可能满足条件时提前返回
- **状态压缩**：减少状态空间复杂度
- **并行计算**：利用多线程加速计算

六、跨语言实现对比

6.1 Java 实现特点
优势：
- 类型安全，代码结构清晰
- 丰富的标准库支持
- 适合算法竞赛和面试

示例：

```java
public class DigitDP {
    private int n;
    private char[] digits;
    private int[][][] dp;

    public int solve() {
        // Java实现代码
    }
}
```

6.2 C++实现特点
优势：
- 性能优化空间大
- STL容器丰富
- 适合需要极致性能的场景

示例：
```cpp
class DigitDP {
private:
    vector<int> digits;
    vector<vector<vector<int>>> dp;

public:
    int solve() {
        // C++实现代码
    }
};
```

6.3 Python实现特点
优势：
- 代码简洁，开发效率高
- 装饰器实现自动记忆化
- 适合快速原型和学习

示例：
```python
class DigitDP:
    def solve(self):
        @lru_cache(maxsize=None)
```

```
        def dfs(pos, state, is_limit, is_num):
            # Python实现代码
        return dfs(0, 0, True, False)
```

七、工程化实践

7.1 代码规范
- **文件结构**：统一的文件命名和组织结构
- **注释文档**：详细的注释和API文档
- **测试覆盖**：完整的单元测试用例

7.2 性能分析
- **复杂度分析**：准确分析时间和空间复杂度
- **性能测试**：不同规模数据的性能测试
- **优化验证**：验证优化措施的有效性

7.3 调试技巧
- **小范围验证**：手动计算小范围结果验证
- **状态跟踪**：打印中间状态理解算法
- **边界测试**：全面测试边界情况

八、实战训练计划

8.1 初级阶段（1-2周）
目标：掌握基本框架和简单题型
题目：
1. 统计数字1的出现次数
2. 统计不含重复数字的数字
3. 统计数位和在范围内的数字

8.2 中级阶段（2-3周）
目标：掌握复杂状态设计和优化技巧
题目：
1. 美丽数字问题
2. 魔法数字问题
3. 数字模式匹配问题

8.3 高级阶段（3-4周）
目标：掌握综合应用和性能优化
题目：
1. 结合字符串匹配的数位DP
2. 大规模数据的性能优化

3. 实际工程问题应用

九、常见问题解答

9.1 如何设计状态参数？
答：分析题目约束条件，确定需要记录的信息：
- 数字相关：数位和、数字使用情况等
- 位置相关：当前位置、前一位状态等
- 约束相关：模运算余数、模式匹配状态等

9.2 如何处理前导零？
答：使用 isNum 参数区分：
- isNum=false：还未开始填数字，可以跳过当前位
- isNum=true：已开始填数字，需要正常处理

9.3 如何优化状态空间？
答：
1. 状态压缩：使用位运算合并多个布尔状态
2. 数学优化：利用数学性质减少状态数
3. 懒记忆化：只记忆化不受限制的状态

9.4 如何调试数位 DP？
答：
1. 小范围手动验证
2. 打印中间状态和决策路径
3. 对比不同解法的结果
4. 测试边界情况

十、扩展应用领域

10.1 机器学习应用
- **特征工程**：数字特征的提取和统计
- **数据生成**：生成满足特定条件的训练数据
- **模式识别**：数字模式的检测和分类

10.2 密码学应用
- **密码分析**：统计特定模式的密码数量
- **密钥生成**：生成满足密码学要求的数字
- **安全验证**：数字模式的安全强度分析

10.3 数据分析应用
- **数据统计**：大规模数字数据的统计分析
- **模式发现**：数字序列中的模式发现

- **预测建模**：基于数字特征的预测模型

通过系统学习本指南，您将全面掌握数位 DP 算法，具备解决各类数字相关问题的能力，为算法竞赛、技术面试和工程实践打下坚实基础。

==========

文件: EXTENDED_PROBLEMS.md
==========
数位 DP 扩展题目完整列表

以下是在学习数位 DP 过程中可以练习的更多题目，涵盖各大 OJ 平台，包含完整的题目描述、解题思路和代码实现。

一、LeetCode 题目

1. 233. 数字 1 的个数
- **题目链接**: https://leetcode.cn/problems/number-of-digit-one/
- **题目描述**: 给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。
- **相关文件**: Code05_CountDigitOne.java, Code05_CountDigitOne.py
- **解题思路**: 数位 DP，记录已出现的 1 的个数
- **时间复杂度**: O(log n)
- **空间复杂度**: O(log n)

2. 600. 不含连续 1 的非负整数
- **题目链接**: https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/
- **题目描述**: 给定一个正整数 n，返回在 [0, n] 范围内不含连续 1 的非负整数的个数。
- **相关文件**: Code06_NonNegativeIntegersWithoutConsecutiveOnes.java, Code06_NonNegativeIntegersWithoutConsecutiveOnes.py, Code16_NumberWithoutConsecutiveOnes.java, Code16_NumberWithoutConsecutiveOnes.cpp, Code16_NumberWithoutConsecutiveOnes.py
- **解题思路**: 二进制数位 DP，记录前一位是否为 1
- **时间复杂度**: O(log n)
- **空间复杂度**: O(log n)

3. 902. 最大为 N 的数字组合
- **题目链接**: https://leetcode.cn/problems/numbers-at-most-n-given-digit-set/
- **题目描述**: 给定一个按 非递减顺序 排列的数字数组 digits 。你可以用任意次数 digits[i] 来写的数字。返回可以生成的小于或等于给定整数 n 的正整数的个数。
- **相关文件**: Code02_NumbersAtMostGivenDigitSet.java
- **解题思路**: 数位 DP，限制数字只能从给定集合中选择
- **时间复杂度**: O(log n * |digits|)
- **空间复杂度**: O(log n)

4. 1012. 至少有 1 位重复的数字
- **题目链接**: https://leetcode.cn/problems/numbers-with-repeated-digits/
- **题目描述**: 给定正整数 n，返回在 [1, n] 范围内具有至少 1 位重复数字的正整数的个数。
- **相关文件**: Code04_NumbersWithRepeatedDigits.java
- **解题思路**: 数位 DP，用位掩码记录已使用的数字
- **时间复杂度**: O(10 * 2^10 * log n)
- **空间复杂度**: O(2^10 * log n)

5. 357. 统计各位数字都不同的数字个数
- **题目链接**: https://leetcode.cn/problems/count-numbers-with-unique-digits/
- **题目描述**: 给你一个整数 n，统计并返回各位数字都不同的数字 x 的个数，其中 0 <= x < 10^n。
- **相关文件**: Code01_CountNumbersWithUniqueDigits.java, Code01_CountNumbersWithUniqueDigits.cpp, Code01_CountNumbersWithUniqueDigits.py
- **解题思路**: 数位 DP 或数学排列组合
- **时间复杂度**: O(n) 或 O(10 * 2^10 * n)
- **空间复杂度**: O(1) 或 O(2^10 * n)

6. 2719. 统计整数数目
- **题目链接**: https://leetcode.cn/problems/count-of-integers/
- **题目描述**: 给你两个数字字符串 num1 和 num2 ，以及两个整数 max_sum 和 min_sum 。如果一个整数 x 满足以下条件，我们称它是一个好整数：num1 <= x <= num2；min_sum <= digit_sum(x) <= max_sum。请你返回好整数的数目。答案可能很大，请你返回答案对 10^9 + 7 取余后的结果。
- **相关文件**: Code03_CountOfIntegers.java
- **解题思路**: 数位 DP，记录数位和范围
- **时间复杂度**: O(log n * max_sum)
- **空间复杂度**: O(log n * max_sum)

7. 2827. 范围中美丽整数的数目
- **题目链接**: https://leetcode.cn/problems/number-of-beautiful-integers-in-the-range/
- **题目描述**: 给你两个正整数：low 和 high 。如果一个整数满足以下条件，我们称它为美丽整数：1. 偶数数位的数目与奇数数位的数目相同；2. 这个整数能被 k 整除。请你返回范围 [low, high] 中美丽整数的数目。
- **相关文件**: Code07_NumberOfBeautifulIntegersInTheRange.java, Code07_NumberOfBeautifulIntegersInTheRange.py, Code07_NumberOfBeautifulIntegersInTheRange.cpp
- **解题思路**: 数位 DP，记录奇偶数位个数和余数
- **时间复杂度**: O(log n * k * 10^2)
- **空间复杂度**: O(log n * k * 10^2)

8. 1397. 找到所有好字符串
- **题目链接**: https://leetcode.cn/problems/find-all-good-strings/
- **题目描述**: 给你两个长度为 n 的字符串 s1 和 s2，以及一个字符串 evil。请你返回好字符串的数目。好字符串的定义是：它的长度为 n，字典序大于等于 s1，字典序小于等于 s2，且不包含 evil 为子字符串。
- **相关文件**: Code08_FindAllGoodStrings.java, Code08_FindAllGoodStrings.py,

Code08_FindAllGoodStrings.cpp
- **解题思路**: 数位 DP + KMP 自动机
- **时间复杂度**: O(n * |evil| * 26)
- **空间复杂度**: O(n * |evil|)

二、Codeforces 题目

1. 55D. Beautiful Numbers
- **题目链接**: https://codeforces.com/problemset/problem/55/D
- **题目描述**: 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。给定区间 [l, r]，求其中美丽数字的个数。
- **相关文件**: Code13_BeautifulNumbersCF.java, Code13_BeautifulNumbersCF.cpp, Code13_BeautifulNumbersCF.py
- **解题思路**: 数位 DP，利用 LCM 性质优化状态
- **时间复杂度**: O(log r * 2520 * 50)
- **空间复杂度**: O(log r * 2520 * 50)

2. 628D. Magic Numbers
- **题目链接**: https://codeforces.com/problemset/problem/628/D
- **题目描述**: 定义一个 d-magic number 为满足特定条件的数字，给定区间 [a, b]，求其中 magic number 的个数。
- **相关文件**: Code14_MagicNumbersCF.java, Code14_MagicNumbersCF.cpp, Code14_MagicNumbersCF.py
- **解题思路**: 数位 DP，根据位置奇偶性应用不同约束
- **时间复杂度**: O(log b * m)
- **空间复杂度**: O(log b * m)

三、AtCoder 题目

1. ABC135 D - Digits Parade
- **题目链接**: https://atcoder.jp/contests/abc135/tasks/abc135_d
- **题目描述**: 给定一个由数字和?组成的字符串，?可以替换成 0-9 的任意数字，求有多少种替换方案使得结果能被 13 整除。
- **相关文件**: Code15_DigitsParadeABC.java, Code15_DigitsParadeABC.cpp, Code15_DigitsParadeABC.py
- **解题思路**: 数位 DP，处理通配符和模运算
- **时间复杂度**: O(n * 13)
- **空间复杂度**: O(n * 13)

四、洛谷题目

1. P2602 [ZJOI2010] 数字计数
- **题目链接**: https://www.luogu.com.cn/problem/P2602
- **题目描述**: 给定两个正整数 a 和 b，求在 [a, b] 范围上的所有整数中，每个数码 (digit) 各出现了

多少次。
- **相关文件**: Code09_DigitCount.java, Code09_DigitCount.py, Code09_DigitCount.cpp
- **解题思路**: 数位DP，分别统计每个数字的出现次数
- **时间复杂度**: O(log b * 10)
- **空间复杂度**: O(log b * 10)

2. P4127 [AHOI2009] 同类分布
- **题目链接**: https://www.luogu.com.cn/problem/P4127
- **题目描述**: 给出两个数 a,b，求出 [a,b] 中各位数字之和能整除原数的数的个数。
- **解题思路**: 数位DP，记录数位和和余数
- **时间复杂度**: O(log b * 162 * 162)
- **空间复杂度**: O(log b * 162 * 162)

3. P4124 [CQOI2016] 手机号码
- **题目链接**: https://www.luogu.com.cn/problem/P4124
- **题目描述**: 给定一个区间 [L,R]，统计这个区间内满足特定条件的手机号码个数。
- **解题思路**: 数位DP，多个约束条件组合
- **时间复杂度**: O(log R * 2^3 * 10^2)
- **空间复杂度**: O(log R * 2^3 * 10^2)

五、其他平台题目

1. HackerRank - Digit DP
- **题目链接**: https://www.hackerrank.com/challenges/digit-dp
- **题目描述**: 各种数位DP变种题目
- **解题思路**: 标准数位DP框架应用

2. 牛客网 - 数位小孩
- **题目链接**: https://ac.nowcoder.com/acm/problem/17023
- **题目描述**: 给出一个区间 [l,r]，求这个区间内有多少个数字满足特定条件。
- **解题思路**: 基础数位DP练习

六、数位DP题型分类

1. 基础计数问题
- **特征**: 统计满足简单条件的数字个数
- **例题**: 数字1的个数、不含连续1的数字
- **状态设计**: 位置 + 限制标记 + 简单计数

2. 数位和问题
- **特征**: 与数字的数位和相关
- **例题**: 数位和在指定范围内、数位和整除原数
- **状态设计**: 位置 + 限制标记 + 数位和 + 余数

3. 数字约束问题
- **特征**: 数字必须满足特定模式
- **例题**: 美丽数字、魔法数字
- **状态设计**: 位置 + 限制标记 + 模式状态

4. 字符串匹配问题
- **特征**: 结合字符串匹配算法
- **例题**: 不包含特定子串的好字符串
- **状态设计**: 位置 + 限制标记 + KMP 状态

5. 二进制数位问题
- **特征**: 处理二进制表示
- **例题**: 不含连续 1 的二进制数
- **状态设计**: 位置 + 限制标记 + 前一位状态

6. 通配符问题
- **特征**: 包含通配符的数字
- **例题**: Digits Parade
- **状态设计**: 位置 + 限制标记 + 余数

七、解题技巧总结

1. 状态设计原则
- **最小化状态数**: 只记录必要的约束信息
- **状态压缩**: 使用位运算压缩多个布尔状态
- **模运算优化**: 利用模运算性质减少状态

2. 记忆化优化
- **缓存键设计**: 合理设计记忆化缓存键
- **状态去重**: 识别并合并相同状态
- **提前剪枝**: 在不可能满足条件时提前返回

3. 边界处理
- **前导零处理**: 正确处理前导零情况
- **上下界处理**: 正确处理区间边界
- **特殊输入**: 处理 n=0, n=1 等特殊情况

4. 性能优化
- **数学性质利用**: 发现并利用数学规律
- **状态转移优化**: 优化状态转移过程
- **空间优化**: 使用滚动数组等技术优化空间

八、代码实现规范

1. 文件命名规范
- 使用统一的命名格式：Code{编号}_{题目描述}.{语言}
- 编号从 01 开始顺序排列
- 题目描述使用驼峰命名法

2. 代码结构规范
- 包含完整的题目描述和链接
- 详细的解题思路和时间空间复杂度分析
- 完整的测试用例和性能测试
- 清晰的注释和文档

3. 跨语言实现
- 每种题目提供 Java、C++、Python 三种实现
- 保持算法逻辑的一致性
- 适应各语言特性的优化

九、学习路径建议

1. 初级阶段
- 先掌握基础数位 DP 框架
- 练习简单的计数问题
- 理解状态设计和记忆化原理

2. 中级阶段
- 学习复杂的状态设计
- 练习数位和、模运算相关问题
- 掌握状态压缩技巧

3. 高级阶段
- 学习结合字符串匹配的数位 DP
- 练习二进制数位问题
- 掌握性能优化技巧

4. 实战阶段
- 参加各大 OJ 平台的数位 DP 比赛
- 总结各类题型的解题模式
- 分享解题经验和技巧

十、扩展资源

1. 在线评测平台

- LeetCode: 丰富的数位 DP 题目
- Codeforces: 高质量的数位 DP 比赛题
- AtCoder: 日本编程比赛的数位 DP 题目
- 洛谷: 中文社区的优秀题目

2. 学习资料
- 算法竞赛入门经典：数位 DP 章节
- 挑战程序设计竞赛：动态规划部分
- 各大博客的技术文章

3. 社区讨论
- Stack Overflow: 技术问题解答
- GitHub: 开源代码参考
- 各大技术论坛：经验交流

通过系统学习以上题目和技巧，可以全面掌握数位 DP 算法，解决各类与数字数位相关的问题。

==========

文件: SUMMARY.md
==========
数位 DP 算法详解

1. 算法简介

数位 DP（Digit Dynamic Programming）是一种用于解决与数字的数位相关问题的动态规划技术。它通常用于统计某个区间内满足特定条件的数字个数，或者计算这些数字的某种属性总和。

2. 适用场景

数位 DP 适用于以下几类问题：

1. **计数问题**：统计区间[l,r]内满足特定条件的数字个数
2. **求和问题**：计算区间内所有满足条件数字的某种属性总和
3. **最值问题**：找出区间内满足条件的最大/最小数字
4. **构造问题**：构造满足特定条件的数字

常见约束条件包括：
- 数字中不能包含某些数位
- 相邻数位之间有特定关系（如不能相等、不能递增等）
- 数位和需要满足特定条件
- 数字需要满足特定的整除性质
- 数字的二进制表示需要满足特定条件

3. 核心思想

数位 DP 的核心思想是逐位确定数字，通过记忆化搜索避免重复计算。

3.1 基本框架

```java
// 一般形式的数位 DP
int dfs(数位位置, 已有状态, 是否达到上界, 是否开始填数字) {
    // 边界条件
    if (处理完所有数位) {
        return 根据是否已填数字返回相应值;
    }

    // 记忆化
    if (已计算过该状态) {
        return 记忆化的结果;
    }

    int ans = 0;

    // 处理前导零
    if (!已开始填数字) {
        ans += dfs(下一位, 状态, false, false);
    }

    // 确定当前位可选数字的范围
    int up = 是否达到上界 ? 当前位上界 : 9;

    // 枚举当前位可选的数字
    for (int d = 已开始填数字 ? 0 : 1; d <= up; d++) {
        // 根据题目条件判断是否可选
        if (满足条件) {
            ans += dfs(下一位, 更新状态, 是否达到上界 && d == up, true);
        }
    }

    // 记忆化存储
    return ans;
}
```

3.2 关键参数说明

1. **数位位置(pos)**：当前处理到数字的第几位
2. **状态(state)**：记录已有的约束信息，如已使用的数字、数位和等
3. **是否达到上界(isLimit)**：当前位是否受到给定数字上界的限制
4. **是否开始填数字(isNum)**：是否已经开始填数字（用于处理前导零）

4. 算法复杂度

- **时间复杂度**：通常为O(log n * S)，其中n是给定的数字，S是状态数
- **空间复杂度**：通常为O(log n * S)，用于记忆化存储

5. 常见题型及解法

5.1 统计特定数字出现次数

例题：统计区间[1,n]中数字1出现的次数

解法要点：
- 状态只需要记录1出现的次数
- 每次当前位填1时，计数器加1

5.2 数位不重复问题

例题：统计区间内各位数字都不相同的数字个数

解法要点：
- 用位掩码记录已使用的数字
- 当前位不能选择已使用的数字

5.3 约束条件问题

例题：统计不含连续1的二进制数个数

解法要点：
- 记录前一位的数字
- 当前位不能与前一位同时为1

5.4 数位和问题

例题：统计数位和在指定范围内的数字个数

解法要点：

- 状态记录当前数位和
- 注意数位和的边界处理

5.5 奇偶数位计数问题

例题：统计区间内奇数位和偶数位数目相等的数字个数

解法要点：
- 分别记录奇数位和偶数位的数目
- 在边界条件判断奇偶数位是否相等

5.6 字符串匹配问题

例题：统计区间内不包含特定子串的字符串个数

解法要点：
- 结合 KMP 算法避免包含特定子串
- 用状态记录当前已匹配的前缀长度

6. 优化技巧

6.1 记忆化搜索

通过缓存已计算过的状态，避免重复计算，是数位 DP 的核心优化技巧。

6.2 状态压缩

对于状态较少的情况，可以用位运算进行状态压缩，减少空间使用。

6.3 提前剪枝

在搜索过程中，如果已经确定不满足条件，可以提前返回，减少搜索空间。

6.4 循环展开

对于状态固定的简单问题，可以手动展开循环，减少函数调用开销。

6.5 多维度状态设计

对于复杂约束条件，需要设计多维度状态：
- 数位位置
- 上界限制标记
- 前导零标记

- 各种约束条件状态（如奇偶计数、余数等）
- 字符串匹配状态（如 KMP 的匹配长度）

6.6 模运算优化

在涉及整除条件的问题中，使用模运算优化：
- 记录当前数字对特定值的余数
- 在状态转移中更新余数
- 在边界条件判断余数是否为 0

7. 工程化考虑

7.1 异常处理

- 处理输入参数的边界情况（如 n<=0）
- 处理数据范围溢出问题

7.2 可读性优化

- 变量命名清晰，如 pos(位置)、limit(限制)、mask(掩码)
- 添加详细注释，说明每个参数的含义和状态转移过程

7.3 跨语言实现

- Java: 使用多维数组进行记忆化
- Python: 使用字典进行记忆化
- C++: 注意避免使用复杂的 STL 容器，优先使用基本数组

7.3 性能优化

- 对于简单问题，可以考虑数学方法替代 DP
- 根据实际需要选择是否使用记忆化

8. 调试技巧

8.1 打印中间状态

在调试时可以打印搜索过程中的关键状态，帮助理解算法执行过程。

8.2 边界测试

特别注意边界情况的测试，如：
- n=0, n=1 等极值

- 数字全为9的情况
- 满足/不满足条件的边界值

8.3 对拍测试

实现多种解法进行对拍，验证结果正确性。

9. 与其他算法的联系

9.1 与组合数学

很多数位DP问题有对应的组合数学解法，通常更高效但不够通用。

9.2 与字符串匹配

某些复杂的数位DP问题可以转化为字符串匹配问题。

9.3 与自动机

数位DP的状态转移过程可以看作自动机的状态转换。

9.4 与数论

数位DP经常与数论知识结合：
- 整除性质
- 同余关系
- 最大公约数和最小公倍数

10. 补充题目详解

10.1 Codeforces 55D - Beautiful Numbers
题目特点：结合数论和数位DP，利用LCM性质优化状态空间
关键技巧：
- 1-9的LCM是2520，所有数字的LCM都是2520的约数
- 使用位掩码记录已使用的数字
- 模运算优化状态转移

时间复杂度：$O(\log r \times 2520 \times 50)$
空间复杂度：$O(\log r \times 2520 \times 50)$

10.2 Codeforces 628D - Magic Numbers
题目特点：位置相关的约束条件
关键技巧：

- 根据位置奇偶性应用不同约束
- 偶数位置必须等于 d，奇数位置必须不等于 d
- 结合模运算约束

时间复杂度：O(log b × m)
空间复杂度：O(log b × m)

10.3 AtCoder ABC135D - Digits Parade
题目特点：通配符处理和模运算
关键技巧：
- '?'可以替换为 0-9 任意数字
- 动态规划处理模 13 的余数
- 两种实现：迭代 DP 和记忆化 DFS

时间复杂度：O(n × 13)
空间复杂度：O(n × 13)

10.4 LeetCode 600 - 不含连续 1 的非负整数
题目特点：二进制数位 DP
关键技巧：
- 处理二进制表示而非十进制
- 记录前一位是否为 1 来避免连续 1
- 数学方法利用斐波那契数列性质

时间复杂度：O(log n)
空间复杂度：O(log n)

10.5 LeetCode 1012 - 至少有 1 位重复的数字
题目特点：补集思想应用
关键技巧：
- 将问题转化为求没有重复数字的数字个数
- 使用位掩码记录数字使用情况
- 数学排列组合方法更高效

时间复杂度：O(log n × 2^10) 或 O(10^2)
空间复杂度：O(log n × 2^10) 或 O(1)

11. 跨语言实现对比

11.1 Java 实现特点
- 使用多维数组进行记忆化
- 类型安全，代码结构清晰
- 适合算法竞赛和面试准备

11.2 C++实现特点
- 使用 vector 和 bitset 等 STL 容器
- 性能优化空间大
- 适合需要极致性能的场景

11.3 Python 实现特点
- 使用装饰器实现自动记忆化
- 代码简洁，开发效率高
- 适合快速原型和算法学习

12. 工程化实践指南

12.1 代码规范
- 统一的文件命名和代码结构
- 详细的注释和文档说明
- 完整的测试用例覆盖

12.2 性能优化
- 合理设计状态参数减少状态数
- 利用数学性质优化算法
- 选择合适的数据结构

12.3 调试技巧
- 小范围手动验证结果
- 打印中间状态理解算法执行
- 边界情况全面测试

13. 学习建议

1. **掌握基础模板**：熟练掌握数位 DP 的基本框架
2. **练习经典题目**：从简单题开始，逐步增加难度
3. **理解状态设计**：重点理解如何设计状态表示约束条件
4. **扩展应用场景**：尝试将数位 DP 应用到更复杂的问题中
5. **关注优化技巧**：学习各种优化方法，提高算法效率
6. **掌握变种题型**：熟悉各种变种题型，如字符串匹配、奇偶计数等
7. **理解工程应用**：了解数位 DP 在实际工程中的应用场景
8. **跨语言实践**：掌握 Java、C++、Python 三种语言的实现
9. **性能分析**：学会分析算法的时间和空间复杂度
10. **实战训练**：参加各大 OJ 平台的数位 DP 比赛

14. 扩展学习资源

14.1 在线评测平台
- **LeetCode**：丰富的数位DP题目，适合面试准备
- **Codeforces**：高质量的比赛题目，适合算法提升
- **AtCoder**：日本编程比赛的优秀题目
- **洛谷**：中文社区的全面题目资源

14.2 参考书籍
- 《算法竞赛入门经典》- 数位DP章节
- 《挑战程序设计竞赛》- 动态规划部分
- 《算法导论》- 动态规划理论基础

14.3 技术博客和社区
- 各大技术博客的数位DP专题
- GitHub上的开源算法实现
- 技术论坛的经验分享和讨论

通过系统学习以上内容，可以全面掌握数位DP算法，具备解决各类数字数位相关问题的能力。

==========

[代码文件]
==========
文件: Code01_CountNumbersWithUniqueDigits.cpp
==========
```
#include <iostream>
#include <vector>
#include <string>
#include <functional>
#include <memory>
#include <cstring>
#include <chrono>
using namespace std;

using namespace std;

/**
 * 统计各位数字都不同的数字个数
 * 题目来源：LeetCode 357. 统计各位数字都不同的数字个数
 * 题目链接：https://leetcode.cn/problems/count-numbers-with-unique-digits/
 *
 * 题目描述：
 * 给你一个整数 n，代表十进制数字最多有 n 位。如果某个数字，每一位都不同，那么这个数字叫做有效数字。
```

```
 * 返回有效数字的个数，不统计负数范围。
 *
 * 解题思路：
 * 1. 数学方法（排列组合）：按位数分别计算，利用排列组合公式
 * 2. 数位 DP 方法：使用数位 DP 框架，逐位确定数字，用位掩码记录已使用的数字
 *
 * 时间复杂度分析：
 * - 数学方法：O(n)，每个位数计算一次
 * - 数位 DP 方法：O(10 * 2^10 * n)，状态数为位数×状态数×数字选择
 *
 * 空间复杂度分析：
 * - 数学方法：O(1)，只使用常数空间
 * - 数位 DP 方法：O(2^10 * n)，用于记忆化存储
 *
 * 最优解分析：
 * 数学方法是最优解，时间复杂度 O(n)，空间复杂度 O(1)
 * 数位 DP 方法更通用但复杂度较高，适合学习数位 DP 框架
 */

class Solution {
public:
    /**
     * 数学方法（排列组合）- 最优解
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)
     *
     * 算法步骤：
     * 1. n=0 时，只有数字 0，返回 1
     * 2. n=1 时，0-9 共 10 个数字
     * 3. 对于 n>=2 的情况：
     *    - 1 位数：10 个
     *    - 2 位数：9*9 个（第一位 1-9，第二位 0-9 除去第一位）
     *    - 3 位数：9*9*8 个
     *    - 以此类推...
     */
    int countNumbersWithUniqueDigits_math(int n) {
        if (n == 0) return 1;
        if (n == 1) return 10;

        int ans = 10;
        int available = 9;
        int current = 9;
```

```
        for (int i = 2; i <= n; i++) {
            current *= available;
            ans += current;
            available--;
        }

        return ans;
    }

    /**
     * 数位DP方法 - 通用解法
     * 时间复杂度: O(10 * 2^10 * n)
     * 空间复杂度: O(2^10 * n)
     *
     * 状态设计：
     * - pos: 当前处理到第几位
     * - mask: 位掩码，记录已使用的数字（1<<d表示数字d已使用）
     * - isLimit: 是否受到上界限制
     * - isNum: 是否已开始填数字（处理前导零）
     *
     * 记忆化优化：避免重复计算相同状态
     */
    int countNumbersWithUniqueDigits_dp(int n) {
        if (n == 0) return 1;

        // 构造上界字符串（n个9）
        string upper(n, '9');

        // 记忆化数组：dp[pos][mask][isLimit][isNum]
        vector<vector<vector<vector<int>>>> dp(
            n, vector<vector<vector<int>>>(
                1 << 10, vector<vector<int>>(
                    2, vector<int>(2, -1)
                )
            )
        );

        // 使用lambda函数实现DFS
        function<int(int, int, bool, bool)> dfs = [&](int pos, int mask, bool isLimit, bool
isNum) -> int {
            // 递归终止条件：处理完所有数位
            if (pos == n) {
                return 1; // 无论是否已填数字，都返回1（表示数字0或已填数字）
```

```
            }

            // 记忆化搜索：如果已计算过且不受限制且已开始填数字
            if (!isLimit && isNum && dp[pos][mask][0][0] != -1) {
                return dp[pos][mask][0][0];
            }

            int ans = 0;

            // 处理前导零：可以选择跳过当前位
            if (!isNum) {
                ans += dfs(pos + 1, mask, false, false);
            }

            // 确定当前位可选数字范围
            int up = isLimit ? upper[pos] - '0' : 9;
            int start = isNum ? 0 : 1; // 处理前导零

            // 枚举当前位可选的数字
            for (int d = start; d <= up; d++) {
                // 检查数字 d 是否已被使用
                if ((mask & (1 << d)) == 0) {
                    // 递归处理下一位，更新状态
                    ans += dfs(pos + 1, mask | (1 << d),
                               isLimit && (d == up), true);
                }
            }

            // 记忆化存储：只存储不受限制且已开始填数字的状态
            if (!isLimit && isNum) {
                dp[pos][mask][0][0] = ans;
            }

            return ans;
        };

        return dfs(0, 0, true, false);
    }
};

/**
 * 单元测试函数
 * 测试用例设计原则：
```

```
 * 1. 边界测试：n=0, n=1 等边界情况
 * 2. 常规测试：n=2, n=3 等正常情况
 * 3. 对拍测试：两种方法结果应该一致
 */
void testSolution() {
    Solution sol;

    cout << "=== 测试统计各位数字都不同的数字个数 ===" << endl;

    // 测试用例 1: n=0
    cout << "n=0: " << endl;
    cout << "数学方法: " << sol.countNumbersWithUniqueDigits_math(0) << endl;
    cout << "数位 DP 方法: " << sol.countNumbersWithUniqueDigits_dp(0) << endl;
    cout << "预期结果: 1" << endl << endl;

    // 测试用例 2: n=1
    cout << "n=1: " << endl;
    cout << "数学方法: " << sol.countNumbersWithUniqueDigits_math(1) << endl;
    cout << "数位 DP 方法: " << sol.countNumbersWithUniqueDigits_dp(1) << endl;
    cout << "预期结果: 10" << endl << endl;

    // 测试用例 3: n=2
    cout << "n=2: " << endl;
    cout << "数学方法: " << sol.countNumbersWithUniqueDigits_math(2) << endl;
    cout << "数位 DP 方法: " << sol.countNumbersWithUniqueDigits_dp(2) << endl;
    cout << "预期结果: 91" << endl << endl;

    // 测试用例 4: n=3
    cout << "n=3: " << endl;
    int math3 = sol.countNumbersWithUniqueDigits_math(3);
    int dp3 = sol.countNumbersWithUniqueDigits_dp(3);
    cout << "数学方法: " << math3 << endl;
    cout << "数位 DP 方法: " << dp3 << endl;
    cout << "预期结果: 739" << endl;
    cout << "两种方法结果一致: " << (math3 == dp3 ? "是" : "否") << endl << endl;
}

/**
 * 性能测试函数
 * 测试两种方法在不同规模下的性能表现
 */
void performanceTest() {
    Solution sol;
```

```
    cout << "=== 性能测试 ===" << endl;

    for (int n = 1; n <= 8; n++) {
        auto start = chrono::high_resolution_clock::now();
        int result_math = sol.countNumbersWithUniqueDigits_math(n);
        auto end_math = chrono::high_resolution_clock::now();

        auto start_dp = chrono::high_resolution_clock::now();
        int result_dp = sol.countNumbersWithUniqueDigits_dp(n);
        auto end_dp = chrono::high_resolution_clock::now();

        auto duration_math = chrono::duration_cast<chrono::microseconds>(end_math - start);
        auto duration_dp = chrono::duration_cast<chrono::microseconds>(end_dp - start_dp);

        cout << "n=" << n << ": " << endl;
        cout << "数学方法时间: " << duration_math.count() << "微秒" << endl;
        cout << "数位 DP 方法时间: " << duration_dp.count() << "微秒" << endl;
        cout << "结果一致: " << (result_math == result_dp ? "是" : "否") << endl;
        cout << "加速比: " << (double)duration_dp.count() / duration_math.count() << "倍" <<
endl;
        cout << endl;
    }
}

/**
 * 工程化考量：
 * 1. 异常处理：处理 n<0 的情况
 * 2. 边界情况：n=0, n=1 的特殊处理
 * 3. 性能优化：数学方法是最优选择
 * 4. 可读性：清晰的变量命名和注释
 * 5. 测试覆盖：全面的测试用例
 *
 * 跨语言实现差异：
 * - C++: 使用 vector 和 lambda 函数，注意内存管理
 * - Java: 使用多维数组，注意数组初始化
 * - Python: 使用字典进行记忆化，语法更简洁
 */

int main() {
    // 运行功能测试
    testSolution();
```

```
    // 运行性能测试（n 较大时数位 DP 方法较慢，只测试到 n=8）
    performanceTest();

    return 0;
}
```

==========

文件: Code01_CountNumbersWithUniqueDigits.java

==========

```
package class084;

// 统计各位数字都不同的数字个数
// 给你一个整数 n，代表十进制数字最多有 n 位
// 如果某个数字，每一位都不同，那么这个数字叫做有效数字
// 返回有效数字的个数，不统计负数范围
// 测试链接 : https://leetcode.cn/problems/count-numbers-with-unique-digits/
public class Code01_CountNumbersWithUniqueDigits {

    /**
     * 数学方法（排列组合）
     * 时间复杂度: O(n)
     * 空间复杂度: O(1)
     *
     * 解题思路:
     * 1. 对于范围[0, 10^n)，我们可以按位数分别计算：
     *    - 0 位数：0 (1 个)
     *    - 1 位数：1-9 共 9 个
     *    - 2 位数：第一位有 9 种选择(1-9)，第二位有 9 种选择(0-9 中除去第一位) = 9*9=81 个
     *    - 3 位数：第一位有 9 种选择，第二位有 9 种选择，第三位有 8 种选择 = 9*9*8 个
     *    - 以此类推...
     * 2. 将所有位数的结果累加
     *
     * 优点：时间复杂度低，代码简洁
     * 缺点：不够通用，难以扩展到其他约束条件
     */
    public static int countNumbersWithUniqueDigits(int n) {
        if (n == 0) {
            return 1;
        }
        int ans = 10; // 0 位数(0) + 1 位数(1-9) = 10 个
        int permutations = 9; // 2 位数的第一位选择(1-9)
```

```
    // 计算2位数到n位数的排列数
    for (int i = 2; i <= n; i++) {
        permutations *= (11 - i); // 第i位有(11-i)种选择
        ans += permutations;
    }
    return ans;
}

/**
 * 数位DP方法（更通用的解法）
 * 时间复杂度: O(10 * 2^10 * n)
 * 空间复杂度: O(2^10 * n)
 *
 * 解题思路:
 * 1. 使用数位DP框架，逐位确定数字
 * 2. 用位掩码记录已使用的数字
 * 3. 通过记忆化避免重复计算
 *
 * 优点：方法通用，容易扩展到其他约束条件
 * 缺点：时间和空间复杂度较高
 */
public static int countNumbersWithUniqueDigitsDP(int n) {
    if (n == 0) {
        return 1;
    }

    // 构造上界字符串 (10^n - 1)
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
        sb.append("9");
    }
    String upper = sb.toString();

    // dp[pos][mask][isLimit][isNum]
    // pos: 当前处理到第几位
    // mask: 已使用数字的位掩码
    // isLimit: 是否受到上界限制
    // isNum: 是否已开始填数字
    int[][][][] dp = new int[upper.length()][1 << 10][2][2];
    for (int i = 0; i < upper.length(); i++) {
        for (int j = 0; j < (1 << 10); j++) {
            dp[i][j][0][0] = -1;
            dp[i][j][0][1] = -1;
```

```
                dp[i][j][1][0] = -1;
                dp[i][j][1][1] = -1;
            }
        }

        return dfs(upper.toCharArray(), 0, 0, true, false, dp);
    }

    private static int dfs(char[] s, int pos, int mask, boolean isLimit, boolean isNum,
int[][][][] dp) {
        // 递归终止条件
        if (pos == s.length) {
            return isNum ? 1 : 1;  // 如果已经填了数字，返回 1 个有效数字；如果没有填数字，也返回 1
（表示数字 0）
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][mask][0][0] != -1) {
            return dp[pos][mask][0][0];
        }

        int ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
            ans += dfs(s, pos + 1, mask, false, false, dp);
        }

        // 确定当前位可以填入的数字范围
        int up = isLimit ? s[pos] - '0' : 9;

        // 枚举当前位可以填入的数字
        for (int d = isNum ? 0 : 1; d <= up; d++) {
            // 如果数字 d 还没有被使用过
            if ((mask & (1 << d)) == 0) {
                // 递归处理下一位
                ans += dfs(s, pos + 1, mask | (1 << d), isLimit && d == up, true, dp);
            }
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][mask][0][0] = ans;
```

```
        }

        return ans;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int n1 = 2;
        System.out.println("n = " + n1);
        System.out.println("数学方法结果: " + countNumbersWithUniqueDigits(n1));
        System.out.println("数位 DP 方法结果: " + countNumbersWithUniqueDigitsDP(n1));
        // 预期输出: 91 (0-99 中除 11,22,...,99 外的所有数字)

        // 测试用例 2
        int n2 = 0;
        System.out.println("n = " + n2);
        System.out.println("数学方法结果: " + countNumbersWithUniqueDigits(n2));
        System.out.println("数位 DP 方法结果: " + countNumbersWithUniqueDigitsDP(n2));
        // 预期输出: 1 (只有数字 0)
    }
}

==========================================

文件: Code01_CountNumbersWithUniqueDigits.py
==========================================
"""
统计各位数字都不同的数字个数
题目来源：LeetCode 357. 统计各位数字都不同的数字个数
题目链接：https://leetcode.cn/problems/count-numbers-with-unique-digits/

题目描述：
给你一个整数 n，代表十进制数字最多有 n 位。如果某个数字，每一位都不同，那么这个数字叫做有效数字。
返回有效数字的个数，不统计负数范围。

解题思路：
1. 数学方法（排列组合）：按位数分别计算，利用排列组合公式
2. 数位 DP 方法：使用数位 DP 框架，逐位确定数字，用位掩码记录已使用的数字

时间复杂度分析：
- 数学方法：O(n)，每个位数计算一次
- 数位 DP 方法：O(10 * 2^10 * n)，状态数为位数×状态数×数字选择
```

空间复杂度分析：
- 数学方法：O(1)，只使用常数空间
- 数位 DP 方法：O(2^10 * n)，用于记忆化存储

最优解分析：
数学方法是最优解，时间复杂度 O(n)，空间复杂度 O(1)
数位 DP 方法更通用但复杂度较高，适合学习数位 DP 框架
"""

```
import sys
from typing import List
from functools import lru_cache

class Solution:
    """
    数学方法（排列组合）- 最优解
    时间复杂度: O(n)
    空间复杂度: O(1)

    算法步骤：
    1. n=0 时，只有数字 0，返回 1
    2. n=1 时，0-9 共 10 个数字
    3. 对于 n>=2 的情况：
        - 1 位数：10 个
        - 2 位数：9*9 个（第一位 1-9，第二位 0-9 除去第一位）
        - 3 位数：9*9*8 个
        - 以此类推...
    """
    def countNumbersWithUniqueDigits_math(self, n: int) -> int:
        if n == 0:
            return 1
        if n == 1:
            return 10

        ans = 10
        available = 9
        current = 9

        for i in range(2, n + 1):
            current *= available
            ans += current
            available -= 1
```

```
    return ans

"""
数位 DP 方法 - 通用解法
时间复杂度: O(10 * 2^10 * n)
空间复杂度: O(2^10 * n)

状态设计：
- pos: 当前处理到第几位
- mask: 位掩码，记录已使用的数字（1<<d 表示数字 d 已使用）
- isLimit: 是否受到上界限制
- isNum: 是否已开始填数字（处理前导零）

记忆化优化：使用 lru_cache 避免重复计算相同状态
Python 特色：使用装饰器实现记忆化，代码更简洁
"""
def countNumbersWithUniqueDigits_dp(self, n: int) -> int:
    if n == 0:
        return 1

    # 构造上界字符串（n 个 9）
    upper = '9' * n

    @lru_cache(maxsize=None)
    def dfs(pos: int, mask: int, is_limit: bool, is_num: bool) -> int:
        """
        数位 DP 递归函数

        Args:
            pos: 当前处理的位置
            mask: 已使用数字的位掩码
            is_limit: 是否受到上界限制
            is_num: 是否已开始填数字

        Returns:
            满足条件的数字个数
        """
        # 递归终止条件：处理完所有数位
        if pos == len(upper):
            return 1  # 无论是否已填数字，都返回 1（表示数字 0 或已填数字）

        ans = 0
```

```
            # 处理前导零：可以选择跳过当前位
            if not is_num:
                ans += dfs(pos + 1, mask, False, False)

            # 确定当前位可选数字范围
            up = int(upper[pos]) if is_limit else 9
            start = 1 if not is_num else 0  # 处理前导零

            # 枚举当前位可选的数字
            for d in range(start, up + 1):
                # 检查数字 d 是否已被使用
                if mask & (1 << d) == 0:
                    # 递归处理下一位，更新状态
                    ans += dfs(pos + 1, mask | (1 << d),
                               is_limit and d == up, True)

            return ans

        # 从第 0 位开始，初始状态：受限制、未开始填数字
        return dfs(0, 0, True, False)

def test_solution():
    """
    单元测试函数
    测试用例设计原则：
    1. 边界测试：n=0, n=1 等边界情况
    2. 常规测试：n=2, n=3 等正常情况
    3. 对拍测试：两种方法结果应该一致
    """
    sol = Solution()

    print("=== 测试统计各位数字都不同的数字个数 ===")

    # 测试用例 1: n=0
    print("n=0:")
    result_math_0 = sol.countNumbersWithUniqueDigits_math(0)
    result_dp_0 = sol.countNumbersWithUniqueDigits_dp(0)
    print(f"数学方法: {result_math_0}")
    print(f"数位 DP 方法: {result_dp_0}")
    print("预期结果: 1")
    print(f"结果一致: {result_math_0 == result_dp_0}")
    print()
```

```
    # 测试用例 2: n=1
    print("n=1:")
    result_math_1 = sol.countNumbersWithUniqueDigits_math(1)
    result_dp_1 = sol.countNumbersWithUniqueDigits_dp(1)
    print(f"数学方法: {result_math_1}")
    print(f"数位 DP 方法: {result_dp_1}")
    print("预期结果: 10")
    print(f"结果一致: {result_math_1 == result_dp_1}")
    print()

    # 测试用例 3: n=2
    print("n=2:")
    result_math_2 = sol.countNumbersWithUniqueDigits_math(2)
    result_dp_2 = sol.countNumbersWithUniqueDigits_dp(2)
    print(f"数学方法: {result_math_2}")
    print(f"数位 DP 方法: {result_dp_2}")
    print("预期结果: 91")
    print(f"结果一致: {result_math_2 == result_dp_2}")
    print()

    # 测试用例 4: n=3
    print("n=3:")
    result_math_3 = sol.countNumbersWithUniqueDigits_math(3)
    result_dp_3 = sol.countNumbersWithUniqueDigits_dp(3)
    print(f"数学方法: {result_math_3}")
    print(f"数位 DP 方法: {result_dp_3}")
    print("预期结果: 739")
    print(f"两种方法结果一致: {result_math_3 == result_dp_3}")
    print()

def performance_test():
    """
    性能测试函数
    测试两种方法在不同规模下的性能表现
    Python 特色：使用 time 模块进行性能测试
    """
    import time
    sol = Solution()

    print("=== 性能测试 ===")

    for n in range(1, 9):
```

```
        # 测试数学方法
        start_math = time.time()
        result_math = sol.countNumbersWithUniqueDigits_math(n)
        end_math = time.time()

        # 测试数位 DP 方法
        start_dp = time.time()
        result_dp = sol.countNumbersWithUniqueDigits_dp(n)
        end_dp = time.time()

        time_math = (end_math - start_math) * 1000  # 转换为毫秒
        time_dp = (end_dp - start_dp) * 1000  # 转换为毫秒

        print(f"n={n}:")
        print(f"数学方法时间: {time_math:.3f}毫秒")
        print(f"数位 DP 方法时间: {time_dp:.3f}毫秒")
        print(f"结果一致: {result_math == result_dp}")
        if time_math > 0:
            print(f"加速比: {time_dp/time_math:.2f}倍")
        print()

def debug_dp(n: int):
    """
    调试函数：打印数位 DP 的中间状态
    用于理解算法执行过程和调试问题
    """
    print(f"=== 调试数位 DP 过程 n={n} ===")

    if n == 0:
        print("直接返回 1")
        return

    upper = '9' * n
    call_count = 0

    @lru_cache(maxsize=None)
    def dfs_debug(pos, mask, is_limit, is_num, depth=0):
        nonlocal call_count
        call_count += 1

        indent = "  " * depth
        print(f"{indent}dfs(pos={pos}, mask={bin(mask)}, is_limit={is_limit}, is_num={is_num})")
```

```
        if pos == len(upper):
            result = 1 if is_num else 0
            print(f"{indent}-> 终止，返回{result}")
            return result

        ans = 0

        if not is_num:
            print(f"{indent}跳过当前位（前导零）")
            ans += dfs_debug(pos + 1, mask, False, False, depth + 1)

        up = int(upper[pos]) if is_limit else 9
        start = 0 if is_num else 1

        for d in range(start, up + 1):
            if mask & (1 << d) == 0:
                print(f"{indent}选择数字{d}")
                ans += dfs_debug(pos + 1, mask | (1 << d),
                                 is_limit and d == up, True, depth + 1)

        print(f"{indent}-> 返回{ans}")
        return ans

    result = dfs_debug(0, 0, True, False)
    print(f"总调用次数: {call_count}")
    print(f"最终结果: {result}")
    print()

"""
工程化考量：
1. 异常处理：处理 n<0 的情况，返回 0 或抛出异常
2. 边界情况：n=0, n=1 的特殊处理
3. 性能优化：数学方法是最优选择，数位 DP 用于学习
4. 可读性：清晰的变量命名、类型注解和文档字符串
5. 测试覆盖：全面的测试用例和调试功能

Python 语言特性利用：
- 类型注解：提高代码可读性和 IDE 支持
- 装饰器：使用@lru_cache 实现记忆化，代码简洁
- 动态类型：灵活处理各种边界情况
- 内置函数：使用 range、bin 等简化代码

跨语言对比：
```

- Python: 代码简洁，使用装饰器实现记忆化
- Java: 使用多维数组，需要手动管理记忆化
- C++: 使用 vector 和 lambda，注意内存管理
"""

```
if __name__ == "__main__":
    # 运行功能测试
    test_solution()

    # 运行性能测试
    performance_test()

    # 调试模式（可选）
    if len(sys.argv) > 1 and sys.argv[1] == "debug":
        debug_dp(2)
```

==========

文件: Code02_NumbersAtMostGivenDigitSet.java
==========

```
package class084;

// 最大为 N 的数字组合
// 给定一个按 非递减顺序 排列的数字数组 digits
// 已知 digits 一定不包含'0'，可能包含'1' ~ '9'，且无重复字符
// 你可以用任意次数 digits[i] 来写的数字
// 例如，如果 digits = ['1','3','5']
// 我们可以写数字，如 '13', '551', 和 '1351315'
// 返回 可以生成的小于或等于给定整数 n 的正整数的个数
// 测试链接 : https://leetcode.cn/problems/numbers-at-most-n-given-digit-set/
public class Code02_NumbersAtMostGivenDigitSet {

    /**
     * 方法一：递归+记忆化搜索
     * 时间复杂度: O(log n * 2 * 2 * |digits|)
     * 空间复杂度: O(log n * 2 * 2)
     *
     * 解题思路:
     * 1. 将问题转化为数位 DP 问题
     * 2. 逐位确定数字，确保生成的数字不超过 n
     * 3. 使用记忆化搜索避免重复计算
     */
    public static int atMostNGivenDigitSet1(String[] strs, int num) {
```

```
    int tmp = num / 10;
    int len = 1;
    int offset = 1;
    while (tmp > 0) {
        tmp /= 10;
        len++;
        offset *= 10;
    }
    int m = strs.length;
    int[] digits = new int[m];
    for (int i = 0; i < m; i++) {
        digits[i] = Integer.valueOf(strs[i]);
    }
    return f1(digits, num, offset, len, 0, 0);
}

// offset 是辅助变量，完全由 len 决定，只是为了方便提取 num 中某一位数字，不是关键变量
// 还剩下 len 位没有决定
// 如果之前的位已经确定比 num 小，那么 free == 1，表示接下的数字可以自由选择
// 如果之前的位和 num 一样，那么 free == 0，表示接下的数字不能大于 num 当前位的数字
// 如果之前的位没有使用过数字，fix == 0
// 如果之前的位已经使用过数字，fix == 1
// 返回最终<=num 的可能性有多少种
public static int f1(int[] digits, int num, int offset, int len, int free, int fix) {
    if (len == 0) {
        return fix == 1 ? 1 : 0;
    }
    int ans = 0;
    // num 在当前位的数字
    int cur = (num / offset) % 10;
    if (fix == 0) {
        // 之前从来没有选择过数字
        // 当前依然可以不要任何数字，累加后续的可能性
        ans += f1(digits, num, offset / 10, len - 1, 1, 0);
    }
    if (free == 0) {
        // 不能自由选择的情况
        for (int i : digits) {
            if (i < cur) {
                ans += f1(digits, num, offset / 10, len - 1, 1, 1);
            } else if (i == cur) {
                ans += f1(digits, num, offset / 10, len - 1, 0, 1);
            } else {
```

```
                // i > cur
                break;
            }
        }
    } else {
        // 可以自由选择的情况
        ans += digits.length * f1(digits, num, offset / 10, len - 1, 1, 1);
    }
    return ans;
}

/**
 * 方法二：优化的数位 DP 解法
 * 时间复杂度: O(log n * |digits|)
 * 空间复杂度: O(log n)
 *
 * 解题思路:
 * 1. 分两部分计算结果：
 *    - 位数小于 n 的数字个数（可以直接计算）
 *    - 位数等于 n 的数字个数（使用数位 DP）
 * 2. 使用预处理优化重复计算
 */
public static int atMostNGivenDigitSet2(String[] strs, int num) {
    int m = strs.length;
    int[] digits = new int[m];
    for (int i = 0; i < m; i++) {
        digits[i] = Integer.valueOf(strs[i]);
    }
    int len = 1;
    int offset = 1;
    int tmp = num / 10;
    while (tmp > 0) {
        tmp /= 10;
        len++;
        offset *= 10;
    }
    // cnt[i] : 已知前缀比 num 小，剩下 i 位没有确定，请问前缀确定的情况下，一共有多少种数字排列
    // cnt[0] = 1，表示后续已经没有了，前缀的状况都已确定，那么就是 1 种
    // cnt[1] = m
    // cnt[2] = m * m
    // cnt[3] = m * m * m
    // ...
    int[] cnt = new int[len];
```

```
    cnt[0] = 1;
    int ans = 0;
    for (int i = m, k = 1; k < len; k++, i *= m) {
        cnt[k] = i;
        ans += i;
    }
    return ans + f2(digits, cnt, num, offset, len);
}

// offset 是辅助变量，由 len 确定，方便提取 num 中某一位数字
// 还剩下 len 位没有决定，之前的位和 num 一样
// 返回最终<=num 的可能性有多少种
public static int f2(int[] digits, int[] cnt, int num, int offset, int len) {
    if (len == 0) {
        // num 自己
        return 1;
    }
    // cur 是 num 当前位的数字
    int cur = (num / offset) % 10;
    int ans = 0;
    for (int i : digits) {
        if (i < cur) {
            ans += cnt[len - 1];
        } else if (i == cur) {
            ans += f2(digits, cnt, num, offset / 10, len - 1);
        } else {
            break;
        }
    }
    return ans;
}

/**
 * 方法三：标准数位 DP 解法
 * 时间复杂度: O(log n * 2 * 2 * |digits|)
 * 空间复杂度: O(log n * 2 * 2)
 *
 * 解题思路:
 * 1. 使用标准数位 DP 框架
 * 2. 用 isLimit 处理上界约束
 * 3. 用 isNum 处理前导零
 */
public static int atMostNGivenDigitSet3(String[] digits, int n) {
```

```
        char[] s = String.valueOf(n).toCharArray();
        int m = digits.length;
        int[] digitValues = new int[m];
        for (int i = 0; i < m; i++) {
            digitValues[i] = digits[i].charAt(0) - '0';
        }

        // dp[i][isLimit][isNum] 表示处理到第 i 位，是否受限制，是否已开始填数字时的方案数
        int[][][] dp = new int[s.length][2][2];
        for (int i = 0; i < s.length; i++) {
            dp[i][0][0] = -1;
            dp[i][0][1] = -1;
            dp[i][1][0] = -1;
            dp[i][1][1] = -1;
        }

        return dfs(s, digitValues, 0, true, false, dp);
    }

    private static int dfs(char[] s, int[] digits, int pos, boolean isLimit, boolean isNum,
int[][][] dp) {
        // 边界条件
        if (pos == s.length) {
            return isNum ? 1 : 0;
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][0][0] != -1) {
            return dp[pos][0][0];
        }

        int ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
            ans += dfs(s, digits, pos + 1, false, false, dp);
        }

        // 确定当前位可以填入的数字范围
        int up = isLimit ? s[pos] - '0' : 9;

        // 枚举当前位可以填入的数字（必须来自给定的 digits 数组）
        for (int d : digits) {
```

```
            if (d <= up) {
                // 递归处理下一位
                ans += dfs(s, digits, pos + 1, isLimit && d == up, true, dp);
            } else {
                // digits 是排序的，后面的数字更大，可以提前退出
                break;
            }
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][0][0] = ans;
        }

        return ans;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        String[] digits1 = {"1", "3", "5", "7"};
        int n1 = 100;
        System.out.println("digits = [\"1\",\"3\",\"5\",\"7\"], n = " + n1);
        System.out.println("方法一结果: " + atMostNGivenDigitSet1(digits1, n1));
        System.out.println("方法二结果: " + atMostNGivenDigitSet2(digits1, n1));
        System.out.println("方法三结果: " + atMostNGivenDigitSet3(digits1, n1));
        // 预期输出: 20

        // 测试用例 2
        String[] digits2 = {"1", "4", "9"};
        int n2 = 1000000000;
        System.out.println("digits = [\"1\",\"4\",\"9\"], n = " + n2);
        System.out.println("方法二结果: " + atMostNGivenDigitSet2(digits2, n2));
        System.out.println("方法三结果: " + atMostNGivenDigitSet3(digits2, n2));
        // 预期输出: 29523

        // 测试用例 3
        String[] digits3 = {"7"};
        int n3 = 8;
        System.out.println("digits = [\"7\"], n = " + n3);
        System.out.println("方法一结果: " + atMostNGivenDigitSet1(digits3, n3));
        System.out.println("方法二结果: " + atMostNGivenDigitSet2(digits3, n3));
        System.out.println("方法三结果: " + atMostNGivenDigitSet3(digits3, n3));
```

```
        // 预期输出: 1
    }
}
```

==========

文件: Code03_CountOfIntegers.java
==========

```
package class084;

// 统计整数数目
// 给你两个数字字符串 num1 和 num2 ，以及两个整数 max_sum 和 min_sum
// 如果一个整数 x 满足以下条件，我们称它是一个好整数
// num1 <= x <= num2
// min_sum <= digit_sum(x) <= max_sum
// 请你返回好整数的数目
// 答案可能很大，答案对 1000000007 取模
// 注意，digit_sum(x)表示 x 各位数字之和
// 测试链接 : https://leetcode.cn/problems/count-of-integers/
public class Code03_CountOfIntegers {

    public static int MOD = 1000000007;

    public static int MAXN = 23;

    public static int MAXM = 401;

    public static int[][][] dp = new int[MAXN][MAXM][2];

    public static void build() {
        for (int i = 0; i < len; i++) {
            for (int j = 0; j <= max; j++) {
                dp[i][j][0] = -1;
                dp[i][j][1] = -1;
            }
        }
    }

    public static char[] num;

    public static int min, max, len;

    public static int count(String num1, String num2, int min_sum, int max_sum) {
```

```
    min = min_sum;
    max = max_sum;
    num = num2.toCharArray();
    len = num2.length();
    build();
    int ans = f(0, 0, 0);
    num = num1.toCharArray();
    len = num1.length();
    build();
    ans = (ans - f(0, 0, 0) + MOD) % MOD;
    if (check()) {
        ans = (ans + 1) % MOD;
    }
    return ans;
}

// 注意：
// 数字，char[] num
// 数字长度，int len
// 累加和最小要求，int min
// 累加和最大要求，int max
// 这四个变量都是全局静态变量，所以不用带参数，直接访问即可
// 递归含义：
// 从 num 的高位出发，当前来到 i 位上
// 之前决定的数字累加和是 sum
// 之前的决定已经比 num 小，后续可以自由选择数字，那么 free == 1
// 之前的决定和 num 一样，后续不可以自由选择数字，那么 free == 0
// 返回有多少种可能性
public static int f(int i, int sum, int free) {
    if (sum > max) {
        return 0;
    }
    if (sum + (len - i) * 9 < min) {
        return 0;
    }
    if (i == len) {
        return 1;
    }
    if (dp[i][sum][free] != -1) {
        return dp[i][sum][free];
    }
    // cur : num 当前位的数字
    int cur = num[i] - '0';
```

```
        int ans = 0;
        if (free == 0) {
            // 还不能自由选择
            for (int pick = 0; pick < cur; pick++) {
                ans = (ans + f(i + 1, sum + pick, 1)) % MOD;
            }
            ans = (ans + f(i + 1, sum + cur, 0)) % MOD;
        } else {
            // 可以自由选择
            for (int pick = 0; pick <= 9; pick++) {
                ans = (ans + f(i + 1, sum + pick, 1)) % MOD;
            }
        }
        dp[i][sum][free] = ans;
        return ans;
    }

    public static boolean check() {
        int sum = 0;
        for (char cha : num) {
            sum += cha - '0';
        }
        return sum >= min && sum <= max;
    }

}
```

==========

文件: Code04_CountSpecialIntegers.java

==========

```
package class084;

// 完全没有重复的数字个数
// 给定正整数 n，返回在[1, n]范围内每一位都互不相同的正整数个数
// 测试链接 : https://leetcode.cn/problems/count-special-integers/
public class Code04_CountSpecialIntegers {

    /**
     * 数位 DP 解法
     * 时间复杂度: O(log n * 2^10)
     * 空间复杂度: O(log n * 2^10)
     *
```

```
     * 解题思路:
     * 1. 使用数位 DP 框架
     * 2. 用位掩码记录已使用的数字
     * 3. 当前位不能选择已使用的数字
     * 4. 通过记忆化搜索避免重复计算
     */
    public static int countSpecialNumbers(int n) {
        int len = 1;
        int offset = 1;
        int tmp = n / 10;
        while (tmp > 0) {
            len++;
            offset *= 10;
            tmp /= 10;
        }
        // cnt[i] :
        // 一共长度为 len，还剩 i 位没有确定，确定的前缀为 len-i 位，且确定的前缀不为空
        // 0~9 一共 10 个数字，没有选择的数字剩下 10-(len-i)个
        // 那么在后续的 i 位上，有多少种排列
        // 比如：len = 4
        // cnt[4]不计算
        // cnt[3] = 9 * 8 * 7
        // cnt[2] = 8 * 7
        // cnt[1] = 7
        // cnt[0] = 1，表示前缀已确定，后续也没有了，那么就是 1 种排列，就是前缀的状况
        // 再比如：len = 6
        // cnt[6]不计算
        // cnt[5] = 9 * 8 * 7 * 6 * 5
        // cnt[4] = 8 * 7 * 6 * 5
        // cnt[3] = 7 * 6 * 5
        // cnt[2] = 6 * 5
        // cnt[1] = 5
        // cnt[0] = 1，表示前缀已确定，后续也没有了，那么就是 1 种排列，就是前缀的状况
        // 下面 for 循环就是求解 cnt 的代码
        int[] cnt = new int[len];
        cnt[0] = 1;
        for (int i = 1, k = 10 - len + 1; i < len; i++, k++) {
            cnt[i] = cnt[i - 1] * k;
        }
        int ans = 0;
        if (len >= 2) {
            // 如果 n 的位数是 len 位，先计算位数少于 len 的数中，每一位都互不相同的正整数个数，并累
加
```

```
        // 所有1位数中，每一位都互不相同的正整数个数 = 9
        // 所有2位数中，每一位都互不相同的正整数个数 = 9 * 9
        // 所有3位数中，每一位都互不相同的正整数个数 = 9 * 9 * 8
        // 所有4位数中，每一位都互不相同的正整数个数 = 9 * 9 * 8 * 7
        // ...比len少的位数都累加...
        ans = 9;
        for (int i = 2, a = 9, b = 9; i < len; i++, b--) {
            a *= b;
            ans += a;
        }
    }
    // 如果n的位数是len位，已经计算了位数少于len个的情况
    // 下面计算一定有len位的数字中，<=n且每一位都互不相同的正整数个数
    int first = n / offset;
    // 小于num最高位数字的情况
    ans += (first - 1) * cnt[len - 1];
    // 后续累加上，等于num最高位数字的情况
    ans += f(cnt, n, len - 1, offset / 10, 1 << first);
    return ans;
}

// 之前已经确定了和num一样的前缀，且确定的部分一定不为空
// 还有len位没有确定
// 哪些数字已经选了，哪些数字没有选，用status表示
// 返回<=num且每一位数字都不一样的正整数有多少个
public static int f(int[] cnt, int num, int len, int offset, int status) {
    if (len == 0) {
        // num自己
        return 1;
    }
    int ans = 0;
    // first是num当前位的数字
    int first = (num / offset) % 10;
    for (int cur = 0; cur < first; cur++) {
        if ((status & (1 << cur)) == 0) {
            ans += cnt[len - 1];
        }
    }
    if ((status & (1 << first)) == 0) {
        ans += f(cnt, num, len - 1, offset / 10, status | (1 << first));
    }
    return ans;
}
```

```
    /**
     * 标准数位DP解法
     * 时间复杂度: O(log n * 2^10)
     * 空间复杂度: O(log n * 2^10)
     *
     * 解题思路:
     * 1. 使用标准数位DP框架
     * 2. 用位掩码记录已使用的数字
     * 3. 当前位不能选择已使用的数字
     * 4. 通过记忆化搜索避免重复计算
     */
    public static int countSpecialNumbersDP(int n) {
        char[] s = String.valueOf(n).toCharArray();

        // dp[i][mask][isLimit][isNum]
        // 表示处理到第i位，已使用数字的掩码为mask，是否受限制，是否已开始填数字时的方案数
        int[][][][] dp = new int[s.length][1 << 10][2][2];
        for (int i = 0; i < s.length; i++) {
            for (int j = 0; j < (1 << 10); j++) {
                dp[i][j][0][0] = -1;
                dp[i][j][0][1] = -1;
                dp[i][j][1][0] = -1;
                dp[i][j][1][1] = -1;
            }
        }

        return dfs(s, 0, 0, true, false, dp);
    }

    private static int dfs(char[] s, int pos, int mask, boolean isLimit, boolean isNum,
int[][][][] dp) {
        // 边界条件
        if (pos == s.length) {
            return isNum ? 1 : 0;  // 如果已经填了数字，返回1个有效数字
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][mask][0][0] != -1) {
            return dp[pos][mask][0][0];
        }

        int ans = 0;
```

```
    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfs(s, pos + 1, mask, false, false, dp);
    }

    // 确定当前位可以填入的数字范围
    int up = isLimit ? s[pos] - '0' : 9;

    // 枚举当前位可以填入的数字
    for (int d = isNum ? 0 : 1; d <= up; d++) {
        // 如果数字 d 还没有被使用过
        if ((mask & (1 << d)) == 0) {
            // 递归处理下一位
            ans += dfs(s, pos + 1, mask | (1 << d), isLimit && d == up, true, dp);
        }
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][mask][0][0] = ans;
    }

    return ans;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int n1 = 20;
    System.out.println("n = " + n1);
    System.out.println("原方法结果: " + countSpecialNumbers(n1));
    System.out.println("数位 DP 方法结果: " + countSpecialNumbersDP(n1));
    // 预期输出: 19 (除了 11 外的所有数字)

    // 测试用例 2
    int n2 = 5;
    System.out.println("n = " + n2);
    System.out.println("原方法结果: " + countSpecialNumbers(n2));
    System.out.println("数位 DP 方法结果: " + countSpecialNumbersDP(n2));
    // 预期输出: 5 (1,2,3,4,5 都满足条件)

    // 测试用例 3
```

```
        int n3 = 100;
        System.out.println("n = " + n3);
        System.out.println("原方法结果: " + countSpecialNumbers(n3));
        System.out.println("数位 DP 方法结果: " + countSpecialNumbersDP(n3));
        // 预期输出: 90
    }
}
```

==========

文件: Code04_NumbersWithRepeatedDigits.java

==========

```
package class084;

// 至少有 1 位重复的数字个数
// 给定正整数 n，返回在[1, n]范围内至少有 1 位重复数字的正整数个数
// 测试链接 : https://leetcode.cn/problems/numbers-with-repeated-digits/
public class Code04_NumbersWithRepeatedDigits {

    public static int numDupDigitsAtMostN(int n) {
        return n - countSpecialNumbers(n);
    }

    public static int countSpecialNumbers(int n) {
        int len = 1;
        int offset = 1;
        int tmp = n / 10;
        while (tmp > 0) {
            len++;
            offset *= 10;
            tmp /= 10;
        }
        int[] cnt = new int[len];
        cnt[0] = 1;
        for (int i = 1, k = 10 - len + 1; i < len; i++, k++) {
            cnt[i] = cnt[i - 1] * k;
        }
        int ans = 0;
        if (len >= 2) {
            ans = 9;
            for (int i = 2, a = 9, b = 9; i < len; i++, b--) {
                a *= b;
                ans += a;
```

```
            }
        }
        int first = n / offset;
        ans += (first - 1) * cnt[len - 1];
        ans += f(cnt, n, len - 1, offset / 10, 1 << first);
        return ans;
    }

    public static int f(int[] cnt, int num, int len, int offset, int status) {
        if (len == 0) {
            return 1;
        }
        int ans = 0;
        int first = (num / offset) % 10;
        for (int cur = 0; cur < first; cur++) {
            if ((status & (1 << cur)) == 0) {
                ans += cnt[len - 1];
            }
        }
        if ((status & (1 << first)) == 0) {
            ans += f(cnt, num, len - 1, offset / 10, status | (1 << first));
        }
        return ans;
    }

}
```

==========

文件: Code05_CountDigitOne.java

==========

```
package class084;

// 统计数字 1 的个数
// 给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数
// 测试链接 : https://leetcode.cn/problems/number-of-digit-one/
public class Code05_CountDigitOne {

    // 数位 DP 方法
    // 时间复杂度: O(log n) 每个数位最多计算两次(受限/不受限)
    // 空间复杂度: O(log n) 递归栈深度
    public static int countDigitOne(int n) {
        if (n <= 0) {
```

```
        return 0;
    }
    // 将数字 n 转换为字符数组，方便按位处理
    char[] s = String.valueOf(n).toCharArray();
    int len = s.length;
    // dp[i][count][isLimit] 表示处理到第 i 位，已经出现了 count 个 1，当前是否受限制时的方案数
    // -1 表示未计算过
    int[][][] dp = new int[len][len][2];
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < len; j++) {
            dp[i][j][0] = -1;
            dp[i][j][1] = -1;
        }
    }
    return f(s, 0, 0, true, dp);
}

// s: 数字的字符数组表示
// i: 当前处理到第几位
// count: 当前已经统计到的 1 的个数
// isLimit: 当前位是否受到上限限制
// dp: 记忆化数组
private static int f(char[] s, int i, int count, boolean isLimit, int[][][] dp) {
    // 递归终止条件：已经处理完所有数位
    if (i == s.length) {
        return count;
    }

    // 记忆化：如果已经计算过该状态，直接返回结果
    if (!isLimit && dp[i][count][0] != -1) {
        return dp[i][count][0];
    }

    // 确定当前位可以填入的数字范围
    // 如果受限制，最大只能填入 s[i]对应的数字，否则可以填入 0-9
    int up = isLimit ? s[i] - '0' : 9;
    int ans = 0;

    // 枚举当前位可以填入的数字
    for (int d = 0; d <= up; d++) {
        // 递归处理下一位
        // 如果当前位填入 1，则 count+1
        // 下一位是否受限制：当前位受限制且填入了上限值
```

```
            ans += f(s, i + 1, count + (d == 1 ? 1 : 0), isLimit && d == up, dp);
        }

        // 记忆化存储结果
        if (!isLimit) {
            dp[i][count][0] = ans;
        }
        return ans;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int n1 = 13;
        System.out.println("n = " + n1 + ", 数字 1 出现的次数: " + countDigitOne(n1));
        // 预期输出: 6 (数字 1, 10, 11, 12, 13 中 1 出现了 6 次)

        // 测试用例 2
        int n2 = 0;
        System.out.println("n = " + n2 + ", 数字 1 出现的次数: " + countDigitOne(n2));
        // 预期输出: 0

        // 测试用例 3
        int n3 = 100;
        System.out.println("n = " + n3 + ", 数字 1 出现的次数: " + countDigitOne(n3));
        // 预期输出: 21
    }
}

==========

文件: Code05_CountDigitOne.py
==========
# 统计数字 1 的个数
# 给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数
# 测试链接 : https://leetcode.cn/problems/number-of-digit-one/



class Solution:
    def countDigitOne(self, n: int) -> int:
        """
        数位 DP 方法
        时间复杂度: O(log n) 每个数位最多计算两次(受限/不受限)
```

```
    空间复杂度: O(log n) 递归栈深度
    """
    if n <= 0:
        return 0

    # 将数字 n 转换为字符串，方便按位处理
    s = str(n)
    length = len(s)

    # dp[i][count][is_limit] 表示处理到第 i 位，已经出现了 count 个 1，当前是否受限制时的方案数
    # 使用字典进行记忆化
    memo = {}

    def dfs(i, count, is_limit):
        # 递归终止条件：已经处理完所有数位
        if i == length:
            return count

        # 记忆化：如果已经计算过该状态，直接返回结果
        if (i, count, is_limit) in memo and not is_limit:
            return memo[(i, count, is_limit)]

        # 确定当前位可以填入的数字范围
        # 如果受限制，最大只能填入 s[i]对应的数字，否则可以填入 0-9
        up = int(s[i]) if is_limit else 9
        ans = 0

        # 枚举当前位可以填入的数字
        for d in range(up + 1):
            # 递归处理下一位
            # 如果当前位填入 1，则 count+1
            # 下一位是否受限制：当前位受限制且填入了上限值
            ans += dfs(i + 1, count + (1 if d == 1 else 0), is_limit and d == up)

        # 记忆化存储结果
        if not is_limit:
            memo[(i, count, is_limit)] = ans
        return ans

    return dfs(0, 0, True)


# 测试方法
```

```
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    n1 = 13
    print(f"n = {n1}, 数字 1 出现的次数: {solution.countDigitOne(n1)}")
    # 预期输出: 6 (数字 1, 10, 11, 12, 13 中 1 出现了 6 次)

    # 测试用例 2
    n2 = 0
    print(f"n = {n2}, 数字 1 出现的次数: {solution.countDigitOne(n2)}")
    # 预期输出: 0

    # 测试用例 3
    n3 = 100
    print(f"n = {n3}, 数字 1 出现的次数: {solution.countDigitOne(n3)}")
    # 预期输出: 21
```

==========

文件: Code06_NonNegativeIntegersWithoutConsecutiveOnes.java

==========

```
package class084;

// 不含连续 1 的非负整数
// 给定一个正整数 n，返回在[0, n]范围内不含连续 1 的非负整数的个数
// 测试链接 : https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/
public class Code06_NonNegativeIntegersWithoutConsecutiveOnes {

    // 数位 DP 方法
    // 时间复杂度: O(log n) 每个数位最多计算几次(受限/不受限, 前一位是 0/1)
    // 空间复杂度: O(log n) 递归栈深度
    public static int findIntegers(int n) {
        // 将数字 n 转换为字符数组，方便按位处理
        char[] s = Integer.toBinaryString(n).toCharArray();
        int len = s.length;
        // dp[i][prev][isLimit][isNum]
        // 表示处理到第 i 位，前一位是 prev，当前是否受限制，是否已填数字时的方案数
        // -1 表示未计算过
        int[][][][] dp = new int[len][2][2][2];
        for (int i = 0; i < len; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 2; k++) {
```

```
                dp[i][j][k][0] = -1;
                dp[i][j][k][1] = -1;
            }
        }
    }
    return f(s, 0, 0, true, false, dp);
}

// s: 数字的二进制字符数组表示
// i: 当前处理到第几位
// prev: 前一位填的数字
// isLimit: 当前位是否受到上限限制
// isNum: 是否已开始填数字（处理前导零）
// dp: 记忆化数组
private static int f(char[] s, int i, int prev, boolean isLimit, boolean isNum, int[][][][]
dp) {
    // 递归终止条件：已经处理完所有数位
    if (i == s.length) {
        // 如果已经填了数字，返回 1 个有效数字，否则返回 0
        return isNum ? 1 : 0;
    }

    // 记忆化：如果已经计算过该状态，直接返回结果
    if (!isLimit && isNum && dp[i][prev][0][0] != -1) {
        return dp[i][prev][0][0];
    }

    // 确定当前位可以填入的数字范围
    // 如果受限制，最大只能填入 s[i]对应的数字，否则可以填入 0-1
    int up = isLimit ? s[i] - '0' : 1;
    int ans = 0;

    // 如果前面没有填数字，可以跳过当前位（继续前导零）
    if (!isNum) {
        ans += f(s, i + 1, 0, false, false, dp);
    }

    // 枚举当前位可以填入的数字
    for (int d = isNum ? 0 : 1; d <= up; d++) {
        // 不能有连续的 1
        if (prev == 1 && d == 1) {
            continue;
        }
```

```
            // 递归处理下一位
            // 下一位是否受限制：当前位受限制且填入了上限值
            ans += f(s, i + 1, d, isLimit && d == up, true, dp);
        }

        // 记忆化存储结果
        if (!isLimit && isNum) {
            dp[i][prev][0][0] = ans;
        }
        return ans;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int n1 = 5;
        System.out.println("n = " + n1 + ", 不含连续 1 的非负整数个数: " + findIntegers(n1));
        // 预期输出: 5 (0, 1, 2, 4, 5 满足条件，3 的二进制是 11，不满足)

        // 测试用例 2
        int n2 = 1;
        System.out.println("n = " + n2 + ", 不含连续 1 的非负整数个数: " + findIntegers(n2));
        // 预期输出: 2 (0, 1 满足条件)

        // 测试用例 3
        int n3 = 2;
        System.out.println("n = " + n3 + ", 不含连续 1 的非负整数个数: " + findIntegers(n3));
        // 预期输出: 3 (0, 1, 2 满足条件)
    }
}

==========================================

文件: Code06_NonNegativeIntegersWithoutConsecutiveOnes.py
==========================================
# 不含连续 1 的非负整数
# 给定一个正整数 n，返回在[0, n]范围内不含连续 1 的非负整数的个数
# 测试链接 : https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/



class Solution:
    def findIntegers(self, n: int) -> int:
        """
```

数位 DP 方法
时间复杂度: O(log n) 每个数位最多计算几次(受限/不受限, 前一位是 0/1)
空间复杂度: O(log n) 递归栈深度
"""
# 将数字 n 转换为二进制字符串，方便按位处理
s = bin(n)[2:] # 去掉"0b"前缀
length = len(s)

# dp[i][prev][is_limit][is_num]
# 表示处理到第 i 位，前一位是 prev，当前是否受限制，是否已填数字时的方案数
# 使用字典进行记忆化
memo = {}

def dfs(i, prev, is_limit, is_num):
    # 递归终止条件：已经处理完所有数位
    if i == length:
        # 如果已经填了数字，返回 1 个有效数字，否则返回 0
        return 1 if is_num else 0

    # 记忆化：如果已经计算过该状态，直接返回结果
    if (i, prev, is_limit, is_num) in memo and not is_limit and is_num:
        return memo[(i, prev, is_limit, is_num)]

    # 确定当前位可以填入的数字范围
    # 如果受限制，最大只能填入 s[i]对应的数字，否则可以填入 0-1
    up = int(s[i]) if is_limit else 1
    ans = 0

    # 如果前面没有填数字，可以跳过当前位（继续前导零）
    if not is_num:
        ans += dfs(i + 1, 0, False, False)

    # 枚举当前位可以填入的数字
    for d in range(0 if is_num else 1, up + 1):
        # 不能有连续的 1
        if prev == 1 and d == 1:
            continue
        # 递归处理下一位
        # 下一位是否受限制：当前位受限制且填入了上限值
        ans += dfs(i + 1, d, is_limit and d == up, True)

    # 记忆化存储结果
    if not is_limit and is_num:

```
                memo[(i, prev, is_limit, is_num)] = ans
            return ans

        return dfs(0, 0, True, False)


# 测试方法
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    n1 = 5
    print(f"n = {n1}, 不含连续 1 的非负整数个数: {solution.findIntegers(n1)}")
    # 预期输出: 5 (0, 1, 2, 4, 5 满足条件，3 的二进制是 11，不满足)

    # 测试用例 2
    n2 = 1
    print(f"n = {n2}, 不含连续 1 的非负整数个数: {solution.findIntegers(n2)}")
    # 预期输出: 2 (0, 1 满足条件)

    # 测试用例 3
    n3 = 2
    print(f"n = {n3}, 不含连续 1 的非负整数个数: {solution.findIntegers(n3)}")
    # 预期输出: 3 (0, 1, 2 满足条件)

==============================================

文件: Code07_NumberOfBeautifulIntegersInTheRange.cpp
==============================================
#include <stdio.h>
#include <string.h>

// 范围中美丽整数的数目
// 给你两个正整数：low 和 high 。如果一个整数满足以下条件，我们称它为美丽整数：
// 1. 偶数数位的数目与奇数数位的数目相同；
// 2. 这个整数能被 k 整除。
// 请你返回范围 [low, high] 中美丽整数的数目。
// 测试链接 : https://leetcode.cn/problems/number-of-beautiful-integers-in-the-range/

// 定义最大数字长度
#define MAX_LEN 15
// 定义最大 k 值
#define MAX_K 25
```

```
// 全局记忆化数组
int dp[MAX_LEN][2][2][10][10][MAX_K];
char num_str[MAX_LEN];

/**
 * 数位 DP 解法
 * 时间复杂度: O(log(high) * 2 * 2 * 10 * 10 * k)
 * 空间复杂度: O(log(high) * 2 * 2 * 10 * 10 * k)
 *
 * 解题思路:
 * 1. 使用数位 DP 框架，逐位确定数字
 * 2. 状态需要记录：
 *    - 当前处理到第几位
 *    - 是否受到上界限制
 *    - 是否已开始填数字（处理前导零）
 *    - 奇数数位的个数
 *    - 偶数数位的个数
 *    - 当前数字对 k 的余数
 * 3. 通过记忆化搜索避免重复计算
 *
 * 最优解分析:
 * 该解法是标准的数位 DP 解法，时间复杂度与状态数相关，是解决此类问题的最优通用方法。
 */

// 初始化记忆化数组
void init_dp() {
    int i, j, l, m, o, p;
    for (i = 0; i < MAX_LEN; i++) {
        for (j = 0; j < 2; j++) {
            for (l = 0; l < 2; l++) {
                for (m = 0; m < 10; m++) {
                    for (o = 0; o < 10; o++) {
                        for (p = 0; p < MAX_K; p++) {
                            dp[i][j][l][m][o][p] = -1;
                        }
                    }
                }
            }
        }
    }
}
```

```
// 将整数转换为字符串
void int_to_str(int n, char* str) {
    int len = 0;
    int temp = n;
    int i;

    // 特殊处理 0
    if (n == 0) {
        str[0] = '0';
        str[1] = '\0';
        return;
    }

    // 计算数字位数
    while (temp > 0) {
        len++;
        temp /= 10;
    }

    // 从低位到高位填充字符串
    temp = n;
    for (i = len - 1; i >= 0; i--) {
        str[i] = (temp % 10) + '0';
        temp /= 10;
    }
    str[len] = '\0';
}

// 数位 DP 递归函数
int dfs(int pos, int isLimit, int isNum, int oddCount, int evenCount, int remainder, int k) {
    // 递归终止条件
    if (num_str[pos] == '\0') {
        // 只有当已经填了数字，且奇数数位和偶数数位个数相等，且能被 k 整除时才算一个美丽整数
        return isNum && oddCount == evenCount && remainder == 0 ? 1 : 0;
    }

    // 记忆化搜索
    if (!isLimit && isNum && dp[pos][0][0][oddCount][evenCount][remainder] != -1) {
        return dp[pos][0][0][oddCount][evenCount][remainder];
    }

    int ans = 0;
    int d;
```

```
    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfs(pos + 1, 0, 0, oddCount, evenCount, remainder, k);
    }

    // 确定当前位可以填入的数字范围
    int up = isLimit ? num_str[pos] - '0' : 9;

    // 枚举当前位可以填入的数字
    for (d = isNum ? 0 : 1; d <= up; d++) {
        // 根据数字的奇偶性更新奇数数位和偶数数位的个数
        int newOddCount = oddCount + (d % 2 == 1 ? 1 : 0);
        int newEvenCount = evenCount + (d % 2 == 0 ? 1 : 0);

        // 更新当前数字对 k 的余数
        int newRemainder = (remainder * 10 + d) % k;

        // 递归处理下一位
        ans += dfs(pos + 1, isLimit && d == up, 1, newOddCount, newEvenCount, newRemainder, k);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][0][0][oddCount][evenCount][remainder] = ans;
    }

    return ans;
}

// 计算[0, n]中美丽整数的个数
int beautifulIntegers(int n, int k) {
    if (n < 0) {
        return 0;
    }

    // 将数字转换为字符串
    int_to_str(n, num_str);

    // 初始化记忆化数组
    init_dp();

    // 调用 DFS 函数
```

```
    return dfs(0, 1, 0, 0, 0, 0, k);
}

// 主函数：计算范围中美丽整数的数目
int numberOfBeautifulIntegers(int low, int high, int k) {
    // 答案为[0, high]中的美丽整数个数减去[0, low-1]中的美丽整数个数
    return beautifulIntegers(high, k) - beautifulIntegers(low - 1, k);
}

// 测试函数
void testNumberOfBeautifulIntegers() {
    printf("=== 测试范围中美丽整数的数目 ===\n");

    // 测试用例 1
    int low1 = 10, high1 = 20, k1 = 3;
    int result1 = numberOfBeautifulIntegers(low1, high1, k1);
    printf("low = %d, high = %d, k = %d\n", low1, high1, k1);
    printf("美丽整数的数目: %d\n", result1);
    printf("预期输出: 2 (11 和 19 是美丽整数)\n\n");

    // 测试用例 2
    int low2 = 1, high2 = 10, k2 = 1;
    int result2 = numberOfBeautifulIntegers(low2, high2, k2);
    printf("low = %d, high = %d, k = %d\n", low2, high2, k2);
    printf("美丽整数的数目: %d\n", result2);
    printf("预期输出: 1 (10 是美丽整数)\n\n");

    // 测试用例 3
    int low3 = 5, high3 = 5, k3 = 2;
    int result3 = numberOfBeautifulIntegers(low3, high3, k3);
    printf("low = %d, high = %d, k = %d\n", low3, high3, k3);
    printf("美丽整数的数目: %d\n", result3);
    printf("预期输出: 0 (没有美丽整数)\n\n");
}

int main() {
    testNumberOfBeautifulIntegers();

    // 额外测试
    printf("=== 边界测试 ===\n");
    printf("low = 1, high = 1, k = 1: %d\n", numberOfBeautifulIntegers(1, 1, 1));
    printf("low = 0, high = 0, k = 1: %d\n", numberOfBeautifulIntegers(0, 0, 1));
```

```
    return 0;
}
```

==========

文件: Code07_NumberOfBeautifulIntegersInTheRange.java
==========

```
package class084;

// 范围中美丽整数的数目
// 给你两个正整数：low 和 high 。如果一个整数满足以下条件，我们称它为美丽整数：
// 1. 偶数数位的数目与奇数数位的数目相同；
// 2. 这个整数能被 k 整除。
// 请你返回范围 [low, high] 中美丽整数的数目。
// 测试链接 : https://leetcode.cn/problems/number-of-beautiful-integers-in-the-range/
public class Code07_NumberOfBeautifulIntegersInTheRange {

    /**
     * 数位 DP 解法
     * 时间复杂度: O(log(high) * 2 * 2 * 10 * 10 * k)
     * 空间复杂度: O(log(high) * 2 * 2 * 10 * 10 * k)
     *
     * 解题思路:
     * 1. 使用数位 DP 框架，逐位确定数字
     * 2. 状态需要记录：
     *    - 当前处理到第几位
     *    - 是否受到上界限制
     *    - 是否已开始填数字（处理前导零）
     *    - 奇数数位的个数
     *    - 偶数数位的个数
     *    - 当前数字对 k 的余数
     * 3. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
     * 该解法是标准的数位 DP 解法，时间复杂度与状态数相关，是解决此类问题的最优通用方法。
     */
    public static int numberOfBeautifulIntegers(int low, int high, int k) {
        // 答案为[0, high]中的美丽整数个数减去[0, low-1]中的美丽整数个数
        return beautifulIntegers(high, k) - beautifulIntegers(low - 1, k);
    }

    // 计算[0, n]中美丽整数的个数
    private static int beautifulIntegers(int n, int k) {
```

```
    if (n < 0) {
        return 0;
    }

    char[] s = String.valueOf(n).toCharArray();
    int len = s.length;

    // dp[pos][isLimit][isNum][oddCount][evenCount][remainder]
    // pos: 当前处理到第几位
    // isLimit: 是否受到上界限制
    // isNum: 是否已开始填数字
    // oddCount: 奇数数位的个数
    // evenCount: 偶数数位的个数
    // remainder: 当前数字对 k 的余数
    int[][][][][][] dp = new int[len][2][2][10][10][k];
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < 2; j++) {
            for (int l = 0; l < 2; l++) {
                for (int m = 0; m < 10; m++) {
                    for (int o = 0; o < 10; o++) {
                        for (int p = 0; p < k; p++) {
                            dp[i][j][l][m][o][p] = -1;
                        }
                    }
                }
            }
        }
    }

    return dfs(s, 0, true, false, 0, 0, 0, k, dp);
}

// 数位 DP 递归函数
private static int dfs(char[] s, int pos, boolean isLimit, boolean isNum,
                       int oddCount, int evenCount, int remainder, int k,
                       int[][][][][][] dp) {
    // 递归终止条件
    if (pos == s.length) {
        // 只有当已经填了数字，且奇数数位和偶数数位个数相等，且能被 k 整除时才算一个美丽整数
        return isNum && oddCount == evenCount && remainder == 0 ? 1 : 0;
    }

    // 记忆化搜索
```

```
    if (!isLimit && isNum && dp[pos][0][0][oddCount][evenCount][remainder] != -1) {
        return dp[pos][0][0][oddCount][evenCount][remainder];
    }

    int ans = 0;

    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfs(s, pos + 1, false, false, oddCount, evenCount, remainder, k, dp);
    }

    // 确定当前位可以填入的数字范围
    int up = isLimit ? s[pos] - '0' : 9;

    // 枚举当前位可以填入的数字
    for (int d = isNum ? 0 : 1; d <= up; d++) {
        // 根据数字的奇偶性更新奇数数位和偶数数位的个数
        int newOddCount = oddCount + (d % 2 == 1 ? 1 : 0);
        int newEvenCount = evenCount + (d % 2 == 0 ? 1 : 0);

        // 更新当前数字对 k 的余数
        int newRemainder = (remainder * 10 + d) % k;

        // 递归处理下一位
        ans += dfs(s, pos + 1, isLimit && d == up, true,
                   newOddCount, newEvenCount, newRemainder, k, dp);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][0][0][oddCount][evenCount][remainder] = ans;
    }

    return ans;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1
    int low1 = 10, high1 = 20, k1 = 3;
    System.out.println("low = " + low1 + ", high = " + high1 + ", k = " + k1);
    System.out.println("美丽整数的数目: " + numberOfBeautifulIntegers(low1, high1, k1));
    // 预期输出: 2 (11 和 19 是美丽整数)
```

```
        // 测试用例 2
        int low2 = 1, high2 = 10, k2 = 1;
        System.out.println("low = " + low2 + ", high = " + high2 + ", k = " + k2);
        System.out.println("美丽整数的数目: " + numberOfBeautifulIntegers(low2, high2, k2));
        // 预期输出: 1 (10 是美丽整数)

        // 测试用例 3
        int low3 = 5, high3 = 5, k3 = 2;
        System.out.println("low = " + low3 + ", high = " + high3 + ", k = " + k3);
        System.out.println("美丽整数的数目: " + numberOfBeautifulIntegers(low3, high3, k3));
        // 预期输出: 0 (没有美丽整数)
    }
}

==========================================

文件: Code07_NumberOfBeautifulIntegersInTheRange.py
==========================================
# 范围中美丽整数的数目
# 给你两个正整数：low 和 high 。如果一个整数满足以下条件，我们称它为美丽整数：
# 1. 偶数数位的数目与奇数数位的数目相同；
# 2. 这个整数能被 k 整除。
# 请你返回范围 [low, high] 中美丽整数的数目。
# 测试链接 : https://leetcode.cn/problems/number-of-beautiful-integers-in-the-range/



class Solution:
    def numberOfBeautifulIntegers(self, low: int, high: int, k: int) -> int:
        """
        数位 DP 解法
        时间复杂度: O(log(high) * 2 * 2 * 10 * 10 * k)
        空间复杂度: O(log(high) * 2 * 2 * 10 * 10 * k)

        解题思路:
        1. 使用数位 DP 框架，逐位确定数字
        2. 状态需要记录：
           - 当前处理到第几位
           - 是否受到上界限制
           - 是否已开始填数字（处理前导零）
           - 奇数数位的个数
           - 偶数数位的个数
           - 当前数字对 k 的余数
```

3. 通过记忆化搜索避免重复计算

最优解分析:
该解法是标准的数位 DP 解法，时间复杂度与状态数相关，是解决此类问题的最优通用方法。
"""

```
def beautiful_integers(n: int) -> int:
    if n < 0:
        return 0

    s = str(n)
    length = len(s)

    # 使用字典进行记忆化
    memo = {}

    def dfs(pos: int, is_limit: bool, is_num: bool,
            odd_count: int, even_count: int, remainder: int) -> int:
        # 递归终止条件
        if pos == length:
            # 只有当已经填了数字，且奇数数位和偶数数位个数相等，且能被 k 整除时才算一个美丽整数
            return 1 if is_num and odd_count == even_count and remainder == 0 else 0

        # 记忆化搜索
        if (pos, is_limit, is_num, odd_count, even_count, remainder) in memo and not is_limit and is_num:
            return memo[(pos, is_limit, is_num, odd_count, even_count, remainder)]

        ans = 0

        # 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if not is_num:
            ans += dfs(pos + 1, False, False, odd_count, even_count, remainder)

        # 确定当前位可以填入的数字范围
        up = int(s[pos]) if is_limit else 9

        # 枚举当前位可以填入的数字
        for d in range(0 if is_num else 1, up + 1):
            # 根据数字的奇偶性更新奇数数位和偶数数位的个数
            new_odd_count = odd_count + (1 if d % 2 == 1 else 0)
            new_even_count = even_count + (1 if d % 2 == 0 else 0)
```

```
                    # 更新当前数字对 k 的余数
                    new_remainder = (remainder * 10 + d) % k

                    # 递归处理下一位
                    ans += dfs(pos + 1, is_limit and d == up, True,
                               new_odd_count, new_even_count, new_remainder)

                # 记忆化存储
                if not is_limit and is_num:
                    memo[(pos, is_limit, is_num, odd_count, even_count, remainder)] = ans

                return ans

            return dfs(0, True, False, 0, 0, 0)

        # 答案为[0, high]中的美丽整数个数减去[0, low-1]中的美丽整数个数
        return beautiful_integers(high) - beautiful_integers(low - 1)


# 测试方法
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    low1, high1, k1 = 10, 20, 3
    print(f"low = {low1}, high = {high1}, k = {k1}")
    print(f"美丽整数的数目: {solution.numberOfBeautifulIntegers(low1, high1, k1)}")
    # 预期输出: 2 (11 和 19 是美丽整数)

    # 测试用例 2
    low2, high2, k2 = 1, 10, 1
    print(f"low = {low2}, high = {high2}, k = {k2}")
    print(f"美丽整数的数目: {solution.numberOfBeautifulIntegers(low2, high2, k2)}")
    # 预期输出: 1 (10 是美丽整数)

    # 测试用例 3
    low3, high3, k3 = 5, 5, 2
    print(f"low = {low3}, high = {high3}, k = {k3}")
    print(f"美丽整数的数目: {solution.numberOfBeautifulIntegers(low3, high3, k3)}")
    # 预期输出: 0 (没有美丽整数)

==========================================
```

文件: Code08_FindAllGoodStrings.cpp

```
==========================================
// 找到所有好字符串
// 给你两个长度为 n 的字符串 s1 和 s2，以及一个字符串 evil。请你返回好字符串的数目。
// 好字符串的定义是：它的长度为 n，字典序大于等于 s1，字典序小于等于 s2，且不包含 evil 为子字符串。
// 由于答案可能很大，返回答案对 10^9 + 7 取余的结果。
// 测试链接 : https://leetcode.cn/problems/find-all-good-strings/

#include <stdio.h>
#include <string.h>

// 定义最大长度
#define MAX_N 505
#define MOD 1000000007

// 全局记忆化数组
int dp[MAX_N][2][MAX_N];
int next_arr[MAX_N];

/**
 * 数位 DP + KMP 解法
 * 时间复杂度: O(n * m * 2 * 2 * 26) 其中 n 是字符串长度，m 是 evil 字符串长度
 * 空间复杂度: O(n * m * 2 * 2)
 *
 * 解题思路:
 * 1. 使用数位 DP 框架，逐位确定字符
 * 2. 结合 KMP 算法避免包含 evil 字符串
 * 3. 状态需要记录：
 *    - 当前处理到第几位
 *    - 是否受到上下界限制
 *    - 当前已匹配 evil 字符串的前缀长度（使用 KMP 的 next 数组）
 * 4. 通过记忆化搜索避免重复计算
 *
 * 最优解分析:
 * 该解法结合了数位 DP 和 KMP 算法，是解决此类问题的最优通用方法。
 */

// 计算 KMP 的 next 数组
void getNext(char* pattern, int patternLen) {
    next_arr[0] = -1;
    int i = 0, j = -1;
```

```
    while (i < patternLen) {
        if (j == -1 || pattern[i] == pattern[j]) {
            i++;
            j++;
            next_arr[i] = j;
        } else {
            j = next_arr[j];
        }
    }
}

// 初始化记忆化数组
void initDp(int n, int evilLen) {
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = 0; j < 2; j++) {
            for (k = 0; k < evilLen; k++) {
                dp[i][j][k] = -1;
            }
        }
    }
}

// 数位 DP 递归函数
int dfs(char* s, char* evil, int pos, int isLimit, int matchLen, int sLen, int evilLen) {
    // 递归终止条件
    if (pos == sLen) {
        return 1;
    }

    // 记忆化搜索
    if (!isLimit && dp[pos][0][matchLen] != -1) {
        return dp[pos][0][matchLen];
    }

    int ans = 0;

    // 确定当前位可以填入的字符范围
    char up = isLimit ? s[pos] : 'z';

    // 枚举当前位可以填入的字符
    char c;
    for (c = 'a'; c <= up; c++) {
```

```
        // 使用 KMP 算法计算填入字符 c 后匹配 evil 的长度
        int newMatchLen = matchLen;
        while (newMatchLen > 0 && evil[newMatchLen] != c) {
            newMatchLen = next_arr[newMatchLen];
        }
        if (newMatchLen < evilLen && evil[newMatchLen] == c) {
            newMatchLen++;
        }

        // 如果已经完全匹配 evil，则不能填入这个字符
        if (newMatchLen < evilLen) {
            // 递归处理下一位
            ans = ((long long)ans + dfs(s, evil, pos + 1, isLimit && c == s[pos], newMatchLen,
sLen, evilLen)) % MOD;
        }
    }

    // 记忆化存储
    if (!isLimit) {
        dp[pos][0][matchLen] = ans;
    }

    return ans;
}

// 检查字符串是否包含 evil 子串
int containsEvil(char* str, int strLen, char* evil, int evilLen) {
    int i, j;
    for (i = 0; i <= strLen - evilLen; i++) {
        int found = 1;
        for (j = 0; j < evilLen; j++) {
            if (str[i + j] != evil[j]) {
                found = 0;
                break;
            }
        }
        if (found) {
            return 1;
        }
    }
    return 0;
}
```

```
// 主函数：计算好字符串的数目
int findGoodStrings(int n, char* s1, char* s2, char* evil) {
    int evilLen = 0;
    while (evil[evilLen] != '\0') {
        evilLen++;
    }

    // 计算 KMP 的 next 数组
    getNext(evil, evilLen);

    // 计算[0, s2]中的好字符串个数
    initDp(n, evilLen);
    int count2 = dfs(s2, evil, 0, 1, 0, n, evilLen);

    // 计算[0, s1)中的好字符串个数
    initDp(n, evilLen);
    int count1 = dfs(s1, evil, 0, 1, 0, n, evilLen);

    // 检查 s1 本身是否是好字符串
    int s1Len = 0;
    while (s1[s1Len] != '\0') {
        s1Len++;
    }
    int s1IsGood = containsEvil(s1, s1Len, evil, evilLen) ? 0 : 1;

    // 返回结果
    return (int)((((long long)count2 - count1 + s1IsGood) % MOD) + MOD) % MOD;
}

// 测试函数
void testFindGoodStrings() {
    printf("=== 测试找到所有好字符串 ===\n");

    // 测试用例 1
    int n1 = 2;
    char s11[] = "aa";
    char s21[] = "da";
    char evil1[] = "b";
    int result1 = findGoodStrings(n1, s11, s21, evil1);
    printf("n = %d, s1 = \"%s\", s2 = \"%s\", evil = \"%s\"\n", n1, s11, s21, evil1);
    printf("好字符串的数目: %d\n", result1);
    printf("预期输出: 51\n\n");
```

```
    // 测试用例 2
    int n2 = 8;
    char s12[] = "leetcode";
    char s22[] = "leetgoes";
    char evil2[] = "leet";
    int result2 = findGoodStrings(n2, s12, s22, evil2);
    printf("n = %d, s1 = \"%s\", s2 = \"%s\", evil = \"%s\"\n", n2, s12, s22, evil2);
    printf("好字符串的数目: %d\n", result2);
    printf("预期输出: 0\n\n");

    // 测试用例 3
    int n3 = 2;
    char s13[] = "gx";
    char s23[] = "gz";
    char evil3[] = "x";
    int result3 = findGoodStrings(n3, s13, s23, evil3);
    printf("n = %d, s1 = \"%s\", s2 = \"%s\", evil = \"%s\"\n", n3, s13, s23, evil3);
    printf("好字符串的数目: %d\n", result3);
    printf("预期输出: 2\n\n");
}

int main() {
    testFindGoodStrings();

    // 额外测试
    printf("=== 边界测试 ===\n");
    int n = 1;
    char s1[] = "a";
    char s2[] = "z";
    char evil[] = "a";
    printf("n = %d, s1 = \"%s\", s2 = \"%s\", evil = \"%s\": %d\n",
           n, s1, s2, evil, findGoodStrings(n, s1, s2, evil));

    return 0;
}

==============================================

文件: Code08_FindAllGoodStrings.java
==============================================
package class084;

// 找到所有好字符串
```

```
// 给你两个长度为 n 的字符串 s1 和 s2，以及一个字符串 evil。请你返回好字符串的数目。
// 好字符串的定义是：它的长度为 n，字典序大于等于 s1，字典序小于等于 s2，且不包含 evil 为子字符串。
// 由于答案可能很大，返回答案对 10^9 + 7 取余的结果。
// 测试链接 : https://leetcode.cn/problems/find-all-good-strings/
public class Code08_FindAllGoodStrings {

    public static int MOD = 1000000007;

    /**
     * 数位 DP + KMP 解法
     * 时间复杂度: O(n * m * 2 * 2 * 26) 其中 n 是字符串长度，m 是 evil 字符串长度
     * 空间复杂度: O(n * m * 2 * 2)
     *
     * 解题思路:
     * 1. 使用数位 DP 框架，逐位确定字符
     * 2. 结合 KMP 算法避免包含 evil 字符串
     * 3. 状态需要记录：
     *    - 当前处理到第几位
     *    - 是否受到上下界限制
     *    - 当前已匹配 evil 字符串的前缀长度（使用 KMP 的 next 数组）
     * 4. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
     * 该解法结合了数位 DP 和 KMP 算法，是解决此类问题的最优通用方法。
     */
    public static int findGoodStrings(int n, String s1, String s2, String evil) {
        // 答案为[0, s2]中的好字符串个数减去[0, s1)中的好字符串个数，再加上 s1 本身是否是好字符串
        char[] cs1 = s1.toCharArray();
        char[] cs2 = s2.toCharArray();
        char[] cevil = evil.toCharArray();

        // 计算 KMP 的 next 数组
        int[] next = getNext(cevil);

        // 计算[0, s2]中的好字符串个数
        int[][][] dp2 = new int[n][2][evil.length()];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < evil.length(); k++) {
                    dp2[i][j][k] = -1;
                }
            }
```

```
        }
        int count2 = dfs(cs2, cevil, next, 0, true, 0, dp2);

        // 计算[0, s1)中的好字符串个数
        int[][][] dp1 = new int[n][2][evil.length()];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < evil.length(); k++) {
                    dp1[i][j][k] = -1;
                }
            }
        }
        int count1 = dfs(cs1, cevil, next, 0, true, 0, dp1);

        // 检查 s1 本身是否是好字符串
        int s1IsGood = (s1.indexOf(evil) == -1) ? 1 : 0;

        // 返回结果
        return (int) (((long) count2 - count1 + s1IsGood) % MOD + MOD) % MOD;
    }

    // 数位 DP 递归函数
    private static int dfs(char[] s, char[] evil, int[] next, int pos, boolean isLimit, int
matchLen, int[][][] dp) {
        // 递归终止条件
        if (pos == s.length) {
            return 1;
        }

        // 记忆化搜索
        if (!isLimit && dp[pos][0][matchLen] != -1) {
            return dp[pos][0][matchLen];
        }

        int ans = 0;

        // 确定当前位可以填入的字符范围
        char up = isLimit ? s[pos] : 'z';

        // 枚举当前位可以填入的字符
        for (char c = 'a'; c <= up; c++) {
            // 使用 KMP 算法计算填入字符 c 后匹配 evil 的长度
            int newMatchLen = matchLen;
```

```
            while (newMatchLen > 0 && evil[newMatchLen] != c) {
                newMatchLen = next[newMatchLen];
            }
            if (evil[newMatchLen] == c) {
                newMatchLen++;
            }

            // 如果已经完全匹配 evil，则不能填入这个字符
            if (newMatchLen < evil.length) {
                // 递归处理下一位
                ans = (ans + dfs(s, evil, next, pos + 1, isLimit && c == up, newMatchLen, dp)) %
MOD;
            }
        }

        // 记忆化存储
        if (!isLimit) {
            dp[pos][0][matchLen] = ans;
        }

        return ans;
    }

    // 计算 KMP 的 next 数组
    private static int[] getNext(char[] pattern) {
        int n = pattern.length;
        int[] next = new int[n + 1];
        next[0] = -1;
        int i = 0, j = -1;

        while (i < n) {
            if (j == -1 || pattern[i] == pattern[j]) {
                i++;
                j++;
                next[i] = j;
            } else {
                j = next[j];
            }
        }

        return next;
    }
```

```
    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int n1 = 2;
        String s11 = "aa", s21 = "da", evil1 = "b";
        System.out.println("n = " + n1 + ", s1 = \"" + s11 + "\", s2 = \"" + s21 + "\", evil = \""
+ evil1 + "\"");
        System.out.println("好字符串的数目: " + findGoodStrings(n1, s11, s21, evil1));
        // 预期输出: 51

        // 测试用例 2
        int n2 = 8;
        String s12 = "leetcode", s22 = "leetgoes", evil2 = "leet";
        System.out.println("n = " + n2 + ", s1 = \"" + s12 + "\", s2 = \"" + s22 + "\", evil = \""
+ evil2 + "\"");
        System.out.println("好字符串的数目: " + findGoodStrings(n2, s12, s22, evil2));
        // 预期输出: 0

        // 测试用例 3
        int n3 = 2;
        String s13 = "gx", s23 = "gz", evil3 = "x";
        System.out.println("n = " + n3 + ", s1 = \"" + s13 + "\", s2 = \"" + s23 + "\", evil = \""
+ evil3 + "\"");
        System.out.println("好字符串的数目: " + findGoodStrings(n3, s13, s23, evil3));
        // 预期输出: 2
    }
}

==============================================

文件: Code08_FindAllGoodStrings.py
==============================================
# 找到所有好字符串
# 给你两个长度为 n 的字符串 s1 和 s2，以及一个字符串 evil。请你返回好字符串的数目。
# 好字符串的定义是：它的长度为 n，字典序大于等于 s1，字典序小于等于 s2，且不包含 evil 为子字符
串。
# 由于答案可能很大，返回答案对 10^9 + 7 取余的结果。
# 测试链接 : https://leetcode.cn/problems/find-all-good-strings/

MOD = 1000000007

class Solution:
    def findGoodStrings(self, n: int, s1: str, s2: str, evil: str) -> int:
```

"""
数位 DP + KMP 解法
时间复杂度: O(n * m * 2 * 2 * 26) 其中 n 是字符串长度，m 是 evil 字符串长度
空间复杂度: O(n * m * 2 * 2)

解题思路:
1. 使用数位 DP 框架，逐位确定字符
2. 结合 KMP 算法避免包含 evil 字符串
3. 状态需要记录：
   - 当前处理到第几位
   - 是否受到上下界限制
   - 当前已匹配 evil 字符串的前缀长度（使用 KMP 的 next 数组）
4. 通过记忆化搜索避免重复计算

最优解分析:
该解法结合了数位 DP 和 KMP 算法，是解决此类问题的最优通用方法。
"""

```
def get_next(pattern: str) -> list:
    """计算 KMP 的 next 数组"""
    n = len(pattern)
    next_arr = [0] * (n + 1)
    next_arr[0] = -1
    i, j = 0, -1

    while i < n:
        if j == -1 or pattern[i] == pattern[j]:
            i += 1
            j += 1
            next_arr[i] = j
        else:
            j = next_arr[j]

    return next_arr

def dfs(s: str, evil: str, next_arr: list) -> int:
    """数位 DP 递归函数"""
    m = len(evil)
    # 使用字典进行记忆化
    memo = {}

    def helper(pos: int, is_limit: bool, match_len: int) -> int:
        # 递归终止条件
```

```
            if pos == len(s):
                return 1

            # 记忆化搜索
            if (pos, is_limit, match_len) in memo and not is_limit:
                return memo[(pos, is_limit, match_len)]

            ans = 0

            # 确定当前位可以填入的字符范围
            up = ord(s[pos]) if is_limit else ord('z')

            # 枚举当前位可以填入的字符
            for c in range(ord('a'), up + 1):
                char = chr(c)
                # 使用KMP算法计算填入字符c后匹配evil的长度
                new_match_len = match_len
                while new_match_len > 0 and evil[new_match_len] != char:
                    new_match_len = next_arr[new_match_len]
                if new_match_len < len(evil) and evil[new_match_len] == char:
                    new_match_len += 1

                # 如果已经完全匹配evil，则不能填入这个字符
                if new_match_len < len(evil):
                    # 递归处理下一位
                    ans = (ans + helper(pos + 1, is_limit and c == ord(s[pos]),
new_match_len)) % MOD

            # 记忆化存储
            if not is_limit:
                memo[(pos, is_limit, match_len)] = ans

            return ans

        return helper(0, True, 0)

    # 计算KMP的next数组
    next_arr = get_next(evil)

    # 答案为[0, s2]中的好字符串个数减去[0, s1)中的好字符串个数，再加上s1本身是否是好字符串
    count2 = dfs(s2, evil, next_arr)
    count1 = dfs(s1, evil, next_arr)
```

```
        # 检查 s1 本身是否是好字符串
        s1_is_good = 0 if evil in s1 else 1

        # 返回结果
        return (count2 - count1 + s1_is_good) % MOD

# 测试方法
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    n1, s11, s21, evil1 = 2, "aa", "da", "b"
    print(f"n = {n1}, s1 = \"{s11}\", s2 = \"{s21}\", evil = \"{evil1}\"")
    print(f"好字符串的数目: {solution.findGoodStrings(n1, s11, s21, evil1)}")
    # 预期输出: 51

    # 测试用例 2
    n2, s12, s22, evil2 = 8, "leetcode", "leetgoes", "leet"
    print(f"n = {n2}, s1 = \"{s12}\", s2 = \"{s22}\", evil = \"{evil2}\"")
    print(f"好字符串的数目: {solution.findGoodStrings(n2, s12, s22, evil2)}")
    # 预期输出: 0

    # 测试用例 3
    n3, s13, s23, evil3 = 2, "gx", "gz", "x"
    print(f"n = {n3}, s1 = \"{s13}\", s2 = \"{s23}\", evil = \"{evil3}\"")
    print(f"好字符串的数目: {solution.findGoodStrings(n3, s13, s23, evil3)}")
    # 预期输出: 2

==========================================

文件: Code09_DigitCount.cpp
==========================================
// ZJOI2010 数字计数
// 给定两个正整数 a 和 b，求在[a,b]中的所有整数中，每个数码（digit）各出现了多少次。
// 测试链接 : https://www.luogu.com.cn/problem/P2602

#include <stdio.h>
#include <string.h>

// 定义最大长度
#define MAX_LEN 20

// 全局记忆化数组
```

```
long long dp[MAX_LEN][2][2][10];
char num_str[MAX_LEN];

/**
 * 数位 DP 解法
 * 时间复杂度: O(log(b) * 2 * 2 * 10 * 10)
 * 空间复杂度: O(log(b) * 2 * 2 * 10)
 *
 * 解题思路:
 * 1. 使用数位 DP 框架，逐位确定数字
 * 2. 对于每个数字 0-9，分别计算其出现次数
 * 3. 状态需要记录：
 *    - 当前处理到第几位
 *    - 是否受到上界限制
 *    - 是否已开始填数字（处理前导零）
 *    - 当前数字的出现次数
 * 4. 通过记忆化搜索避免重复计算
 *
 * 最优解分析:
 * 该解法是标准的数位 DP 解法，时间复杂度与状态数相关，是解决此类问题的最优通用方法。
 */

// 初始化记忆化数组
void initDp() {
    int i, j, k, l;
    for (i = 0; i < MAX_LEN; i++) {
        for (j = 0; j < 2; j++) {
            for (k = 0; k < 2; k++) {
                for (l = 0; l < 10; l++) {
                    dp[i][j][k][l] = -1;
                }
            }
        }
    }
}

// 将长整数转换为字符串
void longTostr(long long n, char* str) {
    int len = 0;
    long long temp = n;
    int i;

    // 特殊处理 0
```

```
    if (n == 0) {
        str[0] = '0';
        str[1] = '\0';
        return;
    }

    // 计算数字位数
    while (temp > 0) {
        len++;
        temp /= 10;
    }

    // 从低位到高位填充字符串
    temp = n;
    for (i = len - 1; i >= 0; i--) {
        str[i] = (temp % 10) + '0';
        temp /= 10;
    }
    str[len] = '\0';
}

// 数位 DP 递归函数
long long dfs(int pos, int isLimit, int isNum, int targetDigit) {
    // 递归终止条件
    if (num_str[pos] == '\0') {
        return 0;
    }

    // 记忆化搜索
    if (!isLimit && isNum && dp[pos][0][0][targetDigit] != -1) {
        return dp[pos][0][0][targetDigit];
    }

    long long ans = 0;

    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfs(pos + 1, 0, 0, targetDigit);
    }

    // 确定当前位可以填入的数字范围
    int up = isLimit ? num_str[pos] - '0' : 9;
```

```
    // 枚举当前位可以填入的数字
    int d;
    for (d = isNum ? 0 : 1; d <= up; d++) {
        // 如果当前位填的是目标数字，则计数加 1
        long long count = (d == targetDigit) ? 1 : 0;

        // 递归处理下一位
        ans += count + dfs(pos + 1, isLimit && d == up, 1, targetDigit);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][0][0][targetDigit] = ans;
    }

    return ans;
}

// 计算[0, n]中每个数字的出现次数
void countDigitsUpTo(long long n, long long result[]) {
    if (n < 0) {
        for (int i = 0; i < 10; i++) {
            result[i] = 0;
        }
        return;
    }

    // 将数字转换为字符串
    longTostr(n, num_str);

    // 对于每个数字 0-9，分别计算其出现次数
    int digit;
    for (digit = 0; digit < 10; digit++) {
        // 初始化记忆化数组
        initDp();
        result[digit] = dfs(0, 1, 0, digit);
    }
}

// 主函数：计算区间中每个数字的出现次数
void countDigits(long long a, long long b, long long result[]) {
    long long countB[10];
    long long countA[10];
```

```
    // 计算[0, b]中的数字计数
    countDigitsUpTo(b, countB);

    // 计算[0, a-1]中的数字计数
    countDigitsUpTo(a - 1, countA);

    // 答案为[0, b]中的数字计数减去[0, a-1]中的数字计数
    for (int i = 0; i < 10; i++) {
        result[i] = countB[i] - countA[i];
    }
}

// 测试函数
void testDigitCount() {
    printf("=== 测试数字计数 ===\n");

    // 测试用例 1
    long long a1 = 1, b1 = 9;
    long long result1[10];
    countDigits(a1, b1, result1);
    printf("a = %lld, b = %lld\n", a1, b1);
    printf("结果: ");
    for (int i = 0; i < 10; i++) {
        printf("%lld ", result1[i]);
    }
    printf("\n预期: 0 1 1 1 1 1 1 1 1 1\n\n");

    // 测试用例 2
    long long a2 = 1, b2 = 99;
    long long result2[10];
    countDigits(a2, b2, result2);
    printf("a = %lld, b = %lld\n", a2, b2);
    printf("结果: ");
    for (int i = 0; i < 10; i++) {
        printf("%lld ", result2[i]);
    }
    printf("\n预期: 9 20 20 20 20 20 20 20 20 20\n\n");

    // 测试用例 3
    long long a3 = 1, b3 = 1000;
    long long result3[10];
    countDigits(a3, b3, result3);
```

```
    printf("a = %lld, b = %lld\n", a3, b3);
    printf("结果: ");
    for (int i = 0; i < 10; i++) {
        printf("%lld ", result3[i]);
    }
    printf("\n");
}

int main() {
    testDigitCount();

    // 额外测试
    printf("=== 边界测试 ===\n");
    long long a = 0, b = 0;
    long long result[10];
    countDigits(a, b, result);
    printf("a = %lld, b = %lld: ", a, b);
    for (int i = 0; i < 10; i++) {
        printf("%lld ", result[i]);
    }
    printf("\n");

    return 0;
}
```

```
================================================
```

文件: Code09_DigitCount.java

```
================================================
package class084;

// ZJOI2010 数字计数
// 给定两个正整数 a 和 b，求在[a,b]中的所有整数中，每个数码（digit）各出现了多少次。
// 测试链接 : https://www.luogu.com.cn/problem/P2602
public class Code09_DigitCount {

    /**
     * 数位 DP 解法
     * 时间复杂度: O(log(b) * 2 * 2 * 10 * 10)
     * 空间复杂度: O(log(b) * 2 * 2 * 10)
     *
     * 解题思路:
     * 1. 使用数位 DP 框架，逐位确定数字
```

```
 * 2. 对于每个数字 0-9，分别计算其出现次数
 * 3. 状态需要记录：
 *    - 当前处理到第几位
 *    - 是否受到上界限制
 *    - 是否已开始填数字（处理前导零）
 *    - 当前数字的出现次数
 * 4. 通过记忆化搜索避免重复计算
 *
 * 最优解分析:
 * 该解法是标准的数位 DP 解法，时间复杂度与状态数相关，是解决此类问题的最优通用方法。
 */
public static long[] countDigits(long a, long b) {
    // 答案为[0, b]中的数字计数减去[0, a-1]中的数字计数
    long[] countB = countDigitsUpTo(b);
    long[] countA = countDigitsUpTo(a - 1);

    long[] result = new long[10];
    for (int i = 0; i < 10; i++) {
        result[i] = countB[i] - countA[i];
    }

    return result;
}

// 计算[0, n]中每个数字的出现次数
private static long[] countDigitsUpTo(long n) {
    if (n < 0) {
        return new long[10];
    }

    char[] s = String.valueOf(n).toCharArray();
    int len = s.length;

    // dp[pos][isLimit][isNum][digit]
    // pos: 当前处理到第几位
    // isLimit: 是否受到上界限制
    // isNum: 是否已开始填数字
    // digit: 当前统计的数字(0-9)
    long[][][][] dp = new long[len][2][2][10];
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                for (int l = 0; l < 10; l++) {
```

```
                    dp[i][j][k][1] = -1;
                }
            }
        }

        long[] result = new long[10];
        for (int digit = 0; digit < 10; digit++) {
            result[digit] = dfs(s, 0, true, false, digit, dp);
        }

        return result;
    }

    // 数位DP递归函数
    private static long dfs(char[] s, int pos, boolean isLimit, boolean isNum, int targetDigit,
long[][][][] dp) {
        // 递归终止条件
        if (pos == s.length) {
            return 0;
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][0][0][targetDigit] != -1) {
            return dp[pos][0][0][targetDigit];
        }

        long ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
            ans += dfs(s, pos + 1, false, false, targetDigit, dp);
        }

        // 确定当前位可以填入的数字范围
        int up = isLimit ? s[pos] - '0' : 9;

        // 枚举当前位可以填入的数字
        for (int d = isNum ? 0 : 1; d <= up; d++) {
            // 如果当前位填的是目标数字，则计数加1
            long count = (d == targetDigit) ? 1 : 0;

            // 递归处理下一位
```

```
            ans += count + dfs(s, pos + 1, isLimit && d == up, true, targetDigit, dp);
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][0][0][targetDigit] = ans;
        }

        return ans;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        long a1 = 1, b1 = 9;
        System.out.println("a = " + a1 + ", b = " + b1);
        long[] result1 = countDigits(a1, b1);
        for (int i = 0; i < 10; i++) {
            System.out.print(result1[i] + " ");
        }
        System.out.println();
        // 预期输出: 0 1 1 1 1 1 1 1 1 1

        // 测试用例 2
        long a2 = 1, b2 = 99;
        System.out.println("a = " + a2 + ", b = " + b2);
        long[] result2 = countDigits(a2, b2);
        for (int i = 0; i < 10; i++) {
            System.out.print(result2[i] + " ");
        }
        System.out.println();
        // 预期输出: 9 20 20 20 20 20 20 20 20 20

        // 测试用例 3
        long a3 = 1, b3 = 1000;
        System.out.println("a = " + a3 + ", b = " + b3);
        long[] result3 = countDigits(a3, b3);
        for (int i = 0; i < 10; i++) {
            System.out.print(result3[i] + " ");
        }
        System.out.println();
    }
}
```

==========

文件: Code09_DigitCount.py
==========
```
# ZJOI2010 数字计数
# 给定两个正整数 a 和 b，求在[a,b]中的所有整数中，每个数码（digit）各出现了多少次。
# 测试链接 : https://www.luogu.com.cn/problem/P2602


class Solution:
    def countDigits(self, a: int, b: int) -> list:
        """
        数位 DP 解法
        时间复杂度: O(log(b) * 2 * 2 * 10 * 10)
        空间复杂度: O(log(b) * 2 * 2 * 10)

        解题思路:
        1. 使用数位 DP 框架，逐位确定数字
        2. 对于每个数字 0-9，分别计算其出现次数
        3. 状态需要记录：
           - 当前处理到第几位
           - 是否受到上界限制
           - 是否已开始填数字（处理前导零）
           - 当前数字的出现次数
        4. 通过记忆化搜索避免重复计算

        最优解分析:
        该解法是标准的数位 DP 解法，时间复杂度与状态数相关，是解决此类问题的最优通用方法。
        """

        def count_digits_up_to(n: int) -> list:
            if n < 0:
                return [0] * 10

            s = str(n)
            length = len(s)

            def dfs(pos: int, is_limit: bool, is_num: bool, target_digit: int) -> int:
                # 递归终止条件
                if pos == length:
                    return 0
```

```
            # 记忆化搜索
            if (pos, is_limit, is_num, target_digit) in memo and not is_limit and is_num:
                return memo[(pos, is_limit, is_num, target_digit)]

            ans = 0

            # 如果还没开始填数字，可以选择跳过当前位（处理前导零）
            if not is_num:
                ans += dfs(pos + 1, False, False, target_digit)

            # 确定当前位可以填入的数字范围
            up = int(s[pos]) if is_limit else 9

            # 枚举当前位可以填入的数字
            for d in range(0 if is_num else 1, up + 1):
                # 如果当前位填的是目标数字，则计数加 1
                count = 1 if d == target_digit else 0

                # 递归处理下一位
                ans += count + dfs(pos + 1, is_limit and d == up, True, target_digit)

            # 记忆化存储
            if not is_limit and is_num:
                memo[(pos, is_limit, is_num, target_digit)] = ans

            return ans

        result = [0] * 10
        for digit in range(10):
            memo = {}  # 每个数字都需要独立的记忆化数组
            result[digit] = dfs(0, True, False, digit)

        return result

    # 答案为[0, b]中的数字计数减去[0, a-1]中的数字计数
    count_b = count_digits_up_to(b)
    count_a = count_digits_up_to(a - 1)

    return [count_b[i] - count_a[i] for i in range(10)]

# 测试方法
if __name__ == "__main__":
    solution = Solution()
```

```
    # 测试用例 1
    a1, b1 = 1, 9
    print(f"a = {a1}, b = {b1}")
    result1 = solution.countDigits(a1, b1)
    print(" ".join(map(str, result1)))
    # 预期输出: 0 1 1 1 1 1 1 1 1 1

    # 测试用例 2
    a2, b2 = 1, 99
    print(f"a = {a2}, b = {b2}")
    result2 = solution.countDigits(a2, b2)
    print(" ".join(map(str, result2)))
    # 预期输出: 9 20 20 20 20 20 20 20 20 20

    # 测试用例 3
    a3, b3 = 1, 1000
    print(f"a = {a3}, b = {b3}")
    result3 = solution.countDigits(a3, b3)
    print(" ".join(map(str, result3)))

==============================================

文件: Code10_DigitCountEnhanced.cpp
==============================================
// 洛谷 P2602 数字计数
// 题目链接: https://www.luogu.com.cn/problem/P2602
// 题目描述: 给定两个正整数 a,b，求在区间[a,b]中的数，数字 0~9 的出现次数之和。

#include <iostream>
#include <vector>
#include <string>
#include <cstring>
using namespace std;

class DigitCountEnhanced {
private:
    // 记忆化数组，用于优化递归过程
    static long long dp[20][20][2][2];

    /**
     * 数位 DP 递归函数
     *
```

```
     * @param digits 数字的字符数组
     * @param pos 当前处理到第几位
     * @param count 当前已经出现 digit 的次数
     * @param isLimit 是否受到上界限制
     * @param isNum 是否已开始填数字（处理前导零）
     * @param digit 要统计的数字
     * @return 从当前状态开始，digit 出现的次数总和
     */
    static long long dfs(const vector<int>& digits, int pos, int count, bool isLimit, bool isNum,
int digit) {
        // 递归终止条件
        if (pos == digits.size()) {
            return count;
        }

        // 记忆化搜索 - 只有当没有限制且已经开始填数字时才可以使用缓存
        if (!isLimit && isNum && dp[pos][count][0][0] != -1) {
            return dp[pos][count][0][0];
        }

        long long ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
            ans += dfs(digits, pos + 1, count, false, false, digit);
        }

        // 确定当前位可以填入的数字范围
        int upper = isLimit ? digits[pos] : 9;

        // 枚举当前位可以填入的数字
        int start = isNum ? 0 : 1;
        for (int d = start; d <= upper; d++) {
            int newCount = count;
            if (d == digit) {
                newCount++;
            }

            bool newIsLimit = isLimit && (d == upper);
            bool newIsNum = isNum || (d > 0);

            // 递归处理下一位
            ans += dfs(digits, pos + 1, newCount, newIsLimit, newIsNum, digit);
```

```
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][count][0][0] = ans;
        }

        return ans;
    }

    /**
     * 优化版数位DP递归函数
     */
    static long long dfsOptimized(const vector<int>& digits, int pos, int count, bool isLimit,
bool hasLeadingZero, int digit) {
        // 递归终止条件
        if (pos == digits.size()) {
            return count;
        }

        // 记忆化搜索
        if (!isLimit && !hasLeadingZero && dp[pos][count][0][0] != -1) {
            return dp[pos][count][0][0];
        }

        long long ans = 0;

        // 确定当前位可以填入的数字范围
        int upper = isLimit ? digits[pos] : 9;

        // 枚举当前位可以填入的数字
        for (int d = 0; d <= upper; d++) {
            bool newIsLimit = isLimit && (d == upper);
            bool newHasLeadingZero = hasLeadingZero && (d == 0);

            int newCount = count;
            if (!newHasLeadingZero && d == digit) {
                newCount++;
            }

            // 递归处理下一位
            ans += dfsOptimized(digits, pos + 1, newCount, newIsLimit, newHasLeadingZero, digit);
        }
```

```
    // 记忆化存储
    if (!isLimit && !hasLeadingZero) {
        dp[pos][count][0][0] = ans;
    }

    return ans;
}

/**
 * 将数字转换为字符数组形式
 */
static vector<int> toDigitVector(long long n) {
    vector<int> digits;
    if (n == 0) {
        digits.push_back(0);
        return digits;
    }
    while (n > 0) {
        digits.push_back(n % 10);
        n /= 10;
    }
    reverse(digits.begin(), digits.end());
    return digits;
}

/**
 * 统计[0, n]范围内 digit 出现的次数
 */
static long long countDigit(long long n, int digit) {
    if (n < 0) {
        return 0;
    }
    if (n < 10) {
        // 对于小于 10 的数，直接判断 digit 是否小于等于 n
        return 0LL + (digit == 0 ? 0 : (digit <= n ? 1 : 0));
    }

    vector<int> digits = toDigitVector(n);

    // 初始化 dp 为-1，表示未计算过
    memset(dp, -1, sizeof(dp));
```

```
    // 处理特殊情况：当 digit 是 0 时，我们需要额外处理
    if (digit == 0) {
        // 对于 0 的计数，我们需要计算[1, n]中 0 的个数
        // 因为在数位 DP 中，前导零不算作有效数字
        long long total = 0;
        // 直接遍历每一位，计算每一位上 0 出现的次数
        for (int i = 1; i <= digits.size(); i++) {
            // 计算高位部分
            long long high = n / (long long)pow(10, i);
            // 计算当前位的数字
            int current = (n / (long long)pow(10, i - 1)) % 10;
            // 计算低位部分
            long long low = n % (long long)pow(10, i - 1);

            if (current > 0) {
                total += high * (long long)pow(10, i - 1);
            } else {
                total += (high - 1) * (long long)pow(10, i - 1) + (low + 1);
            }
        }

        return total;
    }

    return dfs(digits, 0, 0, true, false, digit);
}

/**
 * 优化版统计函数
 */
static long long countDigitOptimized(long long n, int digit) {
    if (n < 0) {
        return 0;
    }
    if (n < 10) {
        return 0LL + (digit == 0 ? 0 : (digit <= n ? 1 : 0));
    }

    vector<int> digits = toDigitVector(n);

    // 初始化 dp 为-1，表示未计算过
    memset(dp, -1, sizeof(dp));
```

```
        return dfsOptimized(digits, 0, 0, true, true, digit);
    }

public:
    /**
     * 数位DP解法
     * 时间复杂度: O(10 * log(b) * 10 * 2)
     * 空间复杂度: O(10 * log(b) * 10 * 2)
     *
     * 解题思路:
     * 1. 将问题转化为统计[0, b]中每个数字出现的次数减去[0, a-1]中每个数字出现的次数
     * 2. 对于每个数字d(0-9)，使用数位DP统计在[0, n]范围内d出现的次数
     * 3. 状态需要记录：当前处理到第几位、当前已经出现d的次数、是否受到上界限制、是否已经开始填数字（处理前导零）
     * 4. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
     * 该解法是标准的数位DP解法，能够高效处理大范围的输入，是解决此类问题的最优通用方法。
     */
    static vector<long long> digitCount(long long a, long long b) {
        vector<long long> result(10, 0);
        // 计算[0, b]中每个数字出现的次数减去[0, a-1]中每个数字出现的次数
        for (int d = 0; d < 10; d++) {
            result[d] = countDigit(b, d) - countDigit(a - 1, d);
        }
        return result;
    }

    /**
     * 使用优化版数位DP函数的解决方案
     */
    static vector<long long> digitCountOptimized(long long a, long long b) {
        vector<long long> result(10, 0);
        for (int d = 0; d < 10; d++) {
            result[d] = countDigitOptimized(b, d) - countDigitOptimized(a - 1, d);
        }
        return result;
    }
};

// 初始化静态成员变量
long long DigitCountEnhanced::dp[20][20][2][2];
```

```
int main() {
    // 测试用例 1: a=1, b=10
    // 预期输出: [1, 2, 1, 1, 1, 1, 1, 1, 1, 1]
    // 解释: 0 出现 1 次(10), 1 出现 2 次(1, 10), 其他数字各出现 1 次
    long long a1 = 1, b1 = 10;
    vector<long long> result1 = DigitCountEnhanced::digitCount(a1, b1);
    cout << "测试用例 1: a=" << a1 << ", b=" << b1 << endl;
    cout << "各数字出现次数: ";
    for (long long cnt : result1) {
        cout << cnt << " ";
    }
    cout << endl;

    // 测试用例 2: a=123, b=456
    long long a2 = 123, b2 = 456;
    vector<long long> result2 = DigitCountEnhanced::digitCount(a2, b2);
    cout << "\n 测试用例 2: a=" << a2 << ", b=" << b2 << endl;
    cout << "各数字出现次数: ";
    for (long long cnt : result2) {
        cout << cnt << " ";
    }
    cout << endl;

    return 0;
}

==========================================

文件: Code10_DigitCountEnhanced.java
==========================================
package class084;

// 洛谷 P2602 [ZJOI2010] 数字计数
// 题目链接: https://www.luogu.com.cn/problem/P2602
// 题目描述: 给定两个正整数 a 和 b，求在[a,b]范围上的所有整数中，每个数码(digit)各出现了多少次。
// 注意：Code09 已经包含了这个题目的实现，但为了展示更详细的分析和优化，这里提供一个更全面的解法
public class Code10_DigitCountEnhanced {

    /**
     * 数位 DP 解法 - 统计[0,n]中每个数字出现的次数
     * 时间复杂度: O(10 * log(n) * 2 * 2)
     * 空间复杂度: O(10 * log(n) * 2 * 2)
     *
```

```
     * 解题思路:
     * 1. 将问题转化为统计[0, b]中每个数字出现的次数减去[0, a-1]中每个数字出现的次数
     * 2. 对于每个数字 d(0-9)，使用数位 DP 计算它在[0,n]中出现的次数
     * 3. 状态需要记录：当前处理到第几位、数字 d、是否受到上界限制、是否已经开始填数字、当前 d 出现的次数
     * 4. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
     * 该解法是标准的数位 DP 解法，能够高效处理大范围的输入，是解决此类问题的最优通用方法。
     */
    public static long[] countDigits(long a, long b) {
        // 计算[0, b]中每个数字出现的次数
        long[] countB = countDigitsUpTo(b);
        // 计算[0, a-1]中每个数字出现的次数
        long[] countA = countDigitsUpTo(a - 1);
        // 两者相减得到[a, b]中每个数字出现的次数
        for (int i = 0; i < 10; i++) {
            countB[i] -= countA[i];
        }
        return countB;
    }

    // 计算[0, n]中每个数字出现的次数
    private static long[] countDigitsUpTo(long n) {
        long[] result = new long[10];
        if (n < 0) {
            return result;
        }

        // 分别统计每个数字出现的次数
        for (int d = 0; d <= 9; d++) {
            result[d] = countDigit(n, d);
        }

        return result;
    }

    // 计算[0, n]中数字 d 出现的次数
    private static long countDigit(long n, int d) {
        String s = String.valueOf(n);
        int len = s.length();

        // dp[pos][isLimit][isNum] = 从 pos 位置开始，限制条件为 isLimit，是否已开始填数字为 isNum
```

时，数字 d 出现的次数

```
        long[][][] dp = new long[len][2][2];
        // 初始化 dp 为-1，表示未计算过
        for (int i = 0; i < len; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 2; k++) {
                    dp[i][j][k] = -1;
                }
            }
        }

        return dfs(s.toCharArray(), 0, true, false, d, 0, dp);
    }

    /**
     * 数位 DP 递归函数
     *
     * @param s 数字的字符数组
     * @param pos 当前处理到第几位
     * @param isLimit 是否受到上界限制
     * @param isNum 是否已开始填数字（处理前导零）
     * @param target 需要统计的数字
     * @param count 当前累计的目标数字出现次数
     * @param dp 记忆化数组
     * @return 从当前状态开始，目标数字出现的总次数
     */
    private static long dfs(char[] s, int pos, boolean isLimit, boolean isNum, int target, long
count, long[][][] dp) {
        // 递归终止条件
        if (pos == s.length) {
            // 如果已经填了数字，返回当前累计的目标数字出现次数
            return isNum ? count : 0;
        }

        // 记忆化搜索 - 只有当没有限制且已经开始填数字时才可以使用缓存
        if (!isLimit && isNum && dp[pos][0][0] != -1) {
            // 注意：这里返回的是该状态下目标数字出现的次数，需要加上当前的 count
            return dp[pos][0][0] + count * Math.pow(10, s.length - pos - 1);
        }

        long ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
```

```
    if (!isNum) {
        ans += dfs(s, pos + 1, false, false, target, count, dp);
    }

    // 确定当前位可以填入的数字范围
    int upper = isLimit ? s[pos] - '0' : 9;

    // 枚举当前位可以填入的数字
    for (int d = isNum ? 0 : 1; d <= upper; d++) {
        boolean newIsLimit = isLimit && (d == upper);
        boolean newIsNum = isNum || (d > 0);
        long newCount = count + (newIsNum && d == target ? 1 : 0);

        // 对于当前位填的是 target 的情况，还需要考虑后续位的可能情况
        if (d == target && newIsNum) {
            // 计算后续位有多少种可能性
            long suffix = 1;
            for (int i = pos + 1; i < s.length; i++) {
                suffix *= 10;
            }
            // 如果当前位不受限制，后续位可以随便填
            if (!newIsLimit) {
                ans += suffix;
            } else {
                // 如果当前位受限制，需要计算受限制情况下的后续位数
                String suffixStr = String.copyValueOf(s, pos + 1, s.length - pos - 1);
                ans += suffixStr.isEmpty() ? 1 : Long.parseLong(suffixStr) + 1;
            }
        } else {
            // 递归处理下一位
            ans += dfs(s, pos + 1, newIsLimit, newIsNum, target, newCount, dp);
        }
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][0][0] = ans - count * Math.pow(10, s.length - pos - 1);
    }

    return ans;
}

/**
```

```
     * 数学方法 - 直接计算每个数字出现的次数
     * 时间复杂度: O(log(n))
     * 空间复杂度: O(1)
     *
     * 解题思路:
     * 1. 逐位分析每个数字的出现次数
     * 2. 对于每一位，考虑高位、当前位和低位的影响
     * 3. 根据当前位与目标数字的大小关系，分情况计算
     *
     * 优点：时间复杂度更低
     * 缺点：实现更复杂，且不容易扩展到其他约束条件
     */
    private static long countDigitMath(long n, int d) {
        if (n < 0) {
            return 0;
        }

        long count = 0;
        long divisor = 1;

        while (n / divisor > 0) {
            long high = n / (divisor * 10); // 高位
            long current = (n / divisor) % 10; // 当前位
            long low = n % divisor; // 低位

            // 处理特殊情况：d=0 时，高位不能全为 0
            if (d == 0) {
                if (high > 0) {
                    high--;
                } else {
                    // 高位全为 0，无法形成有效数字
                    divisor *= 10;
                    continue;
                }
            }

            // 分情况讨论
            if (current > d) {
                count += (high + 1) * divisor;
            } else if (current == d) {
                count += high * divisor + low + 1;
            } else {
                count += high * divisor;
```

```
            }

            divisor *= 10;
        }

        return count;
    }

    // 测试代码
    public static void main(String[] args) {
        // 测试用例 1: a=1, b=13
        // 预期输出: 1 出现 6 次(1,10,11,12,13), 2 出现 2 次(2,12), 3 出现 2 次(3,13), 其他数字各出现 1
次
        long a1 = 1, b1 = 13;
        long[] result1 = countDigits(a1, b1);
        System.out.println("测试用例 1: a=" + a1 + ", b=" + b1);
        for (int i = 0; i < 10; i++) {
            System.out.println("数字 " + i + " 出现了 " + result1[i] + " 次");
        }

        // 测试用例 2: a=100, b=200
        long a2 = 100, b2 = 200;
        long[] result2 = countDigits(a2, b2);
        System.out.println("\n 测试用例 2: a=" + a2 + ", b=" + b2);
        for (int i = 0; i < 10; i++) {
            System.out.println("数字 " + i + " 出现了 " + result2[i] + " 次");
        }
    }
}

==========================================

文件: Code10_DigitCountEnhanced.py
==========================================
# 洛谷 P2602 数字计数
# 题目链接: https://www.luogu.com.cn/problem/P2602
# 题目描述: 给定两个正整数 a,b，求在区间[a,b]中的数，数字 0~9 的出现次数之和。

class DigitCountEnhanced:
    @staticmethod
    def digit_count(a: int, b: int) -> list:
        """
        数位 DP 解法
```

时间复杂度: O(10 * log(b) * 10 * 2)
空间复杂度: O(10 * log(b) * 10 * 2)

解题思路:
1. 将问题转化为统计[0, b]中每个数字出现的次数减去[0, a-1]中每个数字出现的次数
2. 对于每个数字 d(0-9)，使用数位 DP 统计在[0, n]范围内 d 出现的次数
3. 状态需要记录：当前处理到第几位、当前已经出现 d 的次数、是否受到上界限制、是否已经开始填数字（处理前导零）
4. 通过记忆化搜索避免重复计算

最优解分析:
该解法是标准的数位 DP 解法，能够高效处理大范围的输入，是解决此类问题的最优通用方法。
"""
计算[0, b]中每个数字出现的次数减去[0, a-1]中每个数字出现的次数
return [DigitCountEnhanced._count_digit(b, d) - DigitCountEnhanced._count_digit(a - 1, d) for d in range(10)]

```
    @staticmethod
    def _count_digit(n: int, digit: int) -> int:
        """统计[0, n]范围内 digit 出现的次数"""
        if n < 0:
            return 0
        if n < 10:
            # 对于小于 10 的数，直接判断 digit 是否小于等于 n
            return 0 if digit == 0 else (1 if digit <= n else 0)

        s = str(n)

        from functools import lru_cache

        @lru_cache(maxsize=None)
        def dfs(pos: int, count: int, is_limit: bool, is_num: bool) -> int:
            """
            数位 DP 递归函数

            参数:
            - pos: 当前处理到第几位
            - count: 当前已经出现 digit 的次数
            - is_limit: 是否受到上界限制
            - is_num: 是否已开始填数字（处理前导零）

            返回:
            - 从当前状态开始，digit 出现的次数总和
```

```
    """
    # 递归终止条件
    if pos == len(s):
        return count

    res = 0

    # 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if not is_num:
        res += dfs(pos + 1, count, False, False)

    # 确定当前位可以填入的数字范围
    upper = int(s[pos]) if is_limit else 9

    # 枚举当前位可以填入的数字
    start = 0 if is_num else 1
    for d in range(start, upper + 1):
        new_count = count
        if d == digit:
            new_count += 1

        new_is_limit = is_limit and (d == upper)
        new_is_num = is_num or (d > 0)

        # 递归处理下一位
        res += dfs(pos + 1, new_count, new_is_limit, new_is_num)

    return res

# 处理特殊情况：当 digit 是 0 时，我们需要额外处理
if digit == 0:
    # 对于 0 的计数，我们需要计算[1, n]中 0 的个数
    # 因为在数位 DP 中，前导零不算作有效数字
    total = 0
    # 直接遍历每一位，计算每一位上 0 出现的次数
    for i in range(1, len(s) + 1):
        # 计算高位部分
        high = n // (10 ** i)
        # 计算当前位的数字
        current = (n // (10 ** (i - 1))) % 10
        # 计算低位部分
        low = n % (10 ** (i - 1))
```

```
            if current > 0:
                total += (high) * (10 ** (i - 1))
            else:
                total += (high - 1) * (10 ** (i - 1)) + (low + 1)

        return total

    # 清除缓存，避免之前的计算影响当前结果
    dfs.cache_clear()

    return dfs(0, 0, True, False)

@staticmethod
def _count_digit_optimized(n: int, digit: int) -> int:
    """
    优化版数位 DP 统计函数
    1. 更高效地处理前导零问题
    2. 优化记忆化搜索的状态设计
    """
    if n < 0:
        return 0
    if n < 10:
        return 0 if digit == 0 else (1 if digit <= n else 0)

    s = str(n)

    from functools import lru_cache

    @lru_cache(maxsize=None)
    def dfs_optimized(pos: int, count: int, is_limit: bool, has_leading_zero: bool) -> int:
        """优化版数位 DP 递归函数"""
        # 递归终止条件
        if pos == len(s):
            return count

        res = 0

        # 确定当前位可以填入的数字范围
        upper = int(s[pos]) if is_limit else 9

        # 枚举当前位可以填入的数字
        for d in range(0, upper + 1):
            new_is_limit = is_limit and (d == upper)
```

```
                new_has_leading_zero = has_leading_zero and (d == 0)

                new_count = count
                if not new_has_leading_zero and d == digit:
                    new_count += 1

                # 递归处理下一位
                res += dfs_optimized(pos + 1, new_count, new_is_limit, new_has_leading_zero)

            return res

        # 清除缓存，避免之前的计算影响当前结果
        dfs_optimized.cache_clear()

        return dfs_optimized(0, 0, True, True)

    @staticmethod
    def digit_count_optimized(a: int, b: int) -> list:
        """使用优化版数位DP函数的解决方案"""
        return [DigitCountEnhanced._count_digit_optimized(b, d) -
DigitCountEnhanced._count_digit_optimized(a - 1, d) for d in range(10)]

# 测试代码
if __name__ == "__main__":
    # 测试用例1: a=1, b=10
    # 预期输出: [1, 2, 1, 1, 1, 1, 1, 1, 1, 1]
    # 解释: 0出现1次(10)，1出现2次(1, 10)，其他数字各出现1次
    a1, b1 = 1, 10
    result1 = DigitCountEnhanced.digit_count(a1, b1)
    print(f"测试用例1: a={a1}, b={b1}")
    print(f"各数字出现次数: {result1}")

    # 测试用例2: a=123, b=456
    a2, b2 = 123, 456
    result2 = DigitCountEnhanced.digit_count(a2, b2)
    print(f"\n测试用例2: a={a2}, b={b2}")
    print(f"各数字出现次数: {result2}")

==========================================

文件: Code11_SimilarDistribution.cpp
==========================================
// 洛谷P4127 [AHOI2009] 同类分布
```

```
// 题目链接: https://www.luogu.com.cn/problem/P4127
// 题目描述: 给出两个数 a,b，求出[a,b]中各位数字之和能整除原数的数的个数。

#include <iostream>
#include <vector>
#include <string>
#include <cstring>
using namespace std;

class SimilarDistribution {
private:
    // 记忆化数组，用于优化递归过程
    static long long dp[20][163][163][2][2];

    /**
     * 数位 DP 递归函数
     *
     * @param digits 数字的字符数组
     * @param pos 当前处理到第几位
     * @param sum 当前数位和
     * @param mod 当前数值对 s_sum 的余数
     * @param isLimit 是否受到上界限制
     * @param isNum 是否已开始填数字（处理前导零）
     * @param s_sum 目标数位和
     * @return 从当前状态开始，符合条件的数的个数
     */
    static long long dfs(const vector<int>& digits, int pos, int sum, int mod, bool isLimit, bool
isNum, int s_sum) {
        // 递归终止条件
        if (pos == digits.size()) {
            // 只有当已经填了数字，且数位和等于 s_sum，且数值能被 s_sum 整除时才算符合条件
            return isNum && sum == s_sum && mod == 0 ? 1 : 0;
        }

        // 记忆化搜索 - 只有当没有限制且已经开始填数字时才可以使用缓存
        if (!isLimit && isNum && dp[pos][sum][mod][0][0] != -1) {
            return dp[pos][sum][mod][0][0];
        }

        long long ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
```

```
            ans += dfs(digits, pos + 1, sum, mod, false, false, s_sum);
        }

        // 确定当前位可以填入的数字范围
        int upper = isLimit ? digits[pos] : 9;

        // 枚举当前位可以填入的数字
        int start = isNum ? 0 : 1;
        for (int d = start; d <= upper; d++) {
            int newSum = sum + d;
            // 如果新的数位和已经超过了 s_sum，可以提前剪枝
            if (newSum > s_sum) {
                continue;
            }

            // 更新当前数值对 s_sum 的余数
            int newMod = (mod * 10 + d) % s_sum;
            bool newIsLimit = isLimit && (d == upper);
            bool newIsNum = isNum || (d > 0);

            // 递归处理下一位
            ans += dfs(digits, pos + 1, newSum, newMod, newIsLimit, newIsNum, s_sum);
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][sum][mod][0][0] = ans;
        }

        return ans;
    }

    /**
     * 优化版数位 DP 递归函数
     */
    static long long dfsOptimized(const vector<int>& digits, int pos, int sum, int mod, bool
isLimit, bool isNum, int s_sum) {
        // 递归终止条件
        if (pos == digits.size()) {
            return isNum && sum == s_sum && mod == 0 ? 1 : 0;
        }

        // 记忆化搜索
```

```
    if (!isLimit && isNum && dp[pos][0][mod][0][0] != -1) {
        return dp[pos][0][mod][0][0];
    }

    long long ans = 0;

    // 处理前导零
    if (!isNum) {
        ans += dfsOptimized(digits, pos + 1, sum, mod, false, false, s_sum);
    }

    // 确定当前位可以填入的数字范围
    int upper = isLimit ? digits[pos] : 9;

    // 枚举当前位可以填入的数字
    int start = isNum ? 0 : 1;
    for (int d = start; d <= upper; d++) {
        int newSum = sum + d;
        if (newSum > s_sum) {
            continue;
        }

        int newMod = (mod * 10 + d) % s_sum;
        bool newIsLimit = isLimit && (d == upper);
        bool newIsNum = isNum || (d > 0);

        ans += dfsOptimized(digits, pos + 1, newSum, newMod, newIsLimit, newIsNum, s_sum);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][0][mod][0][0] = ans;
    }

    return ans;
}

/**
 * 将数字转换为字符数组形式
 */
static vector<int> toDigitVector(long long n) {
    vector<int> digits;
    if (n == 0) {
```

```
        digits.push_back(0);
        return digits;
    }
    while (n > 0) {
        digits.push_back(n % 10);
        n /= 10;
    }
    reverse(digits.begin(), digits.end());
    return digits;
}

/**
 * 统计数位和等于 s_sum 且能被 s_sum 整除的数的个数
 */
static long long countNumbersWithSumDivisibleBy(long long n, int s_sum) {
    vector<int> digits = toDigitVector(n);

    // 初始化 dp 为-1，表示未计算过
    memset(dp, -1, sizeof(dp));

    return dfs(digits, 0, 0, 0, true, false, s_sum);
}

/**
 * 优化版统计函数
 */
static long long countNumbersWithSumDivisibleByOptimized(long long n, int s_sum) {
    vector<int> digits = toDigitVector(n);

    // 初始化 dp 为-1，表示未计算过
    memset(dp, -1, sizeof(dp));

    return dfsOptimized(digits, 0, 0, 0, true, false, s_sum);
}

/**
 * 计算[0, n]中符合条件的数的个数
 */
static long long countValidNumbers(long long n) {
    if (n < 1) {
        return 0; // 0 不符合条件，因为不能除以 0
    }
```

```
        string s = to_string(n);
        int maxSum = s.length() * 9; // 最大可能的数位和
        long long result = 0;

        // 枚举所有可能的数位和 s
        for (int s_sum = 1; s_sum <= maxSum; s_sum++) {
            // 对于每个数位和 s_sum，统计满足条件的数的个数
            result += countNumbersWithSumDivisibleBy(n, s_sum);
        }

        return result;
    }

public:
    /**
     * 数位 DP 解法
     * 时间复杂度: O(log(b) * 162 * 162 * 2 * 2)
     * 空间复杂度: O(log(b) * 162 * 162 * 2 * 2)
     *
     * 解题思路:
     * 1. 将问题转化为统计[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
     * 2. 由于数位和 s 的最大可能值为 9*18=162（假设最多 18 位数），我们可以枚举数位和 s
     * 3. 对于每个数位和 s，使用数位 DP 统计满足以下条件的数 x 的个数：
     *    - x 的数位和等于 s
     *    - x 能被 s 整除
     * 4. 状态需要记录：当前处理到第几位、当前数位和、当前数值对 s 的余数、是否受到上界限制、是否已经开始填数字
     * 5. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
     * 该解法是标准的数位 DP 解法，能够高效处理大范围的输入，是解决此类问题的最优通用方法。
     * 时间复杂度中的 162 来自于数位和的最大可能值（9*18）。
     */
    static long long similarDistribution(long long a, long long b) {
        // 计算[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
        return countValidNumbers(b) - countValidNumbers(a - 1);
    }
};

// 初始化静态成员变量
long long SimilarDistribution::dp[20][163][163][2][2];

int main() {
```

```
    // 测试用例 1: a=1, b=20
    // 预期输出: 19 (所有数都符合条件，除了那些数位和为 0 的数)
    long long a1 = 1, b1 = 20;
    long long result1 = SimilarDistribution::similarDistribution(a1, b1);
    cout << "测试用例 1: a=" << a1 << ", b=" << b1 << endl;
    cout << "符合条件的数的个数: " << result1 << endl;

    // 测试用例 2: a=1, b=100
    long long a2 = 1, b2 = 100;
    long long result2 = SimilarDistribution::similarDistribution(a2, b2);
    cout << "\n 测试用例 2: a=" << a2 << ", b=" << b2 << endl;
    cout << "符合条件的数的个数: " << result2 << endl;

    return 0;
}

==============================================

文件: Code11_SimilarDistribution.java
==============================================
package class084;

// 洛谷 P4127 [AHOI2009] 同类分布
// 题目链接: https://www.luogu.com.cn/problem/P4127
// 题目描述: 给出两个数 a,b，求出[a,b]中各位数字之和能整除原数的数的个数。
public class Code11_SimilarDistribution {

    /**
     * 数位 DP 解法
     * 时间复杂度: O(log(b) * 162 * 162 * 2 * 2)
     * 空间复杂度: O(log(b) * 162 * 162 * 2 * 2)
     *
     * 解题思路:
     * 1. 将问题转化为统计[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
     * 2. 由于数位和 s 的最大可能值为 9*18=162（假设最多 18 位数），我们可以枚举数位和 s
     * 3. 对于每个数位和 s，使用数位 DP 统计满足以下条件的数 x 的个数：
     *    - x 的数位和等于 s
     *    - x 能被 s 整除
     * 4. 状态需要记录：当前处理到第几位、当前数位和、当前数值对 s 的余数、是否受到上界限制、是否已经开始填数字
     * 5. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
```

```
 * 该解法是标准的数位 DP 解法，能够高效处理大范围的输入，是解决此类问题的最优通用方法。
 * 时间复杂度中的 162 来自于数位和的最大可能值（9*18）。
 */
public static int similarDistribution(long a, long b) {
    // 计算[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
    return countValidNumbers(b) - countValidNumbers(a - 1);
}

// 计算[0, n]中符合条件的数的个数
private static int countValidNumbers(long n) {
    if (n < 1) {
        return 0; // 0 不符合条件，因为不能除以 0
    }

    String s = String.valueOf(n);
    int len = s.length();
    int maxSum = len * 9; // 最大可能的数位和
    int result = 0;

    // 枚举所有可能的数位和 s
    for (int s_sum = 1; s_sum <= maxSum; s_sum++) {
        // 对于每个数位和 s_sum，统计满足条件的数的个数
        result += countNumbersWithSumDivisibleBy(s.toCharArray(), s_sum);
    }

    return result;
}

// 统计数位和等于 s_sum 且能被 s_sum 整除的数的个数
private static int countNumbersWithSumDivisibleBy(char[] digits, int s_sum) {
    int len = digits.length;
    // dp[pos][sum][mod][isLimit][isNum]
    // pos: 当前处理到第几位
    // sum: 当前数位和
    // mod: 当前数值对 s_sum 的余数
    // isLimit: 是否受到上界限制
    // isNum: 是否已开始填数字
    int[][][][][] dp = new int[len][s_sum + 1][s_sum][2][2];

    // 初始化 dp 为-1，表示未计算过
    for (int i = 0; i < len; i++) {
        for (int j = 0; j <= s_sum; j++) {
            for (int k = 0; k < s_sum; k++) {
```

```
                for (int l = 0; l < 2; l++) {
                    for (int m = 0; m < 2; m++) {
                        dp[i][j][k][l][m] = -1;
                    }
                }
            }
        }
    }

    return dfs(digits, 0, 0, 0, true, false, s_sum, dp);
}

/**
 * 数位 DP 递归函数
 *
 * @param digits 数字的字符数组
 * @param pos 当前处理到第几位
 * @param sum 当前数位和
 * @param mod 当前数值对 s_sum 的余数
 * @param isLimit 是否受到上界限制
 * @param isNum 是否已开始填数字（处理前导零）
 * @param s_sum 目标数位和
 * @param dp 记忆化数组
 * @return 从当前状态开始，符合条件的数的个数
 */
private static int dfs(char[] digits, int pos, int sum, int mod, boolean isLimit, boolean
isNum, int s_sum, int[][][][][] dp) {
    // 递归终止条件
    if (pos == digits.length) {
        // 只有当已经填了数字，且数位和等于 s_sum，且数值能被 s_sum 整除时才算符合条件
        return isNum && sum == s_sum && mod == 0 ? 1 : 0;
    }

    // 记忆化搜索
    if (!isLimit && isNum && dp[pos][sum][mod][0][0] != -1) {
        return dp[pos][sum][mod][0][0];
    }

    int ans = 0;

    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfs(digits, pos + 1, sum, mod, false, false, s_sum, dp);
```

```
    }

    // 确定当前位可以填入的数字范围
    int upper = isLimit ? digits[pos] - '0' : 9;

    // 枚举当前位可以填入的数字
    for (int d = isNum ? 0 : 1; d <= upper; d++) {
        int newSum = sum + d;
        // 如果新的数位和已经超过了 s_sum，可以提前剪枝
        if (newSum > s_sum) {
            continue;
        }

        // 更新当前数值对 s_sum 的余数
        int newMod = (mod * 10 + d) % s_sum;
        boolean newIsLimit = isLimit && (d == upper);
        boolean newIsNum = isNum || (d > 0);

        // 递归处理下一位
        ans += dfs(digits, pos + 1, newSum, newMod, newIsLimit, newIsNum, s_sum, dp);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][sum][mod][0][0] = ans;
    }

    return ans;
}

/**
 * 优化版本：减少一维状态
 * 注意到我们只关心数位和等于 s_sum 的情况，所以可以在递归过程中进行剪枝
 * 当 sum > s_sum 时，可以直接跳过
 */
private static int countNumbersWithSumDivisibleByOptimized(char[] digits, int s_sum) {
    int len = digits.length;
    // dp[pos][mod][isLimit][isNum]
    // pos: 当前处理到第几位
    // mod: 当前数值对 s_sum 的余数
    // isLimit: 是否受到上界限制
    // isNum: 是否已开始填数字
    int[][][][] dp = new int[len][s_sum][2][2];
```

```
        // 初始化 dp 为-1，表示未计算过
        for (int i = 0; i < len; i++) {
            for (int j = 0; j < s_sum; j++) {
                for (int k = 0; k < 2; k++) {
                    for (int l = 0; l < 2; l++) {
                        dp[i][j][k][l] = -1;
                    }
                }
            }
        }

        return dfsOptimized(digits, 0, 0, 0, true, false, s_sum, dp);
    }

    private static int dfsOptimized(char[] digits, int pos, int sum, int mod, boolean isLimit,
boolean isNum, int s_sum, int[][][][] dp) {
        // 递归终止条件
        if (pos == digits.length) {
            // 只有当已经填了数字，且数位和等于 s_sum，且数值能被 s_sum 整除时才算符合条件
            return isNum && sum == s_sum && mod == 0 ? 1 : 0;
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][mod][0][0] != -1) {
            return dp[pos][mod][0][0];
        }

        int ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
            ans += dfsOptimized(digits, pos + 1, sum, mod, false, false, s_sum, dp);
        }

        // 确定当前位可以填入的数字范围
        int upper = isLimit ? digits[pos] - '0' : 9;

        // 枚举当前位可以填入的数字
        for (int d = isNum ? 0 : 1; d <= upper; d++) {
            int newSum = sum + d;
            // 如果新的数位和已经超过了 s_sum，可以提前剪枝
            if (newSum > s_sum) {
```

```
                continue;
            }

            // 更新当前数值对 s_sum 的余数
            int newMod = (mod * 10 + d) % s_sum;
            boolean newIsLimit = isLimit && (d == upper);
            boolean newIsNum = isNum || (d > 0);

            // 递归处理下一位
            ans += dfsOptimized(digits, pos + 1, newSum, newMod, newIsLimit, newIsNum, s_sum, dp);
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][mod][0][0] = ans;
        }

        return ans;
    }

    // 测试代码
    public static void main(String[] args) {
        // 测试用例 1: a=1, b=20
        // 预期输出: 19 (所有数都符合条件，除了那些数位和为 0 的数)
        long a1 = 1, b1 = 20;
        int result1 = similarDistribution(a1, b1);
        System.out.println("测试用例 1: a=" + a1 + ", b=" + b1);
        System.out.println("符合条件的数的个数: " + result1);

        // 测试用例 2: a=1, b=100
        long a2 = 1, b2 = 100;
        int result2 = similarDistribution(a2, b2);
        System.out.println("\n 测试用例 2: a=" + a2 + ", b=" + b2);
        System.out.println("符合条件的数的个数: " + result2);
    }
}
```

==========

文件: Code11_SimilarDistribution.py

==========

```
# 洛谷 P4127 [AHOI2009] 同类分布
# 题目链接: https://www.luogu.com.cn/problem/P4127
```

题目描述: 给出两个数 a,b，求出[a,b]中各位数字之和能整除原数的数的个数。

```
class SimilarDistribution:

    @staticmethod
    def similar_distribution(a: int, b: int) -> int:
        """
        数位 DP 解法
        时间复杂度: O(log(b) * 162 * 162 * 2 * 2)
        空间复杂度: O(log(b) * 162 * 162 * 2 * 2)

        解题思路:
        1. 将问题转化为统计[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
        2. 由于数位和 s 的最大可能值为 9*18=162（假设最多 18 位数），我们可以枚举数位和 s
        3. 对于每个数位和 s，使用数位 DP 统计满足以下条件的数 x 的个数：
           - x 的数位和等于 s
           - x 能被 s 整除
        4. 状态需要记录：当前处理到第几位、当前数位和、当前数值对 s 的余数、是否受到上界限制、是否已经开始填数字
        5. 通过记忆化搜索避免重复计算

        最优解分析:
        该解法是标准的数位 DP 解法，能够高效处理大范围的输入，是解决此类问题的最优通用方法。
        时间复杂度中的 162 来自于数位和的最大可能值（9*18）。
        """
        # 计算[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
        return SimilarDistribution._count_valid_numbers(b) -
SimilarDistribution._count_valid_numbers(a - 1)

    @staticmethod
    def _count_valid_numbers(n: int) -> int:
        """计算[0, n]中符合条件的数的个数"""
        if n < 1:
            return 0  # 0 不符合条件，因为不能除以 0

        s = str(n)
        max_sum = len(s) * 9  # 最大可能的数位和
        result = 0

        # 枚举所有可能的数位和 s
        for s_sum in range(1, max_sum + 1):
            # 对于每个数位和 s_sum，统计满足条件的数的个数
            result += SimilarDistribution._count_numbers_with_sum_divisible_by(s, s_sum)
```

```
        return result

    @staticmethod
    def _count_numbers_with_sum_divisible_by(digits_str: str, s_sum: int) -> int:
        """统计数位和等于 s_sum 且能被 s_sum 整除的数的个数"""
        digits = list(digits_str)
        len_digits = len(digits)

        # 使用 lru_cache 进行记忆化搜索
        # 注意：在 Python 中，我们需要将可变参数转换为不可变参数才能使用 lru_cache
        from functools import lru_cache

        @lru_cache(maxsize=None)
        def dfs(pos: int, current_sum: int, current_mod: int, is_limit: bool, is_num: bool) ->
int:
            """
            数位 DP 递归函数

            参数:
            - pos: 当前处理到第几位
            - current_sum: 当前数位和
            - current_mod: 当前数值对 s_sum 的余数
            - is_limit: 是否受到上界限制
            - is_num: 是否已开始填数字（处理前导零）

            返回:
            - 从当前状态开始，符合条件的数的个数
            """
            # 递归终止条件
            if pos == len_digits:
                # 只有当已经填了数字，且数位和等于 s_sum，且数值能被 s_sum 整除时才算符合条件
                return 1 if (is_num and current_sum == s_sum and current_mod == 0) else 0

            res = 0

            # 如果还没开始填数字，可以选择跳过当前位（处理前导零）
            if not is_num:
                res += dfs(pos + 1, current_sum, current_mod, False, False)

            # 确定当前位可以填入的数字范围
            upper = int(digits[pos]) if is_limit else 9
```

```
            # 枚举当前位可以填入的数字
            start = 0 if is_num else 1
            for d in range(start, upper + 1):
                new_sum = current_sum + d
                # 如果新的数位和已经超过了 s_sum，可以提前剪枝
                if new_sum > s_sum:
                    continue

                # 更新当前数值对 s_sum 的余数
                new_mod = (current_mod * 10 + d) % s_sum
                new_is_limit = is_limit and (d == upper)
                new_is_num = is_num or (d > 0)

                # 递归处理下一位
                res += dfs(pos + 1, new_sum, new_mod, new_is_limit, new_is_num)

            return res

        # 清除缓存，避免之前的计算影响当前结果
        dfs.cache_clear()

        return dfs(0, 0, 0, True, False)

    @staticmethod
    def _count_numbers_with_sum_divisible_by_optimized(digits_str: str, s_sum: int) -> int:
        """
        优化版本：减少状态维度
        注意到我们只关心数位和等于 s_sum 的情况，所以可以在递归过程中进行剪枝
        当 sum > s_sum 时，可以直接跳过
        """
        digits = list(digits_str)
        len_digits = len(digits)

        from functools import lru_cache

        @lru_cache(maxsize=None)
        def dfs_optimized(pos: int, current_sum: int, current_mod: int, is_limit: bool, is_num:
bool) -> int:
            """优化版数位 DP 递归函数"""
            # 递归终止条件
            if pos == len_digits:
                # 只有当已经填了数字，且数位和等于 s_sum，且数值能被 s_sum 整除时才算符合条件
                return 1 if (is_num and current_sum == s_sum and current_mod == 0) else 0
```

```
            res = 0

            # 如果还没开始填数字，可以选择跳过当前位（处理前导零）
            if not is_num:
                res += dfs_optimized(pos + 1, current_sum, current_mod, False, False)

            # 确定当前位可以填入的数字范围
            upper = int(digits[pos]) if is_limit else 9

            # 枚举当前位可以填入的数字
            start = 0 if is_num else 1
            for d in range(start, upper + 1):
                new_sum = current_sum + d
                # 如果新的数位和已经超过了 s_sum，可以提前剪枝
                if new_sum > s_sum:
                    continue

                # 更新当前数值对 s_sum 的余数
                new_mod = (current_mod * 10 + d) % s_sum
                new_is_limit = is_limit and (d == upper)
                new_is_num = is_num or (d > 0)

                # 递归处理下一位
                res += dfs_optimized(pos + 1, new_sum, new_mod, new_is_limit, new_is_num)

            return res

        # 清除缓存，避免之前的计算影响当前结果
        dfs_optimized.cache_clear()

        return dfs_optimized(0, 0, 0, True, False)

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: a=1, b=20
    # 预期输出: 19 (所有数都符合条件，除了那些数位和为 0 的数)
    a1, b1 = 1, 20
    result1 = SimilarDistribution.similar_distribution(a1, b1)
    print(f"测试用例 1: a={a1}, b={b1}")
    print(f"符合条件的数的个数: {result1}")

    # 测试用例 2: a=1, b=100
```

```
    a2, b2 = 1, 100
    result2 = SimilarDistribution.similar_distribution(a2, b2)
    print(f"\n测试用例 2: a={a2}, b={b2}")
    print(f"符合条件的数的个数: {result2}")

================================================

文件: Code12_BeautifulNumbers.cpp
================================================
// Codeforces 55D. Beautiful numbers
// 题目链接: https://codeforces.com/problemset/problem/55/D
// 题目描述: 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。给定区间[l,r]，求其中美丽数字的个数。

#include <iostream>
#include <vector>
#include <string>
#include <cstring>
using namespace std;

class BeautifulNumbers {
private:
    static const int MOD = 2520; // 1-9 的最小公倍数

    // 记忆化数组，用于优化递归过程
    static long long dp[20][48][MOD][2][2];

    /**
     * 计算两个数的最大公约数
     */
    static int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    /**
     * 计算两个数的最小公倍数
     */
    static int lcm(int a, int b) {
        if (a == 0 || b == 0) {
            return a + b;
        }
        return a / gcd(a, b) * b;
    }
```

```
/**
 * 预处理：获取 2520 的所有因数
 */
static vector<int> getFactors() {
    vector<int> factors;
    for (int i = 1; i <= MOD; i++) {
        if (MOD % i == 0) {
            factors.push_back(i);
        }
    }
    return factors;
}

/**
 * 获取 LCM 到索引的映射
 */
static vector<int> getLcmToIndex(const vector<int>& factors) {
    vector<int> lcmToIndex(MOD + 1);
    for (int i = 0; i < factors.size(); i++) {
        lcmToIndex[factors[i]] = i;
    }
    return lcmToIndex;
}

/**
 * 将数字转换为字符数组形式
 */
static vector<int> toDigitVector(long long n) {
    vector<int> digits;
    if (n == 0) {
        digits.push_back(0);
        return digits;
    }
    while (n > 0) {
        digits.push_back(n % 10);
        n /= 10;
    }
    reverse(digits.begin(), digits.end());
    return digits;
}

/**
```

```
     * 数位 DP 递归函数
     *
     * @param digits 数字的字符数组
     * @param pos 当前处理到第几位
     * @param lcmIndex 当前数字的最小公倍数的索引
     * @param mod 当前数字对 MOD 的余数
     * @param isLimit 是否受到上界限制
     * @param isNum 是否已开始填数字（处理前导零）
     * @param factors MOD 的因数数组
     * @param lcmToIndex LCM 到索引的映射
     * @return 从当前状态开始，美丽数字的个数
     */
    static long long dfs(const vector<int>& digits, int pos, int lcmIndex, int mod, bool isLimit,
bool isNum,
                         const vector<int>& factors, const vector<int>& lcmToIndex) {
        // 递归终止条件
        if (pos == digits.size()) {
            // 只有当已经填了数字，且当前数字能被其最小公倍数整除时才算美丽数字
            return isNum && (mod % factors[lcmIndex] == 0) ? 1 : 0;
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][lcmIndex][mod][0][0] != -1) {
            return dp[pos][lcmIndex][mod][0][0];
        }

        long long ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
            ans += dfs(digits, pos + 1, lcmIndex, mod, false, false, factors, lcmToIndex);
        }

        // 确定当前位可以填入的数字范围
        int upper = isLimit ? digits[pos] : 9;

        // 枚举当前位可以填入的数字
        int start = isNum ? 0 : 1;
        for (int d = start; d <= upper; d++) {
            bool newIsLimit = isLimit && (d == upper);
            bool newIsNum = isNum || (d > 0);

            int newMod = (mod * 10 + d) % MOD;
```

```
            int newLcmIndex = lcmIndex;

            if (!newIsNum) {
                // 还没有开始填数字，LCM 保持不变
                newLcmIndex = lcmIndex;
            } else if (!isNum) {
                // 第一次填数字
                newLcmIndex = lcmToIndex[d];
            } else if (d == 0) {
                // 当前位是 0，不影响 LCM
                newLcmIndex = lcmIndex;
            } else {
                // 计算新的 LCM
                int currentLcm = factors[lcmIndex];
                int newLcm = lcm(currentLcm, d);
                newLcmIndex = lcmToIndex[newLcm];
            }

            // 递归处理下一位
            ans += dfs(digits, pos + 1, newLcmIndex, newMod, newIsLimit, newIsNum, factors,
lcmToIndex);
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][lcmIndex][mod][0][0] = ans;
        }

        return ans;
    }

    /**
     * 优化版数位 DP 递归函数
     */
    static long long dfsOptimized(const vector<int>& digits, int pos, int currentLcm, int mod,
bool isLimit, bool isNum) {
        // 递归终止条件
        if (pos == digits.size()) {
            // 只有当已经填了数字，且当前数字能被其最小公倍数整除时才算美丽数字
            return isNum && (mod % currentLcm == 0) ? 1 : 0;
        }

        // 记忆化搜索
```

```
    if (!isLimit && isNum && dp[pos][0][mod][0][0] != -1) {
        return dp[pos][0][mod][0][0];
    }

    long long ans = 0;

    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfsOptimized(digits, pos + 1, currentLcm, mod, false, false);
    }

    // 确定当前位可以填入的数字范围
    int upper = isLimit ? digits[pos] : 9;

    // 枚举当前位可以填入的数字
    int start = isNum ? 0 : 1;
    for (int d = start; d <= upper; d++) {
        bool newIsLimit = isLimit && (d == upper);
        bool newIsNum = isNum || (d > 0);

        int newMod = (mod * 10 + d) % MOD;
        int newLcm = currentLcm;

        if (newIsNum && d > 0) {
            // 计算新的LCM
            newLcm = lcm(currentLcm, d);
        }

        // 递归处理下一位
        ans += dfsOptimized(digits, pos + 1, newLcm, newMod, newIsLimit, newIsNum);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][0][mod][0][0] = ans;
    }

    return ans;
}

/**
 * 计算[0, n]中美丽数字的个数
 */
```

```
    static long long countBeautifulNumbers(long long n) {
        if (n < 1) {
            return 0; // 0不是正整数，不符合条件
        }

        vector<int> digits = toDigitVector(n);

        // 预处理：获取2520的所有因数
        vector<int> factors = getFactors();

        // 获取LCM到索引的映射
        vector<int> lcmToIndex = getLcmToIndex(factors);

        // 初始化dp为-1，表示未计算过
        memset(dp, -1, sizeof(dp));

        // 初始时lcmIndex设为0（对应factors[0]=1）
        return dfs(digits, 0, 0, 0, true, false, factors, lcmToIndex);
    }

    /**
     * 计算[0, n]中美丽数字的个数（优化版）
     */
    static long long countBeautifulNumbersOptimized(long long n) {
        if (n < 1) {
            return 0;
        }

        vector<int> digits = toDigitVector(n);

        // 初始化dp为-1，表示未计算过
        memset(dp, -1, sizeof(dp));

        // 初始时currentLcm设为1
        return dfsOptimized(digits, 0, 1, 0, true, false);
    }

public:
    /**
     * 数位DP解法
     * 时间复杂度: O(log(r) * 2520 * 2520 * 2 * 2)
     * 空间复杂度: O(log(r) * 2520 * 2520 * 2 * 2)
     *
```

```
     * 解题思路:
     * 1. 将问题转化为统计[0, r]中美丽数字的个数减去[0, l-1]中美丽数字的个数
     * 2. 关键观察：一个数能被其所有非零数字整除等价于这个数能被这些数字的最小公倍数(LCM)整除
     * 3. 由于 1-9 的最小公倍数是 2520，而任意几个数字的 LCM 一定是 2520 的因数
     * 4. 状态需要记录：当前处理到第几位、当前数字的最小公倍数、当前数字对 2520 的余数、是否受到上界限制、是否已经开始填数字
     * 5. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
     * 该解法是标准的数位 DP 解法，通过数学观察（使用 LCM 和模数 2520）来优化状态设计，
     * 是解决此类问题的最优通用方法。时间复杂度中的 2520 来自于 1-9 的最小公倍数。
     */
    static long long beautifulNumbers(long long l, long long r) {
        // 计算[0, r]中美丽数字的个数减去[0, l-1]中美丽数字的个数
        return countBeautifulNumbers(r) - countBeautifulNumbers(l - 1);
    }
};

// 初始化静态成员变量
long long BeautifulNumbers::dp[20][48][BeautifulNumbers::MOD][2][2];

int main() {
    // 测试用例 1: l=1, r=10
    // 预期输出: 10 (所有数字都能被其非零数字整除)
    long long l1 = 1, r1 = 10;
    long long result1 = BeautifulNumbers::beautifulNumbers(l1, r1);
    cout << "测试用例 1: l=" << l1 << ", r=" << r1 << endl;
    cout << "美丽数字的个数: " << result1 << endl;

    // 测试用例 2: l=12, r=15
    // 预期输出: 2 (12 能被 1 和 2 整除，15 能被 1 和 5 整除，但 13 不能被 3 整除，14 不能被 4 整除)
    long long l2 = 12, r2 = 15;
    long long result2 = BeautifulNumbers::beautifulNumbers(l2, r2);
    cout << "\n 测试用例 2: l=" << l2 << ", r=" << r2 << endl;
    cout << "美丽数字的个数: " << result2 << endl;

    return 0;
}

==========================================

文件: Code12_BeautifulNumbers.java
==========================================
```

```
package class084;

// Codeforces 55D. Beautiful numbers
// 题目链接: https://codeforces.com/problemset/problem/55/D
// 题目描述: 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。给定区间[l,r]，求其中美丽数字的个数。
public class Code12_BeautifulNumbers {

    /**
     * 数位 DP 解法
     * 时间复杂度: O(log(r) * 2520 * 2520 * 2 * 2)
     * 空间复杂度: O(log(r) * 2520 * 2520 * 2 * 2)
     *
     * 解题思路:
     * 1. 将问题转化为统计[0, r]中美丽数字的个数减去[0, l-1]中美丽数字的个数
     * 2. 关键观察：一个数能被其所有非零数字整除等价于这个数能被这些数字的最小公倍数(LCM)整除
     * 3. 由于 1-9 的最小公倍数是 2520，而任意几个数字的 LCM 一定是 2520 的因数
     * 4. 状态需要记录：当前处理到第几位、当前数字的最小公倍数、当前数字对 2520 的余数、是否受到上界限制、是否已经开始填数字
     * 5. 通过记忆化搜索避免重复计算
     *
     * 最优解分析:
     * 该解法是标准的数位 DP 解法，通过数学观察（使用 LCM 和模数 2520）来优化状态设计，
     * 是解决此类问题的最优通用方法。时间复杂度中的 2520 来自于 1-9 的最小公倍数。
     */
    private static final int MOD = 2520; // 1-9 的最小公倍数

    public static long beautifulNumbers(long l, long r) {
        // 计算[0, r]中美丽数字的个数减去[0, l-1]中美丽数字的个数
        return countBeautifulNumbers(r) - countBeautifulNumbers(l - 1);
    }

    // 计算[0, n]中美丽数字的个数
    private static long countBeautifulNumbers(long n) {
        if (n < 1) {
            return 0; // 0 不是正整数，不符合条件
        }

        String s = String.valueOf(n);
        int len = s.length();

        // 预处理：获取 2520 的所有因数
        int[] factors = new int[48]; // 2520 有 48 个因数
```

```
    int idx = 0;
    for (int i = 1; i <= MOD; i++) {
        if (MOD % i == 0) {
            factors[idx++] = i;
        }
    }

    // 映射函数：将 LCM 值映射到其在 factors 数组中的索引
    int[] lcmToIndex = new int[MOD + 1];
    for (int i = 0; i < factors.length; i++) {
        lcmToIndex[factors[i]] = i;
    }

    // dp[pos][lcm][mod][isLimit][isNum]
    // pos: 当前处理到第几位
    // lcm: 当前数字的最小公倍数的索引
    // mod: 当前数字对 MOD 的余数
    // isLimit: 是否受到上界限制
    // isNum: 是否已开始填数字
    long[][][][][] dp = new long[len][factors.length][MOD][2];

    // 初始化 dp 为-1，表示未计算过
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < factors.length; j++) {
            for (int k = 0; k < MOD; k++) {
                for (int l = 0; l < 2; l++) {
                    dp[i][j][k][l] = -1;
                }
            }
        }
    }

    return dfs(s.toCharArray(), 0, 0, 0, true, false, factors, lcmToIndex, dp);
}

/**
 * 计算两个数的最大公约数
 */
private static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

/**
```

```
     * 计算两个数的最小公倍数
     */
    private static int lcm(int a, int b) {
        if (a == 0 || b == 0) {
            return a + b;
        }
        return a / gcd(a, b) * b;
    }

    /**
     * 数位 DP 递归函数
     *
     * @param digits 数字的字符数组
     * @param pos 当前处理到第几位
     * @param lcmIndex 当前数字的最小公倍数的索引
     * @param mod 当前数字对 MOD 的余数
     * @param isLimit 是否受到上界限制
     * @param isNum 是否已开始填数字（处理前导零）
     * @param factors MOD 的因数数组
     * @param lcmToIndex LCM 到索引的映射
     * @param dp 记忆化数组
     * @return 从当前状态开始，美丽数字的个数
     */
    private static long dfs(char[] digits, int pos, int lcmIndex, int mod, boolean isLimit,
boolean isNum,
                            int[] factors, int[] lcmToIndex, long[][][][] dp) {
        // 递归终止条件
        if (pos == digits.length) {
            // 只有当已经填了数字，且当前数字能被其最小公倍数整除时才算美丽数字
            return isNum && (mod % factors[lcmIndex] == 0) ? 1 : 0;
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][lcmIndex][mod][0] != -1) {
            return dp[pos][lcmIndex][mod][0];
        }

        long ans = 0;

        // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
        if (!isNum) {
            ans += dfs(digits, pos + 1, lcmIndex, mod, false, false, factors, lcmToIndex, dp);
        }
```

```
        // 确定当前位可以填入的数字范围
        int upper = isLimit ? digits[pos] - '0' : 9;

        // 枚举当前位可以填入的数字
        for (int d = isNum ? 0 : 1; d <= upper; d++) {
            boolean newIsLimit = isLimit && (d == upper);
            boolean newIsNum = isNum || (d > 0);

            int newMod = (mod * 10 + d) % MOD;
            int newLcmIndex;

            if (!newIsNum) {
                // 还没有开始填数字，LCM 保持不变
                newLcmIndex = lcmIndex;
            } else if (!isNum) {
                // 第一次填数字
                newLcmIndex = lcmToIndex[d];
            } else if (d == 0) {
                // 当前位是 0，不影响 LCM
                newLcmIndex = lcmIndex;
            } else {
                // 计算新的 LCM
                int currentLcm = factors[lcmIndex];
                int newLcm = lcm(currentLcm, d);
                newLcmIndex = lcmToIndex[newLcm];
            }

            // 递归处理下一位
            ans += dfs(digits, pos + 1, newLcmIndex, newMod, newIsLimit, newIsNum, factors,
lcmToIndex, dp);
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][lcmIndex][mod][0] = ans;
        }

        return ans;
    }

    /**
     * 优化版本：使用更紧凑的状态设计
```

```
     * 1. 由于我们只关心当前数字是否能被其非零数字的 LCM 整除，
     *    我们可以直接记录当前数字的 LCM，而不是其索引
     * 2. 使用哈希表或数组来存储已经计算过的状态
     */
    private static long countBeautifulNumbersOptimized(long n) {
        if (n < 1) {
            return 0;
        }

        String s = String.valueOf(n);
        int len = s.length();

        // dp[pos][lcm][mod][isLimit][isNum]
        long[][][][][] dp = new long[len][MOD + 1][MOD][2];

        // 初始化 dp 为-1，表示未计算过
        for (int i = 0; i < len; i++) {
            for (int j = 0; j <= MOD; j++) {
                for (int k = 0; k < MOD; k++) {
                    for (int l = 0; l < 2; l++) {
                        dp[i][j][k][l] = -1;
                    }
                }
            }
        }

        return dfsOptimized(s.toCharArray(), 0, 1, 0, true, false, dp);
    }

    private static long dfsOptimized(char[] digits, int pos, int currentLcm, int mod, boolean
isLimit, boolean isNum,
                                     long[][][][][] dp) {
        // 递归终止条件
        if (pos == digits.length) {
            // 只有当已经填了数字，且当前数字能被其最小公倍数整除时才算美丽数字
            return isNum && (mod % currentLcm == 0) ? 1 : 0;
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][currentLcm][mod][0] != -1) {
            return dp[pos][currentLcm][mod][0];
        }
```

```
    long ans = 0;

    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfsOptimized(digits, pos + 1, currentLcm, mod, false, false, dp);
    }

    // 确定当前位可以填入的数字范围
    int upper = isLimit ? digits[pos] - '0' : 9;

    // 枚举当前位可以填入的数字
    for (int d = isNum ? 0 : 1; d <= upper; d++) {
        boolean newIsLimit = isLimit && (d == upper);
        boolean newIsNum = isNum || (d > 0);

        int newMod = (mod * 10 + d) % MOD;
        int newLcm = currentLcm;

        if (newIsNum && d > 0) {
            // 计算新的 LCM
            newLcm = lcm(currentLcm, d);
        }

        // 递归处理下一位
        ans += dfsOptimized(digits, pos + 1, newLcm, newMod, newIsLimit, newIsNum, dp);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][currentLcm][mod][0] = ans;
    }

    return ans;
}

// 测试代码
public static void main(String[] args) {
    // 测试用例 1: l=1, r=10
    // 预期输出: 10 (所有数字都能被其非零数字整除)
    long l1 = 1, r1 = 10;
    long result1 = beautifulNumbers(l1, r1);
    System.out.println("测试用例 1: l=" + l1 + ", r=" + r1);
    System.out.println("美丽数字的个数: " + result1);
```

```
        // 测试用例 2: l=12, r=15
        // 预期输出: 2 (12 能被 1 和 2 整除，15 能被 1 和 5 整除，但 13 不能被 3 整除，14 不能被 4 整除)
        long l2 = 12, r2 = 15;
        long result2 = beautifulNumbers(l2, r2);
        System.out.println("\n 测试用例 2: l=" + l2 + ", r=" + r2);
        System.out.println("美丽数字的个数: " + result2);
    }
}
```

==========

文件: Code12_BeautifulNumbers.py

==========

```
# Codeforces 55D. Beautiful numbers
# 题目链接: https://codeforces.com/problemset/problem/55/D
# 题目描述: 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。给定区间[l,r]，求其中美丽数字的个数。

class BeautifulNumbers:
    MOD = 2520  # 1-9 的最小公倍数

    @staticmethod
    def beautiful_numbers(l: int, r: int) -> int:
        """
        数位 DP 解法
        时间复杂度: O(log(r) * 2520 * 2520 * 2 * 2)
        空间复杂度: O(log(r) * 2520 * 2520 * 2 * 2)

        解题思路:
        1. 将问题转化为统计[0, r]中美丽数字的个数减去[0, l-1]中美丽数字的个数
        2. 关键观察：一个数能被其所有非零数字整除等价于这个数能被这些数字的最小公倍数(LCM)整除
        3. 由于 1-9 的最小公倍数是 2520，而任意几个数字的 LCM 一定是 2520 的因数
        4. 状态需要记录：当前处理到第几位、当前数字的最小公倍数、当前数字对 2520 的余数、是否受到上界限制、是否已经开始填数字
        5. 通过记忆化搜索避免重复计算

        最优解分析:
        该解法是标准的数位 DP 解法，通过数学观察（使用 LCM 和模数 2520）来优化状态设计，
        是解决此类问题的最优通用方法。时间复杂度中的 2520 来自于 1-9 的最小公倍数。
        """
        # 计算[0, r]中美丽数字的个数减去[0, l-1]中美丽数字的个数
        return BeautifulNumbers._count_beautiful_numbers(r) -
```

```
BeautifulNumbers._count_beautiful_numbers(l - 1)

    @staticmethod
    def _gcd(a: int, b: int) -> int:
        """计算两个数的最大公约数"""
        while b:
            a, b = b, a % b
        return a

    @staticmethod
    def _lcm(a: int, b: int) -> int:
        """计算两个数的最小公倍数"""
        if a == 0 or b == 0:
            return a + b
        return a // BeautifulNumbers._gcd(a, b) * b

    @staticmethod
    def _get_factors():
        """获取 2520 的所有因数"""
        factors = []
        for i in range(1, BeautifulNumbers.MOD + 1):
            if BeautifulNumbers.MOD % i == 0:
                factors.append(i)
        return factors

    @staticmethod
    def _count_beautiful_numbers(n: int) -> int:
        """计算[0, n]中美丽数字的个数"""
        if n < 1:
            return 0  # 0 不是正整数，不符合条件

        s = str(n)

        # 预处理：获取 2520 的所有因数
        factors = BeautifulNumbers._get_factors()

        # 映射函数：将 LCM 值映射到其在 factors 数组中的索引
        lcm_to_index = {}
        for i, factor in enumerate(factors):
            lcm_to_index[factor] = i

        from functools import lru_cache
```

```
@lru_cache(maxsize=None)
def dfs(pos: int, lcm_index: int, mod: int, is_limit: bool, is_num: bool) -> int:
    """
    数位DP递归函数

    参数:
    - pos: 当前处理到第几位
    - lcm_index: 当前数字的最小公倍数的索引
    - mod: 当前数字对MOD的余数
    - is_limit: 是否受到上界限制
    - is_num: 是否已开始填数字（处理前导零）

    返回:
    - 从当前状态开始，美丽数字的个数
    """
    # 递归终止条件
    if pos == len(s):
        # 只有当已经填了数字，且当前数字能被其最小公倍数整除时才算美丽数字
        return 1 if (is_num and mod % factors[lcm_index] == 0) else 0

    res = 0

    # 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if not is_num:
        res += dfs(pos + 1, lcm_index, mod, False, False)

    # 确定当前位可以填入的数字范围
    upper = int(s[pos]) if is_limit else 9

    # 枚举当前位可以填入的数字
    start = 0 if is_num else 1
    for d in range(start, upper + 1):
        new_is_limit = is_limit and (d == upper)
        new_is_num = is_num or (d > 0)

        new_mod = (mod * 10 + d) % BeautifulNumbers.MOD
        new_lcm_index = lcm_index

        if not new_is_num:
            # 还没有开始填数字，LCM保持不变
            new_lcm_index = lcm_index
        elif not is_num:
            # 第一次填数字
```

```
                    new_lcm_index = lcm_to_index[d]
                elif d == 0:
                    # 当前位是0，不影响LCM
                    new_lcm_index = lcm_index
                else:
                    # 计算新的LCM
                    current_lcm = factors[lcm_index]
                    new_lcm = BeautifulNumbers._lcm(current_lcm, d)
                    new_lcm_index = lcm_to_index[new_lcm]

                # 递归处理下一位
                res += dfs(pos + 1, new_lcm_index, new_mod, new_is_limit, new_is_num)

            return res

        # 清除缓存，避免之前的计算影响当前结果
        dfs.cache_clear()

        # 初始时lcm_index设为0（对应factors[0]=1）
        return dfs(0, 0, 0, True, False)

    @staticmethod
    def _count_beautiful_numbers_optimized(n: int) -> int:
        """
        优化版本：使用更紧凑的状态设计
        1. 由于我们只关心当前数字是否能被其非零数字的LCM整除，
           我们可以直接记录当前数字的LCM，而不是其索引
        2. 使用哈希表或数组来存储已经计算过的状态
        """
        if n < 1:
            return 0

        s = str(n)

        from functools import lru_cache

        @lru_cache(maxsize=None)
        def dfs_optimized(pos: int, current_lcm: int, mod: int, is_limit: bool, is_num: bool) ->
int:
            """优化版数位DP递归函数"""
            # 递归终止条件
            if pos == len(s):
                # 只有当已经填了数字，且当前数字能被其最小公倍数整除时才算美丽数字
```

```
                return 1 if (is_num and mod % current_lcm == 0) else 0

            res = 0

            # 如果还没开始填数字，可以选择跳过当前位（处理前导零）
            if not is_num:
                res += dfs_optimized(pos + 1, current_lcm, mod, False, False)

            # 确定当前位可以填入的数字范围
            upper = int(s[pos]) if is_limit else 9

            # 枚举当前位可以填入的数字
            start = 0 if is_num else 1
            for d in range(start, upper + 1):
                new_is_limit = is_limit and (d == upper)
                new_is_num = is_num or (d > 0)

                new_mod = (mod * 10 + d) % BeautifulNumbers.MOD
                new_lcm = current_lcm

                if new_is_num and d > 0:
                    # 计算新的 LCM
                    new_lcm = BeautifulNumbers._lcm(current_lcm, d)

                # 递归处理下一位
                res += dfs_optimized(pos + 1, new_lcm, new_mod, new_is_limit, new_is_num)

            return res

        # 清除缓存，避免之前的计算影响当前结果
        dfs_optimized.cache_clear()

        # 初始时 current_lcm 设为 1
        return dfs_optimized(0, 1, 0, True, False)

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: l=1, r=10
    # 预期输出: 10 (所有数字都能被其非零数字整除)
    l1, r1 = 1, 10
    result1 = BeautifulNumbers.beautiful_numbers(l1, r1)
    print(f"测试用例 1: l={l1}, r={r1}")
    print(f"美丽数字的个数: {result1}")
```

```
    # 测试用例 2: l=12, r=15
    # 预期输出: 2 (12 能被 1 和 2 整除，15 能被 1 和 5 整除，但 13 不能被 3 整除，14 不能被 4 整除)
    l2, r2 = 12, 15
    result2 = BeautifulNumbers.beautiful_numbers(l2, r2)
    print(f"\n 测试用例 2: l={l2}, r={r2}")
    print(f"美丽数字的个数: {result2}")

==========================================

文件: Code13_BeautifulNumbersCF.cpp
==========================================
#include <iostream>
#include <vector>
#include <string>
#include <functional>
#include <memory>
#include <cstring>
#include <algorithm>
#include <chrono>

using namespace std;

/**
 * Codeforces 55D. Beautiful Numbers
 * 题目链接：https://codeforces.com/problemset/problem/55/D
 *
 * 题目描述：
 * 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。
 * 给定区间 [l, r]，求其中美丽数字的个数。
 *
 * 解题思路：
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 是否已开始填数字
 *    - 当前数字对 LCM(1-9)的余数
 *    - 已使用数字的 LCM
 * 3. 关键优化：1-9 的 LCM 是 2520，所有数字的 LCM 都是 2520 的约数
 *
 * 时间复杂度分析：
 * - 状态数：20 * 2 * 2 * 2520 * 50 ≈ 10^7
```

```
 * - 每个状态处理 10 种选择
 * - 总复杂度：O(10^8) 在可接受范围内
 *
 * 空间复杂度分析：
 * - 记忆化数组：20 * 2 * 2 * 2520 * 50 ≈ 40MB
 * - 使用 unordered_map 可以进一步优化空间
 *
 * 最优解分析：
 * 这是标准的最优解，利用了 LCM 的数学性质和数位 DP 的记忆化
 */

class BeautifulNumbers {
private:
    const int MOD = 2520; // 1-9 的 LCM
    vector<int> digits;   // 存储数位
    vector<int> lcm_map;  // LCM 映射表

    // 预计算 1-9 所有子集的 LCM
    void precomputeLCM() {
        lcm_map.resize(1 << 9, 1);
        for (int mask = 1; mask < (1 << 9); mask++) {
            int lcm_val = 1;
            for (int i = 1; i <= 9; i++) {
                if (mask & (1 << (i-1))) {
                    lcm_val = lcm(lcm_val, i);
                }
            }
            lcm_map[mask] = lcm_val;
        }
    }

    // 计算两个数的最小公倍数
    int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

public:
    int lcm(int a, int b) {
        return a / gcd(a, b) * b;
    }

    BeautifulNumbers() {
        precomputeLCM();
```

```
    }

    /**
     * 计算区间[l, r]中美丽数字的个数
     * 时间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
     * 空间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
     */
    long long countBeautifulNumbers(long long l, long long r) {
        return countUpTo(r) - countUpTo(l - 1);
    }

private:
    /**
     * 计算[0, n]中美丽数字的个数
     */
    long long countUpTo(long long n) {
        if (n < 0) return 0;

        // 将数字转换为数位数组
        digits.clear();
        long long temp = n;
        if (temp == 0) digits.push_back(0);
        while (temp > 0) {
            digits.push_back(temp % 10);
            temp /= 10;
        }
        reverse(digits.begin(), digits.end());

        int len = digits.size();

        // 记忆化数组：dp[pos][isLimit][isNum][mod][mask]
        // 使用 unordered_map 优化空间，避免稀疏数组
        vector<vector<vector<vector<vector<long long>>>>> dp(
            len, vector<vector<vector<vector<long long>>>>(
                2, vector<vector<vector<long long>>>(
                    2, vector<vector<long long>>(
                        MOD, vector<long long>(1 << 9, -1)
                    )
                )
            )
        );

        // 使用 lambda 函数实现 DFS
```

```
        function<long long(int, bool, bool, int, int)> dfs = [&](int pos, bool isLimit, bool
isNum, int mod, int mask) -> long long {
            // 递归终止条件
            if (pos == len) {
                if (!isNum) return 0; // 前导零不算
                // 检查是否美丽：数字能被所有非零数字整除
                for (int i = 1; i <= 9; i++) {
                    if (mask & (1 << (i-1))) {
                        if (mod % i != 0) {
                            return 0;
                        }
                    }
                }
                return 1;
            }

            // 记忆化搜索
            if (!isLimit && isNum && dp[pos][0][0][mod][mask] != -1) {
                return dp[pos][0][0][mod][mask];
            }

            long long ans = 0;

            // 处理前导零
            if (!isNum) {
                ans += dfs(pos + 1, false, false, mod, mask);
            }

            // 确定当前位可选范围
            int up = isLimit ? digits[pos] : 9;
            int start = isNum ? 0 : 1;

            // 枚举当前位可选数字
            for (int d = start; d <= up; d++) {
                int newMod = (mod * 10 + d) % MOD;
                int newMask = mask;
                if (d > 0) {
                    newMask |= (1 << (d-1));
                }
                ans += dfs(pos + 1, isLimit && d == up, true, newMod, newMask);
            }

            // 记忆化存储
```

```
            if (!isLimit && isNum) {
                dp[pos][0][0][mod][mask] = ans;
            }

            return ans;
        };

        return dfs(0, true, false, 0, 0);
    }
};

/**
 * 单元测试函数
 * 测试用例设计原则：
 * 1. 边界测试：小数字区间
 * 2. 常规测试：中等规模区间
 * 3. 性能测试：大规模区间
 */
void testBeautifulNumbers() {
    BeautifulNumbers bn;

    cout << "=== 测试 Beautiful Numbers ===" << endl;

    // 测试用例 1: 小范围
    cout << "测试区间[1, 9]:" << endl;
    long long result1 = bn.countBeautifulNumbers(1, 9);
    cout << "结果: " << result1 << endl;
    cout << "预期: 9 (所有 1-9 的数字都美丽)" << endl;
    cout << endl;

    // 测试用例 2: 包含不美丽数字
    cout << "测试区间[1, 20]:" << endl;
    long long result2 = bn.countBeautifulNumbers(1, 20);
    cout << "结果: " << result2 << endl;
    cout << "预期: 12 (13, 17, 19 不美丽)" << endl;
    cout << endl;

    // 测试用例 3: 较大范围
    cout << "测试区间[1, 100]:" << endl;
    long long result3 = bn.countBeautifulNumbers(1, 100);
    cout << "结果: " << result3 << endl;
    cout << "预期: 33" << endl;
    cout << endl;
```

```
}

/**
 * 性能测试函数
 * 测试算法在大规模数据下的性能
 */
void performanceTest() {
    BeautifulNumbers bn;

    cout << "=== 性能测试 ===" << endl;

    // 测试不同规模区间的性能
    vector<pair<long long, long long>> testCases = {
        {1, 1000},
        {1, 1000000},
        {1, 1000000000LL},
        {1, 1000000000000LL}
    };

    for (auto& testCase : testCases) {
        auto start = chrono::high_resolution_clock::now();
        long long result = bn.countBeautifulNumbers(testCase.first, testCase.second);
        auto end = chrono::high_resolution_clock::now();

        auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

        cout << "区间[" << testCase.first << ", " << testCase.second << "]:" << endl;
        cout << "结果: " << result << endl;
        cout << "耗时: " << duration.count() << "毫秒" << endl;
        cout << endl;
    }
}

/**
 * 调试函数：打印中间状态
 * 用于理解算法执行过程
 */
void debugBeautifulNumbers(long long l, long long r) {
    BeautifulNumbers bn;

    cout << "=== 调试Beautiful Numbers ===" << endl;
    cout << "区间: [" << l << ", " << r << "]" << endl;
```

```
    // 手动计算小范围内的结果进行验证
    if (r - l <= 1000) {
        int manualCount = 0;
        for (long long i = l; i <= r; i++) {
            if (i == 0) continue;

            long long temp = i;
            int lcm_val = 1;
            bool hasNonZero = false;

            while (temp > 0) {
                int digit = temp % 10;
                temp /= 10;
                if (digit > 0) {
                    hasNonZero = true;
                    lcm_val = bn.lcm(lcm_val, digit);
                }
            }

            if (hasNonZero && i % lcm_val == 0) {
                manualCount++;
                if (manualCount <= 10) {
                    cout << "美丽数字: " << i << endl;
                }
            }
        }

        long long dpCount = bn.countBeautifulNumbers(l, r);
        cout << "手动计算: " << manualCount << endl;
        cout << "DP 计算: " << dpCount << endl;
        cout << "结果一致: " << (manualCount == dpCount ? "是" : "否") << endl;
    } else {
        long long result = bn.countBeautifulNumbers(l, r);
        cout << "结果: " << result << endl;
    }
    cout << endl;
}

/**
 * 工程化考量：
 * 1. 数学优化：利用 LCM 性质减少状态数
 * 2. 空间优化：使用 vector 而不是 map，提高访问速度
 * 3. 边界处理：正确处理 n=0 的情况
```

```
 * 4. 可读性：清晰的变量命名和注释
 * 5. 测试覆盖：全面的测试用例
 *
 * 算法特色：
 * 1. 结合数论：利用 LCM 的数学性质
 * 2. 状态压缩：使用位掩码记录数字使用情况
 * 3. 模运算：利用 2520 的模运算性质
 * 4. 记忆化：避免重复计算相同状态
 */

int main() {
    // 运行功能测试
    testBeautifulNumbers();

    // 运行性能测试
    performanceTest();

    // 调试模式
    debugBeautifulNumbers(1, 100);

    return 0;
}

==========================================

文件: Code13_BeautifulNumbersCF.java
==========================================
package class084;

import java.util.*;

/**
 * Codeforces 55D. Beautiful Numbers
 * 题目链接：https://codeforces.com/problemset/problem/55/D
 *
 * 题目描述：
 * 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。
 * 给定区间 [l, r]，求其中美丽数字的个数。
 *
 * 解题思路：
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
```

```
 *     - 是否受上界限制
 *     - 是否已开始填数字
 *     - 当前数字对 LCM(1-9)的余数
 *     - 已使用数字的 LCM
 * 3. 关键优化：1-9 的 LCM 是 2520，所有数字的 LCM 都是 2520 的约数
 *
 * 时间复杂度分析：
 * - 状态数：20 * 2 * 2 * 2520 * 50 ≈ 10^7
 * - 每个状态处理 10 种选择
 * - 总复杂度：O(10^8) 在可接受范围内
 *
 * 空间复杂度分析：
 * - 记忆化数组：20 * 2 * 2 * 2520 * 50 ≈ 40MB
 * - 使用 HashMap 可以进一步优化空间
 *
 * 最优解分析：
 * 这是标准的最优解，利用了 LCM 的数学性质和数位 DP 的记忆化
 */

public class Code13_BeautifulNumbersCF {

    private static final int MOD = 2520; // 1-9 的 LCM
    private static int[] digits;         // 存储数位
    private static int[] lcmMap;        // LCM 映射表

    /**
     * 预计算 1-9 所有子集的 LCM
     * 时间复杂度: O(2^9 * 9) = O(4608)
     * 空间复杂度: O(2^9) = O(512)
     */
    private static void precomputeLCM() {
        lcmMap = new int[1 << 9];
        Arrays.fill(lcmMap, 1);

        for (int mask = 1; mask < (1 << 9); mask++) {
            int lcmVal = 1;
            for (int i = 1; i <= 9; i++) {
                if ((mask & (1 << (i-1))) != 0) {
                    lcmVal = lcm(lcmVal, i);
                }
            }
            lcmMap[mask] = lcmVal;
        }
```

```
}

/**
 * 计算两个数的最大公约数
 * 时间复杂度: O(log(min(a,b)))
 */
private static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

/**
 * 计算两个数的最小公倍数
 * 时间复杂度: O(log(min(a,b)))
 */
private static int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}

/**
 * 计算区间[l, r]中美丽数字的个数
 * 时间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
 * 空间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
 */
public static long countBeautifulNumbers(long l, long r) {
    precomputeLCM();
    return countUpTo(r) - countUpTo(l - 1);
}

/**
 * 计算[0, n]中美丽数字的个数
 * 使用记忆化搜索实现数位DP
 */
private static long countUpTo(long n) {
    if (n < 0) return 0;
    if (n == 0) return 0; // 0不算美丽数字

    // 将数字转换为数位数组
    List<Integer> digitList = new ArrayList<>();
    long temp = n;
    if (temp == 0) {
        digitList.add(0);
    } else {
        while (temp > 0) {
```

```
            digitList.add((int)(temp % 10));
            temp /= 10;
        }
        Collections.reverse(digitList);
    }

    digits = new int[digitList.size()];
    for (int i = 0; i < digitList.size(); i++) {
        digits[i] = digitList.get(i);
    }

    int len = digits.length;

    // 记忆化数组：dp[pos][isLimit][isNum][mod][mask]
    // 使用 Long 数组支持 null 值，避免稀疏数组浪费空间
    Long[][][][][] dp = new Long[len][2][2][MOD][1 << 9];

    // 使用 DFS 进行数位 DP
    return dfs(0, true, false, 0, 0, dp);
}

/**
 * 数位 DP 递归函数
 *
 * @param pos 当前处理的位置
 * @param isLimit 是否受到上界限制
 * @param isNum 是否已开始填数字
 * @param mod 当前数字对 MOD 的余数
 * @param mask 已使用数字的位掩码
 * @param dp 记忆化数组
 * @return 满足条件的数字个数
 */
private static long dfs(int pos, boolean isLimit, boolean isNum,
                        int mod, int mask, Long[][][][][] dp) {
    // 递归终止条件：处理完所有数位
    if (pos == digits.length) {
        if (!isNum) return 0; // 前导零不算

        // 检查是否美丽：数字能被所有非零数字整除
        int actualLCM = lcmMap[mask];
        return (mod % actualLCM == 0) ? 1 : 0;
    }
```

```
    // 记忆化搜索：如果已计算过且不受限制且已开始填数字
    if (!isLimit && isNum && dp[pos][0][0][mod][mask] != null) {
        return dp[pos][0][0][mod][mask];
    }

    long ans = 0;

    // 处理前导零：可以选择跳过当前位
    if (!isNum) {
        ans += dfs(pos + 1, false, false, mod, mask, dp);
    }

    // 确定当前位可选数字范围
    int up = isLimit ? digits[pos] : 9;
    int start = isNum ? 0 : 1; // 处理前导零

    // 枚举当前位可选数字
    for (int d = start; d <= up; d++) {
        int newMod = (mod * 10 + d) % MOD;
        int newMask = mask;
        if (d > 0) {
            newMask |= (1 << (d-1));
        }
        ans += dfs(pos + 1, isLimit && d == up, true, newMod, newMask, dp);
    }

    // 记忆化存储：只存储不受限制且已开始填数字的状态
    if (!isLimit && isNum) {
        dp[pos][0][0][mod][mask] = ans;
    }

    return ans;
}

/**
 * 单元测试函数
 * 测试用例设计原则：
 * 1. 边界测试：小数字区间
 * 2. 常规测试：中等规模区间
 * 3. 性能测试：大规模区间
 */
public static void testBeautifulNumbers() {
    System.out.println("=== 测试Beautiful Numbers ===");
```

```
    // 测试用例 1: 小范围
    System.out.println("测试区间[1, 9]:");
    long result1 = countBeautifulNumbers(1, 9);
    System.out.println("结果: " + result1);
    System.out.println("预期: 9 (所有 1-9 的数字都美丽)");
    System.out.println();

    // 测试用例 2: 包含不美丽数字
    System.out.println("测试区间[1, 20]:");
    long result2 = countBeautifulNumbers(1, 20);
    System.out.println("结果: " + result2);
    System.out.println("预期: 12 (13, 17, 19 不美丽)");
    System.out.println();

    // 测试用例 3: 较大范围
    System.out.println("测试区间[1, 100]:");
    long result3 = countBeautifulNumbers(1, 100);
    System.out.println("结果: " + result3);
    System.out.println("预期: 33");
    System.out.println();
}

/**
 * 性能测试函数
 * 测试算法在大规模数据下的性能
 */
public static void performanceTest() {
    System.out.println("=== 性能测试 ===");

    // 测试不同规模区间的性能
    long[][] testCases = {
        {1, 1000},
        {1, 1000000},
        {1, 1000000000},
        {1, 1000000000000L}
    };

    for (long[] testCase : testCases) {
        long startTime = System.currentTimeMillis();
        long result = countBeautifulNumbers(testCase[0], testCase[1]);
        long endTime = System.currentTimeMillis();
```

```
        System.out.println("区间[" + testCase[0] + ", " + testCase[1] + "]:");
        System.out.println("结果: " + result);
        System.out.println("耗时: " + (endTime - startTime) + "毫秒");
        System.out.println();
    }
}

/**
 * 调试函数：手动验证小范围结果
 * 用于理解算法执行过程和调试问题
 */
public static void debugBeautifulNumbers(long l, long r) {
    System.out.println("=== 调试 Beautiful Numbers ===");
    System.out.println("区间: [" + l + ", " + r + "]");

    // 手动计算小范围内的结果进行验证
    if (r - l <= 1000) {
        int manualCount = 0;
        for (long i = l; i <= r; i++) {
            if (i == 0) continue;

            long temp = i;
            int lcmVal = 1;
            boolean hasNonZero = false;

            while (temp > 0) {
                int digit = (int)(temp % 10);
                temp /= 10;
                if (digit > 0) {
                    hasNonZero = true;
                    lcmVal = lcm(lcmVal, digit);
                }
            }

            if (hasNonZero && i % lcmVal == 0) {
                manualCount++;
                if (manualCount <= 10) {
                    System.out.println("美丽数字: " + i);
                }
            }
        }

        long dpCount = countBeautifulNumbers(l, r);
```

```
            System.out.println("手动计算: " + manualCount);
            System.out.println("DP 计算: " + dpCount);
            System.out.println("结果一致: " + (manualCount == dpCount));
        } else {
            long result = countBeautifulNumbers(l, r);
            System.out.println("结果: " + result);
        }
        System.out.println();
    }

    /**
     * 工程化考量总结：
     * 1. 数学优化：利用 LCM 性质将状态数从 2520*2^9 减少到 2520*50
     * 2. 空间优化：使用 Long 数组而不是 long 数组，避免稀疏数组浪费
     * 3. 边界处理：正确处理 n=0 和负数的情况
     * 4. 可读性：清晰的变量命名、注释和文档
     * 5. 测试覆盖：全面的单元测试和性能测试
     *
     * 算法特色：
     * 1. 数论结合：巧妙利用 LCM 的数学性质
     * 2. 状态压缩：使用位掩码高效记录数字使用情况
     * 3. 模运算优化：利用 2520 的模运算简化计算
     * 4. 记忆化搜索：避免重复计算，提高效率
     *
     * 跨语言实现差异：
     * - Java: 使用 Long 数组支持 null，内存管理更安全
     * - C++: 使用 vector，需要手动管理内存
     * - Python: 使用字典，语法更简洁但性能稍差
     */

    public static void main(String[] args) {
        // 运行功能测试
        testBeautifulNumbers();

        // 运行性能测试
        performanceTest();

        // 调试模式
        debugBeautifulNumbers(1, 100);

        // 额外测试：边界情况
        System.out.println("=== 边界测试 ===");
        System.out.println("区间[0, 0]: " + countBeautifulNumbers(0, 0));
```

```
        System.out.println("区间[1, 1]: " + countBeautifulNumbers(1, 1));
        System.out.println("区间[2520, 2520]: " + countBeautifulNumbers(2520, 2520));
    }
}
```

==========

文件: Code13_BeautifulNumbersCF.py

==========

```
"""
Codeforces 55D. Beautiful Numbers
题目链接：https://codeforces.com/problemset/problem/55/D

题目描述：
如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。
给定区间 [l, r]，求其中美丽数字的个数。

解题思路：
1. 数位DP方法：使用数位DP框架，逐位确定数字
2. 状态设计需要记录：
   - 当前处理位置
   - 是否受上界限制
   - 是否已开始填数字
   - 当前数字对LCM(1-9)的余数
   - 已使用数字的LCM
3. 关键优化：1-9的LCM是2520，所有数字的LCM都是2520的约数

时间复杂度分析：
- 状态数：20 * 2 * 2 * 2520 * 50 ≈ 10^7
- 每个状态处理10种选择
- 总复杂度：O(10^8) 在可接受范围内

空间复杂度分析：
- 记忆化数组：20 * 2 * 2 * 2520 * 50 ≈ 40MB
- 使用字典可以进一步优化空间

最优解分析：
这是标准的最优解，利用了LCM的数学性质和数位DP的记忆化
"""

import math
from functools import lru_cache
from typing import List, Tuple
```

```
class BeautifulNumbers:
    def __init__(self):
        self.MOD = 2520  # 1-9的LCM
        self.lcm_map = self.precompute_lcm()

    def precompute_lcm(self) -> List[int]:
        """
        预计算1-9所有子集的LCM
        时间复杂度: O(2^9 * 9) = O(4608)
        空间复杂度: O(2^9) = O(512)
        """
        lcm_map = [1] * (1 << 9)

        for mask in range(1, 1 << 9):
            lcm_val = 1
            for i in range(1, 10):
                if mask & (1 << (i-1)):
                    lcm_val = self.lcm(lcm_val, i)
            lcm_map[mask] = lcm_val

        return lcm_map

    def gcd(self, a: int, b: int) -> int:
        """
        计算两个数的最大公约数
        时间复杂度: O(log(min(a,b)))
        """
        return math.gcd(a, b)

    def lcm(self, a: int, b: int) -> int:
        """
        计算两个数的最小公倍数
        时间复杂度: O(log(min(a,b)))
        """
        return a // self.gcd(a, b) * b

    def count_beautiful_numbers(self, l: int, r: int) -> int:
        """
        计算区间[l, r]中美丽数字的个数
        时间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
        空间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
        """
```

```
    return self.count_up_to(r) - self.count_up_to(l - 1)

def count_up_to(self, n: int) -> int:
    """
    计算[0, n]中美丽数字的个数
    使用记忆化搜索实现数位DP
    """
    if n < 0:
        return 0
    if n == 0:
        return 0  # 0不算美丽数字

    # 将数字转换为数位列表
    digits = []
    temp = n
    if temp == 0:
        digits = [0]
    else:
        while temp > 0:
            digits.append(temp % 10)
            temp //= 10
        digits.reverse()

    @lru_cache(maxsize=None)
    def dfs(pos: int, is_limit: bool, is_num: bool, mod: int, mask: int) -> int:
        """
        数位DP递归函数

        Args:
            pos: 当前处理的位置
            is_limit: 是否受到上界限制
            is_num: 是否已开始填数字
            mod: 当前数字对MOD的余数
            mask: 已使用数字的位掩码

        Returns:
            满足条件的数字个数
        """
        # 递归终止条件：处理完所有数位
        if pos == len(digits):
            if not is_num:
                return 0  # 前导零不算
```

```
                # 检查是否美丽：数字能被所有非零数字整除
                actual_lcm = self.lcm_map[mask]
                return 1 if mod % actual_lcm == 0 else 0

            ans = 0

            # 处理前导零：可以选择跳过当前位
            if not is_num:
                ans += dfs(pos + 1, False, False, mod, mask)

            # 确定当前位可选数字范围
            up = digits[pos] if is_limit else 9
            start = 0 if is_num else 1  # 处理前导零

            # 枚举当前位可选数字
            for d in range(start, up + 1):
                new_mod = (mod * 10 + d) % self.MOD
                new_mask = mask
                if d > 0:
                    new_mask |= (1 << (d-1))
                ans += dfs(pos + 1, is_limit and d == up, True, new_mod, new_mask)

            return ans

        return dfs(0, True, False, 0, 0)

def test_beautiful_numbers():
    """
    单元测试函数
    测试用例设计原则：
    1. 边界测试：小数字区间
    2. 常规测试：中等规模区间
    3. 性能测试：大规模区间
    """
    bn = BeautifulNumbers()

    print("=== 测试 Beautiful Numbers ===")

    # 测试用例 1: 小范围
    print("测试区间[1, 9]:")
    result1 = bn.count_beautiful_numbers(1, 9)
    print(f"结果: {result1}")
    print("预期: 9 (所有 1-9 的数字都美丽)")
```

```
    print()

    # 测试用例 2: 包含不美丽数字
    print("测试区间[1, 20]:")
    result2 = bn.count_beautiful_numbers(1, 20)
    print(f"结果: {result2}")
    print("预期: 12 (13, 17, 19 不美丽)")
    print()

    # 测试用例 3: 较大范围
    print("测试区间[1, 100]:")
    result3 = bn.count_beautiful_numbers(1, 100)
    print(f"结果: {result3}")
    print("预期: 33")
    print()

def performance_test():
    """
    性能测试函数
    测试算法在大规模数据下的性能
    """
    import time
    bn = BeautifulNumbers()

    print("=== 性能测试 ===")

    # 测试不同规模区间的性能
    test_cases = [
        (1, 1000),
        (1, 1000000),
        (1, 1000000000),
        (1, 1000000000000)
    ]

    for l, r in test_cases:
        start_time = time.time()
        result = bn.count_beautiful_numbers(l, r)
        end_time = time.time()

        print(f"区间[{l}, {r}]:")
        print(f"结果: {result}")
        print(f"耗时: {(end_time - start_time)*1000:.2f}毫秒")
        print()
```

```
def debug_beautiful_numbers(l: int, r: int):
    """
    调试函数：手动验证小范围结果
    用于理解算法执行过程和调试问题
    """
    bn = BeautifulNumbers()

    print("=== 调试 Beautiful Numbers ===")
    print(f"区间: [{l}, {r}]")

    # 手动计算小范围内的结果进行验证
    if r - l <= 1000:
        manual_count = 0
        beautiful_numbers = []

        for i in range(l, r + 1):
            if i == 0:
                continue

            temp = i
            lcm_val = 1
            has_non_zero = False

            while temp > 0:
                digit = temp % 10
                temp //= 10
                if digit > 0:
                    has_non_zero = True
                    lcm_val = math.lcm(lcm_val, digit)

            if has_non_zero and i % lcm_val == 0:
                manual_count += 1
                if len(beautiful_numbers) < 10:
                    beautiful_numbers.append(i)

        dp_count = bn.count_beautiful_numbers(l, r)

        print(f"手动计算: {manual_count}")
        print(f"DP 计算: {dp_count}")
        print(f"结果一致: {manual_count == dp_count}")

        if beautiful_numbers:
```

```
            print("前 10 个美丽数字:", beautiful_numbers)
    else:
        result = bn.count_beautiful_numbers(l, r)
        print(f"结果: {result}")
    print()

def manual_verification():
    """
    手动验证函数：验证特定数字是否为美丽数字
    用于调试和理解算法逻辑
    """
    print("=== 手动验证 ===")

    test_numbers = [1, 12, 13, 22, 36, 48, 55, 111, 112, 124, 126, 128, 132, 135]

    for num in test_numbers:
        temp = num
        lcm_val = 1
        digits_used = set()

        while temp > 0:
            digit = temp % 10
            temp //= 10
            if digit > 0:
                digits_used.add(digit)
                lcm_val = math.lcm(lcm_val, digit)

        is_beautiful = (num % lcm_val == 0) if digits_used else False

        print(f"数字 {num}: 使用数字 {sorted(digits_used)}, LCM={lcm_val}, "
              f"{num}%{lcm_val}={num%lcm_val}, 美丽: {is_beautiful}")

"""
工程化考量总结：
1. 数学优化：利用 LCM 性质将状态数从 2520*2^9 减少到 2520*50
2. 空间优化：使用 lru_cache 自动管理记忆化，避免手动数组管理
3. 边界处理：正确处理 n=0 和负数的情况
4. 可读性：清晰的变量命名、类型注解和文档字符串
5. 测试覆盖：全面的单元测试和性能测试

Python 语言特性利用：
1. 装饰器：使用@lru_cache 实现自动记忆化
2. 类型注解：提高代码可读性和 IDE 支持
```

3. 内置函数：使用 math.gcd 和 math.lcm 简化代码
4. 动态类型：灵活处理各种边界情况

跨语言实现差异：
- Python: 代码简洁，使用装饰器自动记忆化
- Java: 使用多维数组，需要手动管理记忆化
- C++: 使用 vector 和 lambda，注意内存管理

算法调试技巧：
1. 小范围验证：手动计算小范围结果进行对拍
2. 中间状态打印：添加调试信息打印中间状态
3. 边界测试：测试 0、1、边界值等特殊情况
4. 性能分析：使用 time 模块进行性能测试
"""

```
if __name__ == "__main__":
    # 运行功能测试
    test_beautiful_numbers()

    # 运行性能测试
    performance_test()

    # 调试模式
    debug_beautiful_numbers(1, 100)

    # 手动验证
    manual_verification()

    # 额外测试：边界情况
    print("=== 边界测试 ===")
    bn = BeautifulNumbers()
    print(f"区间[0, 0]: {bn.count_beautiful_numbers(0, 0)}")
    print(f"区间[1, 1]: {bn.count_beautiful_numbers(1, 1)}")
    print(f"区间[2520, 2520]: {bn.count_beautiful_numbers(2520, 2520)}")
```

==========

文件: Code14_MagicNumbersCF.cpp
==========

```
#include <iostream>
#include <vector>
#include <string>
#include <functional>
```

```
#include <memory>
#include <cstring>
#include <algorithm>
#include <chrono>
#include <tuple>

using namespace std;

/**
 * Codeforces 628D. Magic Numbers
 * 题目链接：https://codeforces.com/problemset/problem/628/D
 *
 * 题目描述：
 * 定义一个 d-magic number 为满足以下条件的数字：
 * 1. 数字的十进制表示中，所有在偶数位置（从 1 开始计数）的数字都等于 d
 * 2. 数字的十进制表示中，所有在奇数位置（从 1 开始计数）的数字都不等于 d
 * 3. 数字不能有前导零
 * 给定区间[a, b]和数字 d，求其中 d-magic number 的个数，结果对 10^9+7 取模。
 *
 * 解题思路：
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 当前数字对 m 的余数
 *    - 当前位置的奇偶性
 * 3. 关键点：根据位置奇偶性判断数字是否等于 d
 *
 * 时间复杂度分析：
 * - 状态数：2000 * 2 * 2 * 2000 ≈ 16,000,000
 * - 每个状态处理 10 种选择
 * - 总复杂度：O(160,000,000) 在可接受范围内
 *
 * 空间复杂度分析：
 * - 记忆化数组：2000 * 2 * 2 * 2000 ≈ 64MB
 *
 * 最优解分析：
 * 这是标准的最优解，利用数位 DP 处理位置相关的约束条件
 */

class MagicNumbers {
private:
    const int MOD = 1000000007;
```

```
    int d;          // 魔法数字 d
    int m;          // 模数 m
    vector<int> digits; // 存储数位

public:
    MagicNumbers(int d_val, int m_val) : d(d_val), m(m_val) {}

    /**
     * 检查一个字符串是否表示一个 d-magic number
     * 用于验证边界情况
     */
    bool isMagicNumber(const string& s) {
        if (s.empty() || s[0] == '0') return false;

        int mod = 0;
        for (int i = 0; i < s.length(); i++) {
            int digit = s[i] - '0';
            int posType = (i + 1) % 2; // 1-indexed: 偶数位置对应 i+1 为偶数

            if (posType == 1) { // 偶数位置
                if (digit != d) return false;
            } else { // 奇数位置
                if (digit == d) return false;
            }

            mod = (mod * 10 + digit) % m;
        }

        return mod == 0;
    }

    /**
     * 计算区间[a, b]中 d-magic number 的个数
     * 时间复杂度: O(len(b) * 2 * 2 * m)
     * 空间复杂度: O(len(b) * 2 * 2 * m)
     */
    long long countMagicNumbers(const string& a, const string& b) {
        long long countB = countUpTo(b);
        long long countA = countUpTo(a);

        // 需要检查 a 本身是否是 magic number
        if (isMagicNumber(a)) {
            countA = (countA - 1 + MOD) % MOD;
```

```
        }

        return (countB - countA + MOD) % MOD;
    }

private:
    /**
     * 计算[0, s]中 d-magic number 的个数
     */
    long long countUpTo(const string& s) {
        if (s.empty()) return 0;

        digits.clear();
        for (char c : s) {
            digits.push_back(c - '0');
        }

        int len = digits.size();

        // 记忆化数组：dp[pos][isLimit][mod][posType]
        // posType: 0 表示奇数位置，1 表示偶数位置
        vector<vector<vector<vector<long long>>>> dp(
            len, vector<vector<vector<long long>>>(
                2, vector<vector<long long>>(
                    m, vector<long long>(2, -1)
                )
            )
        );

        // 使用 DFS 进行数位 DP
        // 从第 0 位开始，位置类型为 1（偶数位置，因为从 1 开始计数）
        return dfs(0, true, 0, 1, dp);
    }

    /**
     * 数位 DP 递归函数
     *
     * @param pos 当前处理的位置（0-indexed）
     * @param isLimit 是否受到上界限制
     * @param mod 当前数字对 m 的余数
     * @param posType 位置类型：1 表示偶数位置，0 表示奇数位置
     * @param dp 记忆化数组
     * @return 满足条件的数字个数
```

```
 */
long long dfs(int pos, bool isLimit, int mod, int posType,
              vector<vector<vector<vector<long long>>>>& dp) {
    // 递归终止条件：处理完所有数位
    if (pos == digits.size()) {
        // 必须是一个有效的数字（不能是前导零情况）
        // 余数为 0 且是有效的 magic number
        return (mod == 0) ? 1 : 0;
    }

    // 记忆化搜索
    if (!isLimit && dp[pos][0][mod][posType] != -1) {
        return dp[pos][0][mod][posType];
    }

    long long ans = 0;

    // 确定当前位可选数字范围
    int up = isLimit ? digits[pos] : 9;
    int start = (pos == 0) ? 1 : 0; // 第一位不能为 0

    // 枚举当前位可选数字
    for (int d_val = start; d_val <= up; d_val++) {
        // 根据位置类型检查约束条件
        if (posType == 1) { // 偶数位置：必须等于 d
            if (d_val != d) continue;
        } else { // 奇数位置：必须不等于 d
            if (d_val == d) continue;
        }

        // 计算新的余数
        int newMod = (mod * 10 + d_val) % m;
        // 下一个位置类型：奇偶交替
        int newPosType = 1 - posType;

        ans = (ans + dfs(pos + 1, isLimit && d_val == up, newMod, newPosType, dp)) % MOD;
    }

    // 记忆化存储
    if (!isLimit) {
        dp[pos][0][mod][posType] = ans;
    }
```

```
        return ans;
    }
};

/**
 * 单元测试函数
 */
void testMagicNumbers() {
    cout << "=== 测试 Magic Numbers ===" << endl;

    // 测试用例 1: 简单情况
    {
        MagicNumbers mn(7, 7);
        string a = "1", b = "100";
        long long result = mn.countMagicNumbers(a, b);
        cout << "d=7, m=7, 区间[" << a << ", " << b << "]" << endl;
        cout << "结果: " << result << endl;
        cout << "预期: 包含 7, 77 等数字" << endl;
        cout << endl;
    }

    // 测试用例 2: 边界情况
    {
        MagicNumbers mn(1, 1);
        string a = "1", b = "9";
        long long result = mn.countMagicNumbers(a, b);
        cout << "d=1, m=1, 区间[" << a << ", " << b << "]" << endl;
        cout << "结果: " << result << endl;
        cout << "预期: 所有奇数位置的数字不能是 1" << endl;
        cout << endl;
    }

    // 测试用例 3: 较大范围
    {
        MagicNumbers mn(4, 13);
        string a = "100", b = "1000";
        long long result = mn.countMagicNumbers(a, b);
        cout << "d=4, m=13, 区间[" << a << ", " << b << "]" << endl;
        cout << "结果: " << result << endl;
        cout << endl;
    }
}
```

```
/**
 * 性能测试函数
 */
void performanceTest() {
    cout << "=== 性能测试 ===" << endl;

    vector<tuple<int, int, string, string>> testCases = {
        {7, 7, "1", "1000000"},
        {3, 11, "1", "1000000000"},
        {9, 19, "1", "1000000000000"}
    };

    for (auto& testCase : testCases) {
        int d = get<0>(testCase);
        int m = get<1>(testCase);
        string a = get<2>(testCase);
        string b = get<3>(testCase);

        MagicNumbers mn(d, m);

        auto start = chrono::high_resolution_clock::now();
        long long result = mn.countMagicNumbers(a, b);
        auto end = chrono::high_resolution_clock::now();

        auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);

        cout << "d=" << d << ", m=" << m << ", 区间[" << a << ", " << b << "]" << endl;
        cout << "结果: " << result << endl;
        cout << "耗时: " << duration.count() << "毫秒" << endl;
        cout << endl;
    }
}

/**
 * 调试函数：验证特定数字是否为 magic number
 */
void debugMagicNumbers() {
    cout << "=== 调试 Magic Numbers ===" << endl;

    MagicNumbers mn(7, 7);

    vector<string> testNumbers = {"7", "17", "27", "77", "177", "707", "717", "727"};
```

```
    for (const string& num : testNumbers) {
        bool isMagic = mn.isMagicNumber(num);
        cout << "数字 " << num << ": " << (isMagic ? "是" : "不是") << "magic number" << endl;

        if (isMagic) {
            int mod = 0;
            for (int i = 0; i < num.length(); i++) {
                int digit = num[i] - '0';
                int posType = (i + 1) % 2;
                cout << "  位置" << (i+1) << "(" << (posType == 1 ? "偶数" : "奇数")
                     << "): 数字=" << digit;
                if (posType == 1) {
                    cout << " 必须等于7: " << (digit == 7 ? "满足" : "不满足");
                } else {
                    cout << " 必须不等于7: " << (digit != 7 ? "满足" : "不满足");
                }
                cout << endl;
                mod = (mod * 10 + digit) % 7;
            }
            cout << "  余数: " << mod << "%7=" << mod % 7 << endl;
        }
        cout << endl;
    }
}

/**
 * 工程化考量：
 * 1. 模运算：结果对10^9+7取模，避免溢出
 * 2. 字符串处理：支持大数字输入
 * 3. 边界处理：正确处理前导零和空字符串
 * 4. 状态设计：合理设计状态参数，减少状态数
 * 5. 记忆化优化：避免重复计算相同状态
 *
 * 算法特色：
 * 1. 位置相关约束：根据位置奇偶性应用不同约束
 * 2. 模运算约束：数字必须能被m整除
 * 3. 前导零处理：第一位不能为0
 * 4. 记忆化搜索：提高算法效率
 */

int main() {
    // 运行功能测试
    testMagicNumbers();
```

```
    // 运行性能测试
    performanceTest();

    // 调试模式
    debugMagicNumbers();

    return 0;
}
```

==========

文件: Code14_MagicNumbersCF.java

==========

```
package class084;

import java.util.*;

/**
 * Codeforces 628D. Magic Numbers
 * 题目链接：https://codeforces.com/problemset/problem/628/D
 *
 * 题目描述：
 * 定义一个 d-magic number 为满足以下条件的数字：
 * 1. 数字的十进制表示中，所有在偶数位置（从 1 开始计数）的数字都等于 d
 * 2. 数字的十进制表示中，所有在奇数位置（从 1 开始计数）的数字都不等于 d
 * 3. 数字不能有前导零
 * 给定区间[a, b]和数字 d，求其中 d-magic number 的个数，结果对 10^9+7 取模。
 *
 * 解题思路：
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 当前数字对 m 的余数
 *    - 当前位置的奇偶性
 * 3. 关键点：根据位置奇偶性判断数字是否等于 d
 *
 * 时间复杂度分析：
 * - 状态数：2000 * 2 * 2 * 2000 ≈ 16,000,000
 * - 每个状态处理 10 种选择
 * - 总复杂度：O(160,000,000) 在可接受范围内
 *
```

```
 * 空间复杂度分析：
 * - 记忆化数组：2000 * 2 * 2 * 2000 ≈ 64MB
 *
 * 最优解分析：
 * 这是标准的最优解，利用数位 DP 处理位置相关的约束条件
 */

public class Code14_MagicNumbersCF {
    private static final int MOD = 1000000007;
    private int d;          // 魔法数字 d
    private int m;          // 模数 m
    private int[] digits;   // 存储数位

    public Code14_MagicNumbersCF(int d, int m) {
        this.d = d;
        this.m = m;
    }

    /**
     * 计算区间[a, b]中 d-magic number 的个数
     * 时间复杂度: O(len(b) * 2 * 2 * m)
     * 空间复杂度: O(len(b) * 2 * 2 * m)
     */
    public long countMagicNumbers(String a, String b) {
        long countB = countUpTo(b);
        long countA = countUpTo(a);

        // 需要检查 a 本身是否是 magic number
        if (isMagicNumber(a)) {
            countA = (countA - 1 + MOD) % MOD;
        }

        return (countB - countA + MOD) % MOD;
    }

    /**
     * 计算[0, s]中 d-magic number 的个数
     */
    private long countUpTo(String s) {
        if (s.isEmpty()) return 0;

        // 将字符串转换为数位数组
        digits = new int[s.length()];
```

```
    for (int i = 0; i < s.length(); i++) {
        digits[i] = s.charAt(i) - '0';
    }

    int len = digits.length;

    // 记忆化数组：dp[pos][isLimit][mod][posType]
    // posType: 0 表示奇数位置，1 表示偶数位置
    Long[][][][] dp = new Long[len][2][m][2];

    // 使用 DFS 进行数位 DP
    // 从第 0 位开始，位置类型为 1（偶数位置，因为从 1 开始计数）
    return dfs(0, true, 0, 1, dp);
}

/**
 * 数位 DP 递归函数
 *
 * @param pos 当前处理的位置（0-indexed）
 * @param isLimit 是否受到上界限制
 * @param mod 当前数字对 m 的余数
 * @param posType 位置类型：1 表示偶数位置，0 表示奇数位置
 * @param dp 记忆化数组
 * @return 满足条件的数字个数
 */
private long dfs(int pos, boolean isLimit, int mod, int posType, Long[][][][] dp) {
    // 递归终止条件：处理完所有数位
    if (pos == digits.length) {
        // 必须是一个有效的数字（不能是前导零情况）
        // 余数为 0 且是有效的 magic number
        return (mod == 0) ? 1 : 0;
    }

    // 记忆化搜索
    if (!isLimit && dp[pos][0][mod][posType] != null) {
        return dp[pos][0][mod][posType];
    }

    long ans = 0;

    // 确定当前位可选数字范围
    int up = isLimit ? digits[pos] : 9;
    int start = (pos == 0) ? 1 : 0; // 第一位不能为 0
```

```
    // 枚举当前位可选数字
    for (int dVal = start; dVal <= up; dVal++) {
        // 根据位置类型检查约束条件
        if (posType == 1) { // 偶数位置：必须等于 d
            if (dVal != d) continue;
        } else { // 奇数位置：必须不等于 d
            if (dVal == d) continue;
        }

        // 计算新的余数
        int newMod = (mod * 10 + dVal) % m;
        // 下一个位置类型：奇偶交替
        int newPosType = 1 - posType;

        ans = (ans + dfs(pos + 1, isLimit && dVal == up, newMod, newPosType, dp)) % MOD;
    }

    // 记忆化存储
    if (!isLimit) {
        dp[pos][0][mod][posType] = ans;
    }

    return ans;
}

/**
 * 检查一个字符串是否表示一个 d-magic number
 * 用于验证边界情况
 */
private boolean isMagicNumber(String s) {
    if (s.isEmpty() || s.charAt(0) == '0') return false;

    int mod = 0;
    for (int i = 0; i < s.length(); i++) {
        int digit = s.charAt(i) - '0';
        int posType = (i + 1) % 2; // 1-indexed: 偶数位置对应 i+1 为偶数

        if (posType == 1) { // 偶数位置
            if (digit != d) return false;
        } else { // 奇数位置
            if (digit == d) return false;
        }
```

```
            mod = (mod * 10 + digit) % m;
        }

        return mod == 0;
    }

    /**
     * 单元测试函数
     */
    public static void testMagicNumbers() {
        System.out.println("=== 测试 Magic Numbers ===");

        // 测试用例 1: 简单情况
        {
            Code14_MagicNumbersCF mn = new Code14_MagicNumbersCF(7, 7);
            String a = "1", b = "100";
            long result = mn.countMagicNumbers(a, b);
            System.out.println("d=7, m=7, 区间[" + a + ", " + b + "]");
            System.out.println("结果: " + result);
            System.out.println("预期: 包含 7, 77 等数字");
            System.out.println();
        }

        // 测试用例 2: 边界情况
        {
            Code14_MagicNumbersCF mn = new Code14_MagicNumbersCF(1, 1);
            String a = "1", b = "9";
            long result = mn.countMagicNumbers(a, b);
            System.out.println("d=1, m=1, 区间[" + a + ", " + b + "]");
            System.out.println("结果: " + result);
            System.out.println("预期: 所有奇数位置的数字不能是 1");
            System.out.println();
        }
    }

    /**
     * 性能测试函数
     */
    public static void performanceTest() {
        System.out.println("=== 性能测试 ===");

        Object[][] testCases = {
```

```
            {7, 7, "1", "1000000"},
            {3, 11, "1", "1000000000"},
            {9, 19, "1", "1000000000000"}
        };

        for (Object[] testCase : testCases) {
            int d = (int) testCase[0];
            int m = (int) testCase[1];
            String a = (String) testCase[2];
            String b = (String) testCase[3];

            Code14_MagicNumbersCF mn = new Code14_MagicNumbersCF(d, m);

            long startTime = System.currentTimeMillis();
            long result = mn.countMagicNumbers(a, b);
            long endTime = System.currentTimeMillis();

            System.out.println("d=" + d + ", m=" + m + ", 区间[" + a + ", " + b + "]");
            System.out.println("结果: " + result);
            System.out.println("耗时: " + (endTime - startTime) + "毫秒");
            System.out.println();
        }
    }

    /**
     * 调试函数：验证特定数字是否为magic number
     */
    public static void debugMagicNumbers() {
        System.out.println("=== 调试Magic Numbers ===");

        Code14_MagicNumbersCF mn = new Code14_MagicNumbersCF(7, 7);

        String[] testNumbers = {"7", "17", "27", "77", "177", "707", "717", "727"};

        for (String num : testNumbers) {
            boolean isMagic = mn.isMagicNumber(num);
            System.out.println("数字 " + num + ": " + (isMagic ? "是" : "不是") + "magic
number");

            if (isMagic) {
                int mod = 0;
                for (int i = 0; i < num.length(); i++) {
                    int digit = num.charAt(i) - '0';
```

```
                int posType = (i + 1) % 2;
                System.out.print("  位置" + (i+1) + "(" + (posType == 1 ? "偶数" : "奇数")
                        + "): 数字=" + digit);
                if (posType == 1) {
                    System.out.print(" 必须等于 7: " + (digit == 7 ? "满足" : "不满足"));
                } else {
                    System.out.print(" 必须不等于 7: " + (digit != 7 ? "满足" : "不满足"));
                }
                System.out.println();
                mod = (mod * 10 + digit) % 7;
            }
            System.out.println("  余数: " + mod + "%7=" + mod % 7);
        }
        System.out.println();
    }
}

/**
 * 工程化考量总结：
 * 1. 模运算：结果对 10^9+7 取模，避免溢出
 * 2. 字符串处理：支持大数字输入
 * 3. 边界处理：正确处理前导零和空字符串
 * 4. 状态设计：合理设计状态参数，减少状态数
 * 5. 记忆化优化：避免重复计算相同状态
 *
 * 算法特色：
 * 1. 位置相关约束：根据位置奇偶性应用不同约束
 * 2. 模运算约束：数字必须能被 m 整除
 * 3. 前导零处理：第一位不能为 0
 * 4. 记忆化搜索：提高算法效率
 */

public static void main(String[] args) {
    // 运行功能测试
    testMagicNumbers();

    // 运行性能测试
    performanceTest();

    // 调试模式
    debugMagicNumbers();

    // 边界测试
```

```
        System.out.println("=== 边界测试 ===");
        Code14_MagicNumbersCF mn = new Code14_MagicNumbersCF(7, 7);
        System.out.println("区间[7, 7]: " + mn.countMagicNumbers("7", "7"));
        System.out.println("区间[1, 1]: " + mn.countMagicNumbers("1", "1"));
    }
}
```

==========

文件: Code14_MagicNumbersCF.py

==========

```
"""
Codeforces 628D. Magic Numbers
题目链接：https://codeforces.com/problemset/problem/628/D

题目描述：
定义一个 d-magic number 为满足以下条件的数字：
1. 数字的十进制表示中，所有在偶数位置（从 1 开始计数）的数字都等于 d
2. 数字的十进制表示中，所有在奇数位置（从 1 开始计数）的数字都不等于 d
3. 数字不能有前导零
给定区间[a, b]和数字 d，求其中 d-magic number 的个数，结果对 10^9+7 取模。

解题思路：
1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
2. 状态设计需要记录：
   - 当前处理位置
   - 是否受上界限制
   - 当前数字对 m 的余数
   - 当前位置的奇偶性
3. 关键点：根据位置奇偶性判断数字是否等于 d

时间复杂度分析：
- 状态数：2000 * 2 * 2 * 2000 ≈ 16,000,000
- 每个状态处理 10 种选择
- 总复杂度：O(160,000,000) 在可接受范围内

空间复杂度分析：
- 记忆化数组：2000 * 2 * 2 * 2000 ≈ 64MB

最优解分析：
这是标准的最优解，利用数位 DP 处理位置相关的约束条件
"""
```

```
from functools import lru_cache

class MagicNumbers:
    def __init__(self, d: int, m: int):
        self.MOD = 10**9 + 7
        self.d = d  # 魔法数字 d
        self.m = m  # 模数 m

    def count_magic_numbers(self, a: str, b: str) -> int:
        """
        计算区间[a, b]中 d-magic number 的个数
        时间复杂度: O(len(b) * 2 * 2 * m)
        空间复杂度: O(len(b) * 2 * 2 * m)
        """
        count_b = self.count_up_to(b)
        count_a = self.count_up_to(a)

        # 需要检查 a 本身是否是 magic number
        if self.is_magic_number(a):
            count_a = (count_a - 1) % self.MOD

        return (count_b - count_a) % self.MOD

    def count_up_to(self, s: str) -> int:
        """
        计算[0, s]中 d-magic number 的个数
        """
        if not s:
            return 0

        digits = [int(c) for c in s]

        @lru_cache(maxsize=None)
        def dfs(pos: int, is_limit: bool, mod: int, pos_type: int) -> int:
            """
            数位 DP 递归函数

            Args:
                pos: 当前处理的位置（0-indexed）
                is_limit: 是否受到上界限制
                mod: 当前数字对 m 的余数
                pos_type: 位置类型：1 表示偶数位置，0 表示奇数位置
```

```
        Returns:
            满足条件的数字个数
        """
        # 递归终止条件：处理完所有数位
        if pos == len(digits):
            # 必须是一个有效的数字（不能是前导零情况）
            # 余数为 0 且是有效的 magic number
            return 1 if mod == 0 else 0

        ans = 0

        # 确定当前位可选数字范围
        up = digits[pos] if is_limit else 9
        start = 1 if pos == 0 else 0  # 第一位不能为 0

        # 枚举当前位可选数字
        for d_val in range(start, up + 1):
            # 根据位置类型检查约束条件
            if pos_type == 1:  # 偶数位置：必须等于 d
                if d_val != self.d:
                    continue
            else:  # 奇数位置：必须不等于 d
                if d_val == self.d:
                    continue

            # 计算新的余数
            new_mod = (mod * 10 + d_val) % self.m
            # 下一个位置类型：奇偶交替
            new_pos_type = 1 - pos_type

            ans += dfs(pos + 1, is_limit and d_val == up, new_mod, new_pos_type)

        return ans % self.MOD

    # 从第 0 位开始，位置类型为 1（偶数位置，因为从 1 开始计数）
    return dfs(0, True, 0, 1)

def is_magic_number(self, s: str) -> bool:
    """
    检查一个字符串是否表示一个 d-magic number
    用于验证边界情况
    """
    if not s or s[0] == '0':
```

```
            return False

        mod = 0
        for i, char in enumerate(s):
            digit = int(char)
            pos_type = (i + 1) % 2  # 1-indexed: 偶数位置对应 i+1 为偶数

            if pos_type == 1:  # 偶数位置
                if digit != self.d:
                    return False
            else:  # 奇数位置
                if digit == self.d:
                    return False

            mod = (mod * 10 + digit) % self.m

        return mod == 0

def test_magic_numbers():
    """
    单元测试函数
    """
    print("=== 测试 Magic Numbers ===")

    # 测试用例 1: 简单情况
    mn = MagicNumbers(7, 7)
    a, b = "1", "100"
    result = mn.count_magic_numbers(a, b)
    print(f"d=7, m=7, 区间[{a}, {b}]")
    print(f"结果: {result}")
    print("预期: 包含 7, 77 等数字")
    print()

    # 测试用例 2: 边界情况
    mn = MagicNumbers(1, 1)
    a, b = "1", "9"
    result = mn.count_magic_numbers(a, b)
    print(f"d=1, m=1, 区间[{a}, {b}]")
    print(f"结果: {result}")
    print("预期: 所有奇数位置的数字不能是 1")
    print()

def performance_test():
```

```
    """
    性能测试函数
    """
    import time
    print("=== 性能测试 ===")

    test_cases = [
        (7, 7, "1", "1000000"),
        (3, 11, "1", "1000000000"),
        (9, 19, "1", "1000000000000")
    ]

    for d, m, a, b in test_cases:
        mn = MagicNumbers(d, m)

        start_time = time.time()
        result = mn.count_magic_numbers(a, b)
        end_time = time.time()

        print(f"d={d}, m={m}, 区间[{a}, {b}]")
        print(f"结果: {result}")
        print(f"耗时: {(end_time - start_time)*1000:.2f}毫秒")
        print()

def debug_magic_numbers():
    """
    调试函数：验证特定数字是否为 magic number
    """
    print("=== 调试 Magic Numbers ===")

    mn = MagicNumbers(7, 7)

    test_numbers = ["7", "17", "27", "77", "177", "707", "717", "727"]

    for num in test_numbers:
        is_magic = mn.is_magic_number(num)
        print(f"数字 {num}: {'是' if is_magic else '不是'}magic number")

        if is_magic:
            mod = 0
            for i, char in enumerate(num):
                digit = int(char)
                pos_type = (i + 1) % 2
```

```
                print(f"  位置{i+1}({'偶数' if pos_type == 1 else '奇数'}): 数字={digit}",
end="")
                if pos_type == 1:
                    print(f" 必须等于 7: {'满足' if digit == 7 else '不满足'}")
                else:
                    print(f" 必须不等于 7: {'满足' if digit != 7 else '不满足'}")
                mod = (mod * 10 + digit) % 7
            print(f"  余数: {mod}%7={mod % 7}")
        print()

def manual_verification():
    """
    手动验证函数：验证算法逻辑
    """
    print("=== 手动验证 ===")

    # 验证位置计数规则
    test_cases = [
        ("7", True),    # 位置 1(偶数):7=7 ✓
        ("17", False),  # 位置 1(偶数):1≠7 ✗
        ("27", False),  # 位置 1(偶数):2≠7 ✗
        ("77", True),   # 位置 1(偶数):7=7 ✓, 位置 2(奇数):7=7 ✗
        ("707", False), # 位置 1(偶数):7=7 ✓, 位置 2(奇数):0≠7 ✓, 位置 3(偶数):7=7 ✓, 但 0 是前导
零?
    ]

    mn = MagicNumbers(7, 7)

    for num, expected in test_cases:
        actual = mn.is_magic_number(num)
        status = "✓" if actual == expected else "✗"
        print(f"{num}: 预期{expected}, 实际{actual} {status}")

"""
工程化考量总结：
1. 模运算：结果对 10^9+7 取模，避免溢出
2. 字符串处理：支持大数字输入
3. 边界处理：正确处理前导零和空字符串
4. 状态设计：合理设计状态参数，减少状态数
5. 记忆化优化：使用 lru_cache 自动管理记忆化

Python 语言特性利用：
1. 装饰器：使用@lru_cache 实现自动记忆化
```

2. 类型注解：提高代码可读性
3. 内置函数：简化代码逻辑
4. 动态类型：灵活处理各种情况

跨语言实现差异：
- Python: 代码简洁，使用装饰器自动记忆化
- Java: 使用多维数组，需要手动管理记忆化
- C++: 使用 vector 和 lambda，注意内存管理

算法调试技巧：
1. 小范围验证：手动计算小范围结果进行对拍
2. 中间状态打印：添加调试信息打印中间状态
3. 边界测试：测试 0、1、边界值等特殊情况
4. 性能分析：使用 time 模块进行性能测试
"""

```
if __name__ == "__main__":
    # 运行功能测试
    test_magic_numbers()

    # 运行性能测试
    performance_test()

    # 调试模式
    debug_magic_numbers()

    # 手动验证
    manual_verification()

    # 边界测试
    print("=== 边界测试 ===")
    mn = MagicNumbers(7, 7)
    print(f"区间[7, 7]: {mn.count_magic_numbers('7', '7')}")
    print(f"区间[1, 1]: {mn.count_magic_numbers('1', '1')}")
```

==========

文件: Code15_DigitsParadeABC.cpp
==========
```
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
```

```
#include <algorithm>
#include <chrono>
#include <cmath>

using namespace std;

/**
 * AtCoder ABC135 D - Digits Parade
 * 题目链接：https://atcoder.jp/contests/abc135/tasks/abc135_d
 *
 * 题目描述：
 * 给定一个由数字和'?'组成的字符串 S，'?'可以替换成 0-9 的任意数字。
 * 求有多少种替换方案使得结果能被 13 整除，结果对 10^9+7 取模。
 *
 * 解题思路：
 * 1. 动态规划方法：使用 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 当前数字对 13 的余数
 * 3. 关键点：'?'可以替换为 0-9 的任意数字
 *
 * 时间复杂度分析：
 * - 状态数：字符串长度 × 13 ≈ 10^5 × 13 = 1.3×10^6
 * - 每个状态处理最多 10 种选择
 * - 总复杂度：O(13×10^5) 在可接受范围内
 *
 * 空间复杂度分析：
 * - DP 数组：10^5 × 13 ≈ 1.3MB
 *
 * 最优解分析：
 * 这是标准的最优解，利用 DP 处理模运算和通配符替换
 */

class DigitsParade {
private:
    static const int MOD = 1000000007;
    static const int DIVISOR = 13;

public:
    /**
     * 计算有多少种替换方案使得结果能被 13 整除
     * 时间复杂度: O(n * 13)
     * 空间复杂度: O(n * 13)
```

```
 */
int countDivisibleBy13(const string& s) {
    int n = s.length();

    // dp[i][r] 表示处理到第 i 位，当前余数为 r 的方案数
    vector<vector<long long>> dp(n + 1, vector<long long>(DIVISOR, 0));
    dp[0][0] = 1;  // 初始状态：余数为 0 有 1 种方案（空数字）

    // 从高位到低位动态规划
    for (int i = 0; i < n; i++) {
        for (int r = 0; r < DIVISOR; r++) {
            if (dp[i][r] == 0) continue;

            if (s[i] == '?') {
                // '?'可以替换为 0-9 的任意数字
                for (int d = 0; d <= 9; d++) {
                    int newR = (r * 10 + d) % DIVISOR;
                    dp[i + 1][newR] = (dp[i + 1][newR] + dp[i][r]) % MOD;
                }
            } else {
                // 固定数字
                int d = s[i] - '0';
                int newR = (r * 10 + d) % DIVISOR;
                dp[i + 1][newR] = (dp[i + 1][newR] + dp[i][r]) % MOD;
            }
        }
    }

    return dp[n][0];
}

/**
 * 使用记忆化 DFS 的替代解法（更符合数位 DP 传统风格）
 * 时间复杂度: O(n * 13)
 * 空间复杂度: O(n * 13)
 */
int countDivisibleBy13DFS(const string& s) {
    int n = s.length();
    vector<vector<long long>> memo(n, vector<long long>(DIVISOR, -1));

    return dfs(s, 0, 0, memo);
}
```

```
private:
    long long dfs(const string& s, int pos, int remainder, vector<vector<long long>>& memo) {
        // 递归终止条件：处理完所有字符
        if (pos == s.length()) {
            return (remainder == 0) ? 1 : 0;
        }

        // 记忆化搜索
        if (memo[pos][remainder] != -1) {
            return memo[pos][remainder];
        }

        long long count = 0;

        if (s[pos] == '?') {
            // '?'可以替换为0-9的任意数字
            for (int d = 0; d <= 9; d++) {
                int newRemainder = (remainder * 10 + d) % DIVISOR;
                count = (count + dfs(s, pos + 1, newRemainder, memo)) % MOD;
            }
        } else {
            // 固定数字
            int d = s[pos] - '0';
            int newRemainder = (remainder * 10 + d) % DIVISOR;
            count = (count + dfs(s, pos + 1, newRemainder, memo)) % MOD;
        }

        // 记忆化存储
        memo[pos][remainder] = count;
        return count;
    }
};

/**
 * 单元测试函数
 */
void testDigitsParade() {
    cout << "=== 测试Digits Parade ===" << endl;

    DigitsParade dp;

    // 测试用例1: 简单情况
    string s1 = "??";
```

```
    int result1 = dp.countDivisibleBy13(s1);
    int result1DFS = dp.countDivisibleBy13DFS(s1);
    cout << "输入: " << s1 << endl;
    cout << "DP 结果: " << result1 << endl;
    cout << "DFS 结果: " << result1DFS << endl;
    cout << "结果一致: " << (result1 == result1DFS) << endl;
    cout << "预期: 100 种组合中有几个能被 13 整除" << endl;
    cout << endl;

    // 测试用例 2: 固定数字
    string s2 = "13";
    int result2 = dp.countDivisibleBy13(s2);
    int result2DFS = dp.countDivisibleBy13DFS(s2);
    cout << "输入: " << s2 << endl;
    cout << "DP 结果: " << result2 << endl;
    cout << "DFS 结果: " << result2DFS << endl;
    cout << "结果一致: " << (result2 == result2DFS) << endl;
    cout << "预期: 13 能被 13 整除，所以为 1" << endl;
    cout << endl;

    // 测试用例 3: 混合情况
    string s3 = "1?2";
    int result3 = dp.countDivisibleBy13(s3);
    int result3DFS = dp.countDivisibleBy13DFS(s3);
    cout << "输入: " << s3 << endl;
    cout << "DP 结果: " << result3 << endl;
    cout << "DFS 结果: " << result3DFS << endl;
    cout << "结果一致: " << (result3 == result3DFS) << endl;
    cout << endl;
}

/**
 * 性能测试函数
 */
void performanceTest() {
    cout << "=== 性能测试 ===" << endl;

    DigitsParade dp;

    // 生成测试用例
    string longString(1000, '?');

    // 测试 DP 方法
```

```
    auto startTimeDP = chrono::high_resolution_clock::now();
    int resultDP = dp.countDivisibleBy13(longString);
    auto endTimeDP = chrono::high_resolution_clock::now();

    // 测试 DFS 方法
    auto startTimeDFS = chrono::high_resolution_clock::now();
    int resultDFS = dp.countDivisibleBy13DFS(longString);
    auto endTimeDFS = chrono::high_resolution_clock::now();

    auto durationDP = chrono::duration_cast<chrono::milliseconds>(endTimeDP - startTimeDP);
    auto durationDFS = chrono::duration_cast<chrono::milliseconds>(endTimeDFS - startTimeDFS);

    cout << "字符串长度: " << longString.length() << endl;
    cout << "DP 方法耗时: " << durationDP.count() << "ms" << endl;
    cout << "DFS 方法耗时: " << durationDFS.count() << "ms" << endl;
    cout << "结果一致: " << (resultDP == resultDFS) << endl;
    cout << endl;
}

/**
 * 调试函数：验证特定字符串的替换方案
 */
void debugDigitsParade() {
    cout << "=== 调试 Digits Parade ===" << endl;

    DigitsParade dp;

    vector<string> testCases = {
        "0", "1", "13", "26", "39", "52",
        "1?", "?3", "??", "1?3"
    };

    for (const string& s : testCases) {
        int result = dp.countDivisibleBy13(s);
        cout << "输入: " << s << ", 方案数: " << result << endl;

        // 对于短字符串，可以手动验证
        if (s.length() <= 2 && s.find('?') != string::npos) {
            cout << "  具体方案: ";
            int count = 0;
            int maxNum = pow(10, s.length());
            for (int i = 0; i < maxNum; i++) {
                string candidate = to_string(i);
```

```
                while (candidate.length() < s.length()) {
                    candidate = "0" + candidate;
                }

                bool match = true;
                for (int j = 0; j < s.length(); j++) {
                    if (s[j] != '?' && s[j] != candidate[j]) {
                        match = false;
                        break;
                    }
                }

                if (match && i % 13 == 0) {
                    cout << candidate << " ";
                    count++;
                }
            }
            cout << endl << "  手动计数: " << count << endl;
        }
        cout << endl;
    }
}

/**
 * 工程化考量总结：
 * 1. 模运算：结果对 10^9+7 取模，避免溢出
 * 2. 状态设计：合理设计状态参数，减少状态数
 * 3. 两种实现：提供 DP 和 DFS 两种解法，便于理解
 * 4. 性能优化：使用迭代 DP 避免递归栈开销
 * 5. 边界处理：正确处理空字符串和全'?'情况
 *
 * 算法特色：
 * 1. 通配符处理：'?'可以替换为任意数字
 * 2. 模运算约束：结果必须能被 13 整除
 * 3. 动态规划：从高位到低位逐步计算
 * 4. 记忆化搜索：DFS 解法更符合数位 DP 传统
 *
 * C++实现特点：
 * 1. 使用 vector 代替数组，更安全
 * 2. 使用 constexpr 定义常量
 * 3. 避免使用全局变量
 * 4. 提供完整的测试框架
 */
```

```
int main() {
    // 运行功能测试
    testDigitsParade();

    // 运行性能测试
    performanceTest();

    // 调试模式
    debugDigitsParade();

    // 边界测试
    cout << "=== 边界测试 ===" << endl;
    DigitsParade dp;
    cout << "空字符串: " << dp.countDivisibleBy13("") << endl;
    cout << "单个'?': " << dp.countDivisibleBy13("?") << endl;
    cout << "全'?': " << dp.countDivisibleBy13("???") << endl;

    return 0;
}
```

```
==============================================
```

文件: Code15_DigitsParadeABC.java

```
==============================================
package class084;

import java.util.*;

/**
 * AtCoder ABC135 D - Digits Parade
 * 题目链接：https://atcoder.jp/contests/abc135/tasks/abc135_d
 *
 * 题目描述：
 * 给定一个由数字和'?'组成的字符串 S，'?'可以替换成 0-9 的任意数字。
 * 求有多少种替换方案使得结果能被 13 整除，结果对 10^9+7 取模。
 *
 * 解题思路：
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 当前数字对 13 的余数
 * 3. 关键点：'?'可以替换为 0-9 的任意数字
```

```
 *
 * 时间复杂度分析：
 * - 状态数：字符串长度 × 13 ≈ 10^5 × 13 = 1.3×10^6
 * - 每个状态处理最多 10 种选择
 * - 总复杂度：O(13×10^5) 在可接受范围内
 *
 * 空间复杂度分析：
 * - 记忆化数组：10^5 × 13 ≈ 1.3MB
 *
 * 最优解分析：
 * 这是标准的最优解，利用数位 DP 处理模运算和通配符替换
 */

public class Code15_DigitsParadeABC {
    private static final int MOD = 1000000007;
    private static final int DIVISOR = 13;

    /**
     * 计算有多少种替换方案使得结果能被 13 整除
     * 时间复杂度: O(n * 13)
     * 空间复杂度: O(n * 13)
     */
    public static int countDivisibleBy13(String s) {
        int n = s.length();
        char[] chars = s.toCharArray();

        // dp[i][r] 表示处理到第 i 位，当前余数为 r 的方案数
        long[][] dp = new long[n + 1][DIVISOR];
        dp[0][0] = 1;  // 初始状态：余数为 0 有 1 种方案（空数字）

        // 从高位到低位动态规划
        for (int i = 0; i < n; i++) {
            for (int r = 0; r < DIVISOR; r++) {
                if (dp[i][r] == 0) continue;

                if (chars[i] == '?') {
                    // '?'可以替换为 0-9 的任意数字
                    for (int d = 0; d <= 9; d++) {
                        int newR = (r * 10 + d) % DIVISOR;
                        dp[i + 1][newR] = (dp[i + 1][newR] + dp[i][r]) % MOD;
                    }
                } else {
                    // 固定数字
```

```
                int d = chars[i] - '0';
                int newR = (r * 10 + d) % DIVISOR;
                dp[i + 1][newR] = (dp[i + 1][newR] + dp[i][r]) % MOD;
            }
        }
    }

    return (int) dp[n][0];
}

/**
 * 使用记忆化 DFS 的替代解法（更符合数位 DP 传统风格）
 * 时间复杂度: O(n * 13)
 * 空间复杂度: O(n * 13)
 */
public static int countDivisibleBy13DFS(String s) {
    char[] chars = s.toCharArray();
    int n = chars.length;

    // 记忆化数组
    Long[][] memo = new Long[n][DIVISOR];

    return (int) dfs(chars, 0, 0, memo);
}

private static long dfs(char[] chars, int pos, int remainder, Long[][] memo) {
    // 递归终止条件：处理完所有字符
    if (pos == chars.length) {
        return (remainder == 0) ? 1 : 0;
    }

    // 记忆化搜索
    if (memo[pos][remainder] != null) {
        return memo[pos][remainder];
    }

    long count = 0;

    if (chars[pos] == '?') {
        // '?'可以替换为 0-9 的任意数字
        for (int d = 0; d <= 9; d++) {
            int newRemainder = (remainder * 10 + d) % DIVISOR;
            count = (count + dfs(chars, pos + 1, newRemainder, memo)) % MOD;
```

```
        }
    } else {
        // 固定数字
        int d = chars[pos] - '0';
        int newRemainder = (remainder * 10 + d) % DIVISOR;
        count = (count + dfs(chars, pos + 1, newRemainder, memo)) % MOD;
    }

    // 记忆化存储
    memo[pos][remainder] = count;
    return count;
}

/**
 * 单元测试函数
 */
public static void testDigitsParade() {
    System.out.println("=== 测试 Digits Parade ===");

    // 测试用例 1: 简单情况
    String s1 = "??";
    int result1 = countDivisibleBy13(s1);
    int result1DFS = countDivisibleBy13DFS(s1);
    System.out.println("输入: " + s1);
    System.out.println("DP 结果: " + result1);
    System.out.println("DFS 结果: " + result1DFS);
    System.out.println("结果一致: " + (result1 == result1DFS));
    System.out.println("预期: 100 种组合中有几个能被 13 整除");
    System.out.println();

    // 测试用例 2: 固定数字
    String s2 = "13";
    int result2 = countDivisibleBy13(s2);
    int result2DFS = countDivisibleBy13DFS(s2);
    System.out.println("输入: " + s2);
    System.out.println("DP 结果: " + result2);
    System.out.println("DFS 结果: " + result2DFS);
    System.out.println("结果一致: " + (result2 == result2DFS));
    System.out.println("预期: 13 能被 13 整除，所以为 1");
    System.out.println();

    // 测试用例 3: 混合情况
    String s3 = "1?2";
```

```
    int result3 = countDivisibleBy13(s3);
    int result3DFS = countDivisibleBy13DFS(s3);
    System.out.println("输入: " + s3);
    System.out.println("DP 结果: " + result3);
    System.out.println("DFS 结果: " + result3DFS);
    System.out.println("结果一致: " + (result3 == result3DFS));
    System.out.println();
}

/**
 * 性能测试函数
 */
public static void performanceTest() {
    System.out.println("=== 性能测试 ===");

    // 生成测试用例
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 10000; i++) {
        sb.append('?');
    }
    String longString = sb.toString();

    // 测试 DP 方法
    long startTimeDP = System.currentTimeMillis();
    int resultDP = countDivisibleBy13(longString.substring(0, 1000));
    long endTimeDP = System.currentTimeMillis();

    // 测试 DFS 方法
    long startTimeDFS = System.currentTimeMillis();
    int resultDFS = countDivisibleBy13DFS(longString.substring(0, 1000));
    long endTimeDFS = System.currentTimeMillis();

    System.out.println("字符串长度: 1000");
    System.out.println("DP 方法耗时: " + (endTimeDP - startTimeDP) + "ms");
    System.out.println("DFS 方法耗时: " + (endTimeDFS - startTimeDFS) + "ms");
    System.out.println("结果一致: " + (resultDP == resultDFS));
    System.out.println();
}

/**
 * 调试函数：验证特定字符串的替换方案
 */
public static void debugDigitsParade() {
```

```
        System.out.println("=== 调试 Digits Parade ===");

        String[] testCases = {
            "0", "1", "13", "26", "39", "52",
            "1?", "?3", "??", "1?3"
        };

        for (String s : testCases) {
            int result = countDivisibleBy13(s);
            System.out.println("输入: " + s + ", 方案数: " + result);

            // 对于短字符串，可以手动验证
            if (s.length() <= 2 && s.contains("?")) {
                System.out.print("  具体方案: ");
                int count = 0;
                for (int i = 0; i < Math.pow(10, s.length()); i++) {
                    String candidate = String.format("%0" + s.length() + "d", i);
                    boolean match = true;
                    for (int j = 0; j < s.length(); j++) {
                        if (s.charAt(j) != '?' && s.charAt(j) != candidate.charAt(j)) {
                            match = false;
                            break;
                        }
                    }
                    if (match && i % 13 == 0) {
                        System.out.print(candidate + " ");
                        count++;
                    }
                }
                System.out.println("\n  手动计数: " + count);
            }
            System.out.println();
        }
    }

    /**
     * 工程化考量总结：
     * 1. 模运算：结果对 10^9+7 取模，避免溢出
     * 2. 状态设计：合理设计状态参数，减少状态数
     * 3. 两种实现：提供 DP 和 DFS 两种解法，便于理解
     * 4. 性能优化：使用迭代 DP 避免递归栈开销
     * 5. 边界处理：正确处理空字符串和全'?'情况
     *
```

```
     * 算法特色：
     * 1. 通配符处理：'?'可以替换为任意数字
     * 2. 模运算约束：结果必须能被 13 整除
     * 3. 动态规划：从高位到低位逐步计算
     * 4. 记忆化搜索：DFS 解法更符合数位 DP 传统
     */

    public static void main(String[] args) {
        // 运行功能测试
        testDigitsParade();

        // 运行性能测试
        performanceTest();

        // 调试模式
        debugDigitsParade();

        // 边界测试
        System.out.println("=== 边界测试 ===");
        System.out.println("空字符串: " + countDivisibleBy13(""));
        System.out.println("单个'?': " + countDivisibleBy13("?"));
        System.out.println("全'?': " + countDivisibleBy13("???"));
    }
}
```

==========

文件: Code15_DigitsParadeABC.py

==========

```
"""
AtCoder ABC135 D - Digits Parade
题目链接：https://atcoder.jp/contests/abc135/tasks/abc135_d

题目描述：
给定一个由数字和'?'组成的字符串 S，'?'可以替换成 0-9 的任意数字。
求有多少种替换方案使得结果能被 13 整除，结果对 10^9+7 取模。

解题思路：
1. 动态规划方法：使用 DP 框架，逐位确定数字
2. 状态设计需要记录：
   - 当前处理位置
   - 当前数字对 13 的余数
3. 关键点：'?'可以替换为 0-9 的任意数字
```

时间复杂度分析：
- 状态数：字符串长度 × 13 ≈ 10^5 × 13 = 1.3×10^6
- 每个状态处理最多 10 种选择
- 总复杂度：O(13×10^5) 在可接受范围内

空间复杂度分析：
- DP 数组：10^5 × 13 ≈ 1.3MB

最优解分析：
这是标准的最优解，利用 DP 处理模运算和通配符替换
"""

```
class DigitsParade:
    MOD = 10**9 + 7
    DIVISOR = 13

    def count_divisible_by_13(self, s: str) -> int:
        """
        计算有多少种替换方案使得结果能被 13 整除
        时间复杂度: O(n * 13)
        空间复杂度: O(n * 13)
        """
        n = len(s)

        # dp[i][r] 表示处理到第 i 位，当前余数为 r 的方案数
        dp = [[0] * self.DIVISOR for _ in range(n + 1)]
        dp[0][0] = 1  # 初始状态：余数为 0 有 1 种方案（空数字）

        # 从高位到低位动态规划
        for i in range(n):
            for r in range(self.DIVISOR):
                if dp[i][r] == 0:
                    continue

                if s[i] == '?':
                    # '?'可以替换为 0-9 的任意数字
                    for d in range(10):
                        new_r = (r * 10 + d) % self.DIVISOR
                        dp[i + 1][new_r] = (dp[i + 1][new_r] + dp[i][r]) % self.MOD
                else:
                    # 固定数字
                    d = int(s[i])
```

```
                    new_r = (r * 10 + d) % self.DIVISOR
                    dp[i + 1][new_r] = (dp[i + 1][new_r] + dp[i][r]) % self.MOD

        return dp[n][0]

    def count_divisible_by_13_dfs(self, s: str) -> int:
        """
        使用记忆化 DFS 的替代解法（更符合数位 DP 传统风格）
        时间复杂度: O(n * 13)
        空间复杂度: O(n * 13)
        """
        from functools import lru_cache

        @lru_cache(maxsize=None)
        def dfs(pos: int, remainder: int) -> int:
            """
            递归函数：计算从位置 pos 开始，当前余数为 remainder 的方案数
            """
            # 递归终止条件：处理完所有字符
            if pos == len(s):
                return 1 if remainder == 0 else 0

            count = 0

            if s[pos] == '?':
                # '?'可以替换为 0-9 的任意数字
                for d in range(10):
                    new_remainder = (remainder * 10 + d) % self.DIVISOR
                    count = (count + dfs(pos + 1, new_remainder)) % self.MOD
            else:
                # 固定数字
                d = int(s[pos])
                new_remainder = (remainder * 10 + d) % self.DIVISOR
                count = (count + dfs(pos + 1, new_remainder)) % self.MOD

            return count

        return dfs(0, 0)

def test_digits_parade():
    """
    单元测试函数
    """
```

```
    print("=== 测试 Digits Parade ===")

    dp = DigitsParade()

    # 测试用例 1: 简单情况
    s1 = "??"
    result1 = dp.count_divisible_by_13(s1)
    result1_dfs = dp.count_divisible_by_13_dfs(s1)
    print(f"输入: {s1}")
    print(f"DP 结果: {result1}")
    print(f"DFS 结果: {result1_dfs}")
    print(f"结果一致: {result1 == result1_dfs}")
    print("预期: 100 种组合中有几个能被 13 整除")
    print()

    # 测试用例 2: 固定数字
    s2 = "13"
    result2 = dp.count_divisible_by_13(s2)
    result2_dfs = dp.count_divisible_by_13_dfs(s2)
    print(f"输入: {s2}")
    print(f"DP 结果: {result2}")
    print(f"DFS 结果: {result2_dfs}")
    print(f"结果一致: {result2 == result2_dfs}")
    print("预期: 13 能被 13 整除，所以为 1")
    print()

    # 测试用例 3: 混合情况
    s3 = "1?2"
    result3 = dp.count_divisible_by_13(s3)
    result3_dfs = dp.count_divisible_by_13_dfs(s3)
    print(f"输入: {s3}")
    print(f"DP 结果: {result3}")
    print(f"DFS 结果: {result3_dfs}")
    print(f"结果一致: {result3 == result3_dfs}")
    print()

def performance_test():
    """
    性能测试函数
    """
    import time
    print("=== 性能测试 ===")
```

```
    dp = DigitsParade()

    # 生成测试用例
    long_string = "?" * 1000

    # 测试 DP 方法
    start_time_dp = time.time()
    result_dp = dp.count_divisible_by_13(long_string)
    end_time_dp = time.time()

    # 测试 DFS 方法
    start_time_dfs = time.time()
    result_dfs = dp.count_divisible_by_13_dfs(long_string)
    end_time_dfs = time.time()

    time_dp = (end_time_dp - start_time_dp) * 1000
    time_dfs = (end_time_dfs - start_time_dfs) * 1000

    print(f"字符串长度: {len(long_string)}")
    print(f"DP 方法耗时: {time_dp:.2f}ms")
    print(f"DFS 方法耗时: {time_dfs:.2f}ms")
    print(f"结果一致: {result_dp == result_dfs}")
    print()

def debug_digits_parade():
    """
    调试函数：验证特定字符串的替换方案
    """
    print("=== 调试 Digits Parade ===")

    dp = DigitsParade()

    test_cases = [
        "0", "1", "13", "26", "39", "52",
        "1?", "?3", "??", "1?3"
    ]

    for s in test_cases:
        result = dp.count_divisible_by_13(s)
        print(f"输入: {s}, 方案数: {result}")

        # 对于短字符串，可以手动验证
        if len(s) <= 2 and '?' in s:
```

```
            print("  具体方案: ", end="")
            count = 0
            max_num = 10 ** len(s)
            for i in range(max_num):
                candidate = str(i).zfill(len(s))

                match = True
                for j in range(len(s)):
                    if s[j] != '?' and s[j] != candidate[j]:
                        match = False
                        break

                if match and i % 13 == 0:
                    print(candidate, end=" ")
                    count += 1
            print(f"\n  手动计数: {count}")
        print()

def manual_verification():
    """
    手动验证函数：验证算法逻辑
    """
    print("=== 手动验证 ===")

    # 验证模运算逻辑
    test_cases = [
        ("0", True),    # 0 % 13 = 0 ✓
        ("13", True),   # 13 % 13 = 0 ✓
        ("26", True),   # 26 % 13 = 0 ✓
        ("1", False),   # 1 % 13 = 1 ✗
        ("27", False),  # 27 % 13 = 1 ✗
    ]

    dp = DigitsParade()

    for s, expected in test_cases:
        result = dp.count_divisible_by_13(s)
        actual = (result == 1)
        status = "✓" if actual == expected else "✗"
        print(f"{s}: 预期{expected}, 实际{actual} {status}")

"""
工程化考量总结：
```

1. 模运算：结果对 10^9+7 取模，避免溢出
2. 状态设计：合理设计状态参数，减少状态数
3. 两种实现：提供 DP 和 DFS 两种解法，便于理解
4. 性能优化：使用迭代 DP 避免递归栈开销
5. 边界处理：正确处理空字符串和全'?'情况

Python 语言特性利用：
1. 装饰器：使用@lru_cache 实现自动记忆化
2. 类型注解：提高代码可读性
3. 内置函数：使用 zfill 等简化代码
4. 动态类型：灵活处理各种情况

算法特色：
1. 通配符处理：'?'可以替换为任意数字
2. 模运算约束：结果必须能被 13 整除
3. 动态规划：从高位到低位逐步计算
4. 记忆化搜索：DFS 解法更符合数位 DP 传统

调试技巧：
1. 小范围验证：手动计算小范围结果进行对拍
2. 中间状态打印：添加调试信息打印中间状态
3. 边界测试：测试 0、1、边界值等特殊情况
4. 性能分析：使用 time 模块进行性能测试
"""

```
if __name__ == "__main__":
    # 运行功能测试
    test_digits_parade()

    # 运行性能测试
    performance_test()

    # 调试模式
    debug_digits_parade()

    # 手动验证
    manual_verification()

    # 边界测试
    print("=== 边界测试 ===")
    dp = DigitsParade()
    print(f"空字符串: {dp.count_divisible_by_13('')}")
    print(f"单个'?': {dp.count_divisible_by_13('?')}")
```

```
    print(f"全'?': {dp.count_divisible_by_13('???')}")

==========================================

文件: Code16_NumberWithoutConsecutiveOnes.cpp
==========================================
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <bitset>

using namespace std;

/**
 * 不含连续1的非负整数
 * 题目来源：LeetCode 600. 不含连续1的非负整数
 * 题目链接：https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/
 *
 * 题目描述：
 * 给定一个正整数 n，返回在 [0, n] 范围内不含连续1的非负整数的个数。
 *
 * 解题思路：
 * 1. 数位DP方法：使用数位DP框架，逐位确定二进制数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 前一位是否为1
 * 3. 关键点：当前位不能与前一位同时为1
 *
 * 时间复杂度分析：
 * - 状态数：二进制位数 × 2 × 2 ≈ 32 × 4 = 128
 * - 每个状态处理2种选择
 * - 总复杂度：O(256) 非常高效
 *
 * 空间复杂度分析：
 * - 记忆化数组：32 × 2 × 2 ≈ 128个状态
 *
 * 最优解分析：
 * 这是标准的最优解，利用数位DP处理二进制约束条件
 */

class Solution {
```

```
public:
    /**
     * 计算[0, n]中不含连续1的二进制数的个数
     * 时间复杂度: O(log n)
     * 空间复杂度: O(log n)
     */
    int findIntegers(int n) {
        if (n == 0) return 1;
        if (n == 1) return 2;

        // 将n转换为二进制字符串
        string binary = bitset<32>(n).to_string();
        // 去除前导零
        size_t start = binary.find('1');
        if (start == string::npos) {
            binary = "0";
        } else {
            binary = binary.substr(start);
        }

        int len = binary.length();

        // 记忆化数组：dp[pos][isLimit][lastIsOne]
        vector<vector<vector<int>>> dp(len, vector<vector<int>>(2, vector<int>(2, -1)));

        return dfs(binary, 0, true, false, dp);
    }

    /**
     * 数学方法（斐波那契数列）- 更高效的解法
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     *
     * 解题思路：
     * 1. 观察发现，不含连续1的二进制数个数满足斐波那契数列
     * 2. 对于k位二进制数，有效数字个数为fib(k+2)
     * 3. 利用这个性质可以快速计算
     */
    int findIntegersMath(int n) {
        if (n == 0) return 1;
        if (n == 1) return 2;

        // 预处理斐波那契数列
```

```
        vector<int> fib(32, 0);
        fib[0] = 1;
        fib[1] = 2;
        for (int i = 2; i < 32; i++) {
            fib[i] = fib[i-1] + fib[i-2];
        }

        string binary = bitset<32>(n).to_string();
        size_t start = binary.find('1');
        if (start == string::npos) {
            binary = "0";
        } else {
            binary = binary.substr(start);
        }

        int len = binary.length();
        int ans = 0;
        bool prevBit = false; // 前一位是否为1

        for (int i = 0; i < len; i++) {
            if (binary[i] == '1') {
                // 如果当前位为1，可以选择填0，后面位可以任意填
                ans += fib[len - i - 1];

                // 如果前一位也是1，说明出现了连续1，后面的数字都不满足条件
                if (prevBit) {
                    return ans;
                }
                prevBit = true;
            } else {
                prevBit = false;
            }
        }

        // 加上n本身（如果n本身满足条件）
        return ans + 1;
    }

private:
    /**
     * 数位DP递归函数（二进制版本）
     *
     * @param binary 二进制字符串
```

```
     * @param pos 当前处理位置
     * @param isLimit 是否受到上界限制
     * @param lastIsOne 前一位是否为1
     * @param dp 记忆化数组
     * @return 满足条件的数字个数
     */
    int dfs(const string& binary, int pos, bool isLimit, bool lastIsOne,
            vector<vector<vector<int>>>& dp) {
        // 递归终止条件：处理完所有二进制位
        if (pos == binary.length()) {
            return 1; // 成功构造一个有效数字
        }

        // 记忆化搜索
        int limitIndex = isLimit ? 1 : 0;
        int lastIndex = lastIsOne ? 1 : 0;
        if (dp[pos][limitIndex][lastIndex] != -1) {
            return dp[pos][limitIndex][lastIndex];
        }

        int ans = 0;

        // 确定当前位可选数字范围（二进制只有0和1）
        int up = isLimit ? (binary[pos] - '0') : 1;

        // 枚举当前位可选数字
        for (int d = 0; d <= up; d++) {
            // 检查约束条件：不能有连续的1
            if (lastIsOne && d == 1) {
                continue; // 连续1，跳过
            }

            // 递归处理下一位
            ans += dfs(binary, pos + 1, isLimit && (d == up), d == 1, dp);
        }

        // 记忆化存储
        dp[pos][limitIndex][lastIndex] = ans;
        return ans;
    }
};

/**
```

```
 * 单元测试函数
 */
void testFindIntegers() {
    cout << "=== 测试不含连续 1 的非负整数 ===" << endl;

    Solution sol;

    // 测试用例 1: 小数字
    int n1 = 5;
    int result1 = sol.findIntegers(n1);
    int result1Math = sol.findIntegersMath(n1);
    cout << "n = " << n1 << endl;
    cout << "DP 结果: " << result1 << endl;
    cout << "数学结果: " << result1Math << endl;
    cout << "结果一致: " << (result1 == result1Math) << endl;
    cout << "预期: [0,5]中不含连续 1 的数字有 0,1,2,4,5 共 5 个" << endl;
    cout << endl;

    // 测试用例 2: 中等数字
    int n2 = 10;
    int result2 = sol.findIntegers(n2);
    int result2Math = sol.findIntegersMath(n2);
    cout << "n = " << n2 << endl;
    cout << "DP 结果: " << result2 << endl;
    cout << "数学结果: " << result2Math << endl;
    cout << "结果一致: " << (result2 == result2Math) << endl;
    cout << endl;

    // 测试用例 3: 边界情况
    int n3 = 1;
    int result3 = sol.findIntegers(n3);
    int result3Math = sol.findIntegersMath(n3);
    cout << "n = " << n3 << endl;
    cout << "DP 结果: " << result3 << endl;
    cout << "数学结果: " << result3Math << endl;
    cout << "结果一致: " << (result3 == result3Math) << endl;
    cout << "预期: [0,1]中所有数字都满足，共 2 个" << endl;
    cout << endl;
}

/**
 * 性能测试函数
 */
```

```
void performanceTest() {
    cout << "=== 性能测试 ===" << endl;

    Solution sol;
    vector<int> testCases = {100, 1000, 10000, 100000, 1000000, 10000000};

    for (int n : testCases) {
        auto startTimeDP = chrono::high_resolution_clock::now();
        int resultDP = sol.findIntegers(n);
        auto endTimeDP = chrono::high_resolution_clock::now();

        auto startTimeMath = chrono::high_resolution_clock::now();
        int resultMath = sol.findIntegersMath(n);
        auto endTimeMath = chrono::high_resolution_clock::now();

        auto timeDP = chrono::duration_cast<chrono::nanoseconds>(endTimeDP - startTimeDP);
        auto timeMath = chrono::duration_cast<chrono::nanoseconds>(endTimeMath - startTimeMath);

        cout << "n = " << n << endl;
        cout << "DP 方法耗时: " << timeDP.count() << "ns" << endl;
        cout << "数学方法耗时: " << timeMath.count() << "ns" << endl;
        cout << "加速比: " << (double)timeDP.count() / timeMath.count() << "倍" << endl;
        cout << "结果一致: " << (resultDP == resultMath) << endl;
        cout << endl;
    }
}

/**
 * 调试函数：验证特定范围内的结果
 */
void debugFindIntegers() {
    cout << "=== 调试不含连续 1 的非负整数 ===" << endl;

    Solution sol;

    for (int n = 0; n <= 20; n++) {
        int count = 0;
        string validNumbers;

        for (int i = 0; i <= n; i++) {
            // 检查二进制表示是否包含"11"
            bool hasConsecutiveOnes = false;
            int temp = i;
```

```
            int lastBit = 0;
            while (temp > 0) {
                int bit = temp & 1;
                if (bit == 1 && lastBit == 1) {
                    hasConsecutiveOnes = true;
                    break;
                }
                lastBit = bit;
                temp >>= 1;
            }

            if (!hasConsecutiveOnes) {
                count++;
                if (validNumbers.length() < 100) { // 限制输出长度
                    validNumbers += to_string(i) + " ";
                }
            }
        }

        int dpResult = sol.findIntegers(n);
        int mathResult = sol.findIntegersMath(n);

        cout << "n = " << n << ", 有效数字个数: " << count << endl;
        cout << "DP 结果: " << dpResult << ", 数学结果: " << mathResult << endl;
        cout << "结果一致: " << (count == dpResult && dpResult == mathResult) << endl;

        if (n <= 10) {
            cout << "有效数字: " << validNumbers << endl;
        }
        cout << endl;
    }
}

/**
 * 工程化考量总结：
 * 1. 两种解法：提供 DP 和数学两种解法，便于理解和选择
 * 2. 性能优化：数学方法更高效，DP 方法更通用
 * 3. 边界处理：正确处理 n=0 和 n=1 的情况
 * 4. 状态设计：合理设计状态参数，减少状态数
 * 5. 二进制处理：使用 bitset 进行二进制转换
 *
 * 算法特色：
 * 1. 二进制处理：针对二进制数的特殊约束
```

```
 * 2. 斐波那契性质：发现并利用数学规律
 * 3. 记忆化搜索：DP 解法避免重复计算
 * 4. 提前终止：数学解法在发现连续 1 时提前返回
 *
 * C++实现特点：
 * 1. 使用 bitset 进行二进制转换
 * 2. 使用 vector 代替原生数组
 * 3. 提供完整的测试框架
 * 4. 使用 chrono 进行性能测试
 */

int main() {
    // 运行功能测试
    testFindIntegers();

    // 运行性能测试
    performanceTest();

    // 调试模式
    debugFindIntegers();

    // 边界测试
    cout << "=== 边界测试 ===" << endl;
    Solution sol;
    cout << "n=0: " << sol.findIntegers(0) << endl;
    cout << "n=1: " << sol.findIntegers(1) << endl;
    cout << "n=2: " << sol.findIntegers(2) << endl;
    cout << "n=3: " << sol.findIntegers(3) << endl;

    return 0;
}

==============================================

文件: Code16_NumberWithoutConsecutiveOnes.java
==============================================
package class084;

/**
 * 不含连续 1 的非负整数
 * 题目来源：LeetCode 600. 不含连续 1 的非负整数
 * 题目链接：https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/
 *
```

```
 * 题目描述：
 * 给定一个正整数 n，返回在 [0, n] 范围内不含连续 1 的非负整数的个数。
 *
 * 解题思路：
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定二进制数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 前一位是否为 1
 * 3. 关键点：当前位不能与前一位同时为 1
 *
 * 时间复杂度分析：
 * - 状态数：二进制位数 × 2 × 2 ≈ 32 × 4 = 128
 * - 每个状态处理 2 种选择
 * - 总复杂度：O(256) 非常高效
 *
 * 空间复杂度分析：
 * - 记忆化数组：32 × 2 × 2 ≈ 128 个状态
 *
 * 最优解分析：
 * 这是标准的最优解，利用数位 DP 处理二进制约束条件
 */

public class Code16_NumberWithoutConsecutiveOnes {

    /**
     * 计算[0, n]中不含连续 1 的二进制数的个数
     * 时间复杂度: O(log n)
     * 空间复杂度: O(log n)
     */
    public static int findIntegers(int n) {
        if (n == 0) return 1;
        if (n == 1) return 2;

        // 将 n 转换为二进制字符串
        String binary = Integer.toBinaryString(n);
        int len = binary.length();

        // 记忆化数组：dp[pos][isLimit][lastIsOne]
        Integer[][][] dp = new Integer[len][2][2];

        return dfs(binary.toCharArray(), 0, true, false, dp);
    }
```

```
    /**
     * 数位DP递归函数（二进制版本）
     *
     * @param bits 二进制位数组
     * @param pos 当前处理位置
     * @param isLimit 是否受到上界限制
     * @param lastIsOne 前一位是否为1
     * @param dp 记忆化数组
     * @return 满足条件的数字个数
     */
    private static int dfs(char[] bits, int pos, boolean isLimit, boolean lastIsOne,
Integer[][][] dp) {
        // 递归终止条件：处理完所有二进制位
        if (pos == bits.length) {
            return 1; // 成功构造一个有效数字
        }

        // 记忆化搜索
        int limitIndex = isLimit ? 1 : 0;
        int lastIndex = lastIsOne ? 1 : 0;
        if (dp[pos][limitIndex][lastIndex] != null) {
            return dp[pos][limitIndex][lastIndex];
        }

        int ans = 0;

        // 确定当前位可选数字范围（二进制只有0和1）
        int up = isLimit ? (bits[pos] - '0') : 1;

        // 枚举当前位可选数字
        for (int d = 0; d <= up; d++) {
            // 检查约束条件：不能有连续的1
            if (lastIsOne && d == 1) {
                continue; // 连续1，跳过
            }

            // 递归处理下一位
            ans += dfs(bits, pos + 1, isLimit && (d == up), d == 1, dp);
        }

        // 记忆化存储
        dp[pos][limitIndex][lastIndex] = ans;
```

```
        return ans;
    }

    /**
     * 数学方法（斐波那契数列）- 更高效的解法
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     *
     * 解题思路：
     * 1. 观察发现，不含连续1的二进制数个数满足斐波那契数列
     * 2. 对于k位二进制数，有效数字个数为fib(k+2)
     * 3. 利用这个性质可以快速计算
     */
    public static int findIntegersMath(int n) {
        if (n == 0) return 1;
        if (n == 1) return 2;

        // 预处理斐波那契数列
        int[] fib = new int[32];
        fib[0] = 1;
        fib[1] = 2;
        for (int i = 2; i < 32; i++) {
            fib[i] = fib[i-1] + fib[i-2];
        }

        String binary = Integer.toBinaryString(n);
        int len = binary.length();
        int ans = 0;
        boolean prevBit = false; // 前一位是否为1

        for (int i = 0; i < len; i++) {
            if (binary.charAt(i) == '1') {
                // 如果当前位为1，可以选择填0，后面位可以任意填
                ans += fib[len - i - 1];

                // 如果前一位也是1，说明出现了连续1，后面的数字都不满足条件
                if (prevBit) {
                    return ans;
                }
                prevBit = true;
            } else {
                prevBit = false;
            }
```

```
    }

    // 加上 n 本身（如果 n 本身满足条件）
    return ans + 1;
}

/**
 * 单元测试函数
 */
public static void testFindIntegers() {
    System.out.println("=== 测试不含连续 1 的非负整数 ===");

    // 测试用例 1: 小数字
    int n1 = 5;
    int result1 = findIntegers(n1);
    int result1Math = findIntegersMath(n1);
    System.out.println("n = " + n1);
    System.out.println("DP 结果: " + result1);
    System.out.println("数学结果: " + result1Math);
    System.out.println("结果一致: " + (result1 == result1Math));
    System.out.println("预期: [0,5]中不含连续 1 的数字有 0,1,2,4,5 共 5 个");
    System.out.println();

    // 测试用例 2: 中等数字
    int n2 = 10;
    int result2 = findIntegers(n2);
    int result2Math = findIntegersMath(n2);
    System.out.println("n = " + n2);
    System.out.println("DP 结果: " + result2);
    System.out.println("数学结果: " + result2Math);
    System.out.println("结果一致: " + (result2 == result2Math));
    System.out.println();

    // 测试用例 3: 边界情况
    int n3 = 1;
    int result3 = findIntegers(n3);
    int result3Math = findIntegersMath(n3);
    System.out.println("n = " + n3);
    System.out.println("DP 结果: " + result3);
    System.out.println("数学结果: " + result3Math);
    System.out.println("结果一致: " + (result3 == result3Math));
    System.out.println("预期: [0,1]中所有数字都满足，共 2 个");
    System.out.println();
```

```
}

/**
 * 性能测试函数
 */
public static void performanceTest() {
    System.out.println("=== 性能测试 ===");

    int[] testCases = {100, 1000, 10000, 100000, 1000000, 10000000};

    for (int n : testCases) {
        long startTimeDP = System.nanoTime();
        int resultDP = findIntegers(n);
        long endTimeDP = System.nanoTime();

        long startTimeMath = System.nanoTime();
        int resultMath = findIntegersMath(n);
        long endTimeMath = System.nanoTime();

        long timeDP = endTimeDP - startTimeDP;
        long timeMath = endTimeMath - startTimeMath;

        System.out.println("n = " + n);
        System.out.println("DP 方法耗时: " + timeDP + "ns");
        System.out.println("数学方法耗时: " + timeMath + "ns");
        System.out.println("加速比: " + (double)timeDP / timeMath + "倍");
        System.out.println("结果一致: " + (resultDP == resultMath));
        System.out.println();
    }
}

/**
 * 调试函数：验证特定范围内的结果
 */
public static void debugFindIntegers() {
    System.out.println("=== 调试不含连续 1 的非负整数 ===");

    for (int n = 0; n <= 20; n++) {
        int count = 0;
        StringBuilder validNumbers = new StringBuilder();

        for (int i = 0; i <= n; i++) {
            String binary = Integer.toBinaryString(i);
```

```
            if (!binary.contains("11")) {
                count++;
                if (validNumbers.length() < 50) { // 限制输出长度
                    validNumbers.append(i).append(" ");
                }
            }
        }

        int dpResult = findIntegers(n);
        int mathResult = findIntegersMath(n);

        System.out.println("n = " + n + ", 有效数字个数: " + count);
        System.out.println("DP 结果: " + dpResult + ", 数学结果: " + mathResult);
        System.out.println("结果一致: " + (count == dpResult && dpResult == mathResult));

        if (n <= 10) {
            System.out.println("有效数字: " + validNumbers);
        }
        System.out.println();
    }
}

/**
 * 工程化考量总结:
 * 1. 两种解法：提供 DP 和数学两种解法，便于理解和选择
 * 2. 性能优化：数学方法更高效，DP 方法更通用
 * 3. 边界处理：正确处理 n=0 和 n=1 的情况
 * 4. 状态设计：合理设计状态参数，减少状态数
 * 5. 测试覆盖：全面的测试用例
 *
 * 算法特色:
 * 1. 二进制处理：针对二进制数的特殊约束
 * 2. 斐波那契性质：发现并利用数学规律
 * 3. 记忆化搜索：DP 解法避免重复计算
 * 4. 提前终止：数学解法在发现连续 1 时提前返回
 */

public static void main(String[] args) {
    // 运行功能测试
    testFindIntegers();

    // 运行性能测试
    performanceTest();
```

```
        // 调试模式
        debugFindIntegers();

        // 边界测试
        System.out.println("=== 边界测试 ===");
        System.out.println("n=0: " + findIntegers(0));
        System.out.println("n=1: " + findIntegers(1));
        System.out.println("n=2: " + findIntegers(2));
        System.out.println("n=3: " + findIntegers(3));
    }
}
```

==========

文件: Code16_NumberWithoutConsecutiveOnes.py
==========
"""
不含连续 1 的非负整数
题目来源：LeetCode 600. 不含连续 1 的非负整数
题目链接：https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/

题目描述：
给定一个正整数 n，返回在 [0, n] 范围内不含连续 1 的非负整数的个数。

解题思路：
1. 数位 DP 方法：使用数位 DP 框架，逐位确定二进制数字
2. 状态设计需要记录：
   - 当前处理位置
   - 是否受上界限制
   - 前一位是否为 1
3. 关键点：当前位不能与前一位同时为 1

时间复杂度分析：
- 状态数：二进制位数 × 2 × 2 ≈ 32 × 4 = 128
- 每个状态处理 2 种选择
- 总复杂度：O(256) 非常高效

空间复杂度分析：
- 记忆化数组：32 × 2 × 2 ≈ 128 个状态

最优解分析：
这是标准的最优解，利用数位 DP 处理二进制约束条件

```
"""

class Solution:
    def findIntegers(self, n: int) -> int:
        """
        计算[0, n]中不含连续1的二进制数的个数
        时间复杂度: O(log n)
        空间复杂度: O(log n)
        """
        if n == 0:
            return 1
        if n == 1:
            return 2

        # 将n转换为二进制字符串
        binary = bin(n)[2:]

        from functools import lru_cache

        @lru_cache(maxsize=None)
        def dfs(pos: int, is_limit: bool, last_is_one: bool) -> int:
            """
            数位DP递归函数（二进制版本）

            Args:
                pos: 当前处理位置
                is_limit: 是否受到上界限制
                last_is_one: 前一位是否为1

            Returns:
                满足条件的数字个数
            """
            # 递归终止条件：处理完所有二进制位
            if pos == len(binary):
                return 1  # 成功构造一个有效数字

            ans = 0

            # 确定当前位可选数字范围（二进制只有0和1）
            up = int(binary[pos]) if is_limit else 1

            # 枚举当前位可选数字
            for d in range(0, up + 1):
```

```
            # 检查约束条件：不能有连续的 1
            if last_is_one and d == 1:
                continue  # 连续 1，跳过

            # 递归处理下一位
            ans += dfs(pos + 1, is_limit and d == up, d == 1)

        return ans

    # 从第 0 位开始，初始状态：受限制、前一位不是 1
    return dfs(0, True, False)

def findIntegersMath(self, n: int) -> int:
    """
    数学方法（斐波那契数列）- 更高效的解法
    时间复杂度: O(log n)
    空间复杂度: O(1)

    解题思路：
    1. 观察发现，不含连续 1 的二进制数个数满足斐波那契数列
    2. 对于 k 位二进制数，有效数字个数为 fib(k+2)
    3. 利用这个性质可以快速计算
    """
    if n == 0:
        return 1
    if n == 1:
        return 2

    # 预处理斐波那契数列
    fib = [1, 2]
    for i in range(2, 32):
        fib.append(fib[i-1] + fib[i-2])

    binary = bin(n)[2:]
    ans = 0
    prev_bit = False  # 前一位是否为 1

    for i in range(len(binary)):
        if binary[i] == '1':
            # 如果当前位为 1，可以选择填 0，后面位可以任意填
            ans += fib[len(binary) - i - 1]

            # 如果前一位也是 1，说明出现了连续 1，后面的数字都不满足条件
```

```
                if prev_bit:
                    return ans
                prev_bit = True
            else:
                prev_bit = False

        # 加上 n 本身（如果 n 本身满足条件）
        return ans + 1

def test_find_integers():
    """
    单元测试函数
    """
    print("=== 测试不含连续 1 的非负整数 ===")

    sol = Solution()

    # 测试用例 1: 小数字
    n1 = 5
    result1 = sol.findIntegers(n1)
    result1_math = sol.findIntegersMath(n1)
    print(f"n = {n1}")
    print(f"DP 结果: {result1}")
    print(f"数学结果: {result1_math}")
    print(f"结果一致: {result1 == result1_math}")
    print("预期: [0,5]中不含连续 1 的数字有 0,1,2,4,5 共 5 个")
    print()

    # 测试用例 2: 中等数字
    n2 = 10
    result2 = sol.findIntegers(n2)
    result2_math = sol.findIntegersMath(n2)
    print(f"n = {n2}")
    print(f"DP 结果: {result2}")
    print(f"数学结果: {result2_math}")
    print(f"结果一致: {result2 == result2_math}")
    print()

    # 测试用例 3: 边界情况
    n3 = 1
    result3 = sol.findIntegers(n3)
    result3_math = sol.findIntegersMath(n3)
    print(f"n = {n3}")
```

```
    print(f"DP 结果: {result3}")
    print(f"数学结果: {result3_math}")
    print(f"结果一致: {result3 == result3_math}")
    print("预期: [0,1]中所有数字都满足，共 2 个")
    print()

def performance_test():
    """
    性能测试函数
    """
    import time
    print("=== 性能测试 ===")

    sol = Solution()
    test_cases = [100, 1000, 10000, 100000, 1000000, 10000000]

    for n in test_cases:
        # 测试 DP 方法
        start_time_dp = time.time()
        result_dp = sol.findIntegers(n)
        end_time_dp = time.time()

        # 测试数学方法
        start_time_math = time.time()
        result_math = sol.findIntegersMath(n)
        end_time_math = time.time()

        time_dp = (end_time_dp - start_time_dp) * 1e9  # 转换为纳秒
        time_math = (end_time_math - start_time_math) * 1e9  # 转换为纳秒

        print(f"n = {n}")
        print(f"DP 方法耗时: {time_dp:.0f}ns")
        print(f"数学方法耗时: {time_math:.0f}ns")
        if time_math > 0:
            print(f"加速比: {time_dp/time_math:.2f}倍")
        print(f"结果一致: {result_dp == result_math}")
        print()

def debug_find_integers():
    """
    调试函数：验证特定范围内的结果
    """
    print("=== 调试不含连续 1 的非负整数 ===")
```

```
    sol = Solution()

    for n in range(0, 21):
        count = 0
        valid_numbers = []

        for i in range(0, n + 1):
            # 检查二进制表示是否包含"11"
            binary_str = bin(i)[2:]
            if "11" not in binary_str:
                count += 1
                if len(valid_numbers) < 10:  # 限制输出长度
                    valid_numbers.append(i)

        dp_result = sol.findIntegers(n)
        math_result = sol.findIntegersMath(n)

        print(f"n = {n}, 有效数字个数: {count}")
        print(f"DP 结果: {dp_result}, 数学结果: {math_result}")
        print(f"结果一致: {count == dp_result == math_result}")

        if n <= 10:
            print(f"有效数字: {valid_numbers}")
        print()

def manual_verification():
    """
    手动验证函数：验证算法逻辑
    """
    print("=== 手动验证 ===")

    # 验证斐波那契性质
    fib = [1, 2]
    for i in range(2, 10):
        fib.append(fib[i-1] + fib[i-2])

    print("斐波那契数列（表示 k 位二进制数的有效数字个数）:")
    for i in range(10):
        print(f"fib({i}) = {fib[i]}")

    # 验证小范围结果
    test_cases = [
```

```
        (0, 1),  # 只有0
        (1, 2),  # 0,1
        (2, 3),  # 0,1,2
        (3, 4),  # 0,1,2,3? 3的二进制是11，应该排除
        (4, 5),  # 0,1,2,4,5
    ]

    sol = Solution()

    for n, expected in test_cases:
        actual = sol.findIntegers(n)
        status = "✓" if actual == expected else "✗"
        print(f"n={n}: 预期{expected}, 实际{actual} {status}")

"""
工程化考量总结：
1. 两种解法：提供DP和数学两种解法，便于理解和选择
2. 性能优化：数学方法更高效，DP方法更通用
3. 边界处理：正确处理n=0和n=1的情况
4. 状态设计：合理设计状态参数，减少状态数
5. 测试覆盖：全面的测试用例

Python语言特性利用：
1. 装饰器：使用@lru_cache实现自动记忆化
2. 内置函数：使用bin()进行二进制转换
3. 动态类型：灵活处理各种边界情况
4. 列表推导：简化代码逻辑

算法特色：
1. 二进制处理：针对二进制数的特殊约束
2. 斐波那契性质：发现并利用数学规律
3. 记忆化搜索：DP解法避免重复计算
4. 提前终止：数学解法在发现连续1时提前返回

调试技巧：
1. 小范围验证：手动计算小范围结果进行对拍
2. 中间状态打印：添加调试信息打印中间状态
3. 边界测试：测试0、1、边界值等特殊情况
4. 性能分析：使用time模块进行性能测试
"""

if __name__ == "__main__":
    # 运行功能测试
```

```
    test_find_integers()

    # 运行性能测试
    performance_test()

    # 调试模式
    debug_find_integers()

    # 手动验证
    manual_verification()

    # 边界测试
    print("=== 边界测试 ===")
    sol = Solution()
    print(f"n=0: {sol.findIntegers(0)}")
    print(f"n=1: {sol.findIntegers(1)}")
    print(f"n=2: {sol.findIntegers(2)}")
    print(f"n=3: {sol.findIntegers(3)}")
```

```
================================================
```

文件: Code17_NumberWithRepeatedDigits.java

```
================================================
package class084;

import java.util.*;

/**
 * 至少有 1 位重复的数字
 * 题目来源：LeetCode 1012. 至少有 1 位重复的数字
 * 题目链接：https://leetcode.cn/problems/numbers-with-repeated-digits/
 *
 * 题目描述：
 * 给定正整数 n，返回在 [1, n] 范围内具有至少 1 位重复数字的正整数的个数。
 *
 * 解题思路：
 * 1. 数位DP方法：使用数位DP框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 是否已开始填数字
 *    - 已使用数字的位掩码
 * 3. 关键点：统计至少有一位重复的数字，可以转化为统计所有数字减去没有重复数字的数字
```

```
 *
 * 时间复杂度分析：
 * - 状态数：log n × 2 × 2 × 2^10 ≈ 10 × 4 × 1024 = 40,960
 * - 每个状态处理最多 10 种选择
 * - 总复杂度：O(400,000) 在可接受范围内
 *
 * 空间复杂度分析：
 * - 记忆化数组：10 × 2 × 2 × 1024 ≈ 40,960 个状态
 *
 * 最优解分析：
 * 这是标准的最优解，利用补集思想将问题转化为求没有重复数字的数字个数
 */

public class Code17_NumberWithRepeatedDigits {

    /**
     * 计算[1, n]中至少有一位重复数字的数字个数
     * 时间复杂度: O(log n × 2^10)
     * 空间复杂度: O(log n × 2^10)
     */
    public static int numDupDigitsAtMostN(int n) {
        if (n <= 10) return 0; // 1-10 中没有重复数字的数字

        // 总数字个数减去没有重复数字的数字个数
        int totalNumbers = n;
        int numbersWithUniqueDigits = countNumbersWithUniqueDigits(n);

        return totalNumbers - numbersWithUniqueDigits;
    }

    /**
     * 计算[0, n]中没有重复数字的数字个数
     */
    private static int countNumbersWithUniqueDigits(int n) {
        if (n == 0) return 1;

        // 将数字转换为字符数组
        char[] digits = String.valueOf(n).toCharArray();
        int len = digits.length;

        // 记忆化数组：dp[pos][isLimit][isNum][mask]
        Integer[][][][] dp = new Integer[len][2][2][1 << 10];
```

```
    return dfs(digits, 0, true, false, 0, dp);
}

/**
 * 数位DP递归函数
 *
 * @param digits 数字的字符数组表示
 * @param pos 当前处理位置
 * @param isLimit 是否受到上界限制
 * @param isNum 是否已开始填数字
 * @param mask 已使用数字的位掩码
 * @param dp 记忆化数组
 * @return 满足条件的数字个数
 */
private static int dfs(char[] digits, int pos, boolean isLimit, boolean isNum,
                       int mask, Integer[][][][] dp) {
    // 递归终止条件：处理完所有数位
    if (pos == digits.length) {
        return isNum ? 1 : 0; // 已填数字才计数
    }

    // 记忆化搜索
    if (!isLimit && isNum && dp[pos][0][0][mask] != null) {
        return dp[pos][0][0][mask];
    }

    int ans = 0;

    // 处理前导零：可以选择跳过当前位
    if (!isNum) {
        ans += dfs(digits, pos + 1, false, false, mask, dp);
    }

    // 确定当前位可选数字范围
    int up = isLimit ? (digits[pos] - '0') : 9;
    int start = isNum ? 0 : 1; // 处理前导零

    // 枚举当前位可选数字
    for (int d = start; d <= up; d++) {
        // 检查数字d是否已被使用
        if ((mask & (1 << d)) != 0) {
            continue; // 数字重复，跳过
        }
```

```
            // 递归处理下一位
            ans += dfs(digits, pos + 1, isLimit && (d == up), true,
                       mask | (1 << d), dp);
        }

        // 记忆化存储
        if (!isLimit && isNum) {
            dp[pos][0][0][mask] = ans;
        }

        return ans;
    }

    /**
     * 数学方法（排列组合）- 更高效的解法
     * 时间复杂度: O(10^2)
     * 空间复杂度: O(1)
     *
     * 解题思路：
     * 1. 对于 n 位数，没有重复数字的数字个数可以用排列组合计算
     * 2. 总个数 = 1 位数个数 + 2 位数个数 + ... + n 位数个数
     * 3. k 位数个数 = 9 × 9 × 8 × ... × (11-k)
     */
    public static int numDupDigitsAtMostNMath(int n) {
        if (n <= 10) return 0;

        // 计算没有重复数字的数字个数
        int uniqueCount = countUniqueDigitsMath(n);
        return n - uniqueCount;
    }

    private static int countUniqueDigitsMath(int n) {
        if (n == 0) return 1;

        // 将 n 转换为字符串获取位数
        String s = String.valueOf(n);
        int len = s.length();

        // 计算位数小于 len 的数字个数
        int count = 0;
        for (int i = 1; i < len; i++) {
            count += countUniqueDigitsOfLength(i);
```

```
    }

    // 计算位数等于 len 且小于等于 n 的数字个数
    count += countUniqueDigitsEqualToLength(s);

    return count;
}

/**
 * 计算恰好 k 位数且没有重复数字的数字个数
 */
private static int countUniqueDigitsOfLength(int k) {
    if (k == 0) return 1;
    if (k == 1) return 9;

    int count = 9; // 第一位有 9 种选择（1-9）
    int available = 9; // 剩余可用数字个数

    for (int i = 2; i <= k; i++) {
        count *= available;
        available--;
    }

    return count;
}

/**
 * 计算位数等于给定字符串长度且没有重复数字的数字个数
 */
private static int countUniqueDigitsEqualToLength(String s) {
    int len = s.length();
    boolean[] used = new boolean[10];
    int count = 0;

    // 逐位处理
    for (int i = 0; i < len; i++) {
        int currentDigit = s.charAt(i) - '0';

        // 确定当前位可以选择的数字范围
        int start = (i == 0) ? 1 : 0; // 第一位不能为 0
        for (int d = start; d < currentDigit; d++) {
            if (!used[d]) {
                // 计算剩余位数的排列数
```

```
                count += countPermutations(10 - countUsed(used) - 1, len - i - 1);
            }
        }

        // 如果当前数字已被使用，后面的数字都不满足条件
        if (used[currentDigit]) {
            break;
        }

        used[currentDigit] = true;

        // 如果是最后一位且所有数字都不重复，计数加 1
        if (i == len - 1) {
            count++;
        }
    }

    return count;
}

/**
 * 计算排列数：从 m 个元素中取 n 个的排列数
 */
private static int countPermutations(int m, int n) {
    if (n == 0) return 1;
    if (n > m) return 0;

    int result = 1;
    for (int i = 0; i < n; i++) {
        result *= (m - i);
    }
    return result;
}

/**
 * 统计已使用的数字个数
 */
private static int countUsed(boolean[] used) {
    int count = 0;
    for (boolean u : used) {
        if (u) count++;
    }
    return count;
```

```
}

/**
 * 单元测试函数
 */
public static void testNumDupDigitsAtMostN() {
    System.out.println("=== 测试至少有 1 位重复的数字 ===");

    // 测试用例 1: 小数字
    int n1 = 20;
    int result1 = numDupDigitsAtMostN(n1);
    int result1Math = numDupDigitsAtMostNMath(n1);
    System.out.println("n = " + n1);
    System.out.println("DP 结果: " + result1);
    System.out.println("数学结果: " + result1Math);
    System.out.println("结果一致: " + (result1 == result1Math));
    System.out.println("预期: 11, 22 等数字有重复");
    System.out.println();

    // 测试用例 2: 中等数字
    int n2 = 100;
    int result2 = numDupDigitsAtMostN(n2);
    int result2Math = numDupDigitsAtMostNMath(n2);
    System.out.println("n = " + n2);
    System.out.println("DP 结果: " + result2);
    System.out.println("数学结果: " + result2Math);
    System.out.println("结果一致: " + (result2 == result2Math));
    System.out.println();

    // 测试用例 3: 边界情况
    int n3 = 10;
    int result3 = numDupDigitsAtMostN(n3);
    int result3Math = numDupDigitsAtMostNMath(n3);
    System.out.println("n = " + n3);
    System.out.println("DP 结果: " + result3);
    System.out.println("数学结果: " + result3Math);
    System.out.println("结果一致: " + (result3 == result3Math));
    System.out.println("预期: 1-10 中没有重复数字");
    System.out.println();
}

/**
 * 性能测试函数
```

```
 */
public static void performanceTest() {
    System.out.println("=== 性能测试 ===");

    int[] testCases = {100, 1000, 10000, 100000, 1000000, 10000000};

    for (int n : testCases) {
        long startTimeDP = System.nanoTime();
        int resultDP = numDupDigitsAtMostN(n);
        long endTimeDP = System.nanoTime();

        long startTimeMath = System.nanoTime();
        int resultMath = numDupDigitsAtMostNMath(n);
        long endTimeMath = System.nanoTime();

        long timeDP = endTimeDP - startTimeDP;
        long timeMath = endTimeMath - startTimeMath;

        System.out.println("n = " + n);
        System.out.println("DP 方法耗时: " + timeDP + "ns");
        System.out.println("数学方法耗时: " + timeMath + "ns");
        if (timeMath > 0) {
            System.out.println("加速比: " + (double)timeDP / timeMath + "倍");
        }
        System.out.println("结果一致: " + (resultDP == resultMath));
        System.out.println();
    }
}

/**
 * 调试函数：验证特定范围内的结果
 */
public static void debugNumDupDigitsAtMostN() {
    System.out.println("=== 调试至少有 1 位重复的数字 ===");

    for (int n = 1; n <= 30; n++) {
        int manualCount = 0;
        StringBuilder duplicateNumbers = new StringBuilder();

        for (int i = 1; i <= n; i++) {
            if (hasDuplicateDigits(i)) {
                manualCount++;
                if (duplicateNumbers.length() < 50) {
```

```
                    duplicateNumbers.append(i).append(" ");
                }
            }
        }

        int dpResult = numDupDigitsAtMostN(n);
        int mathResult = numDupDigitsAtMostNMath(n);

        System.out.println("n = " + n + ", 重复数字个数: " + manualCount);
        System.out.println("DP 结果: " + dpResult + ", 数学结果: " + mathResult);
        System.out.println("结果一致: " + (manualCount == dpResult && dpResult ==
mathResult));

        if (n <= 20) {
            System.out.println("重复数字: " + duplicateNumbers);
        }
        System.out.println();
    }
}

/**
 * 检查数字是否有重复数字
 */
private static boolean hasDuplicateDigits(int num) {
    if (num < 10) return false;

    boolean[] digits = new boolean[10];
    while (num > 0) {
        int digit = num % 10;
        if (digits[digit]) {
            return true;
        }
        digits[digit] = true;
        num /= 10;
    }
    return false;
}

/**
 * 工程化考量总结：
 * 1. 补集思想：将问题转化为求补集，简化问题
 * 2. 两种解法：提供 DP 和数学两种解法，便于理解和选择
 * 3. 性能优化：数学方法更高效，DP 方法更通用
```

```
     * 4. 边界处理：正确处理 n=0 和 n=1 的情况
     * 5. 状态设计：合理设计状态参数，减少状态数
     *
     * 算法特色：
     * 1. 位掩码技术：高效记录数字使用情况
     * 2. 排列组合：利用数学性质优化计算
     * 3. 记忆化搜索：避免重复计算
     * 4. 补集转换：将复杂问题转化为简单问题
     */

    public static void main(String[] args) {
        // 运行功能测试
        testNumDupDigitsAtMostN();

        // 运行性能测试
        performanceTest();

        // 调试模式
        debugNumDupDigitsAtMostN();

        // 边界测试
        System.out.println("=== 边界测试 ===");
        System.out.println("n=0: " + numDupDigitsAtMostN(0));
        System.out.println("n=1: " + numDupDigitsAtMostN(1));
        System.out.println("n=11: " + numDupDigitsAtMostN(11)); // 第一个有重复数字的数
    }
}

==============================================

文件: Code18_BeautifulNumbersCF_Enhanced.cpp
==============================================
#include <iostream>
#include <vector>
#include <string>
#include <functional>
#include <algorithm>
#include <cstring>

using namespace std;

/**
 * Codeforces 55D. Beautiful Numbers (增强版)
```

```
 * 题目链接：https://codeforces.com/problemset/problem/55/D
 *
 * 题目描述：
 * 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。
 * 给定区间 [l, r]，求其中美丽数字的个数。
 *
 * 解题思路：
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 是否已开始填数字
 *    - 当前数字对 LCM(1-9)的余数
 *    - 已使用数字的 LCM
 * 3. 关键优化：1-9 的 LCM 是 2520，所有数字的 LCM 都是 2520 的约数
 *
 * 时间复杂度分析：
 * - 状态数：log(r) * 2 * 2 * 2520 * 50 ≈ 10^7
 * - 每个状态处理 10 种选择
 * - 总复杂度：O(10^8) 在可接受范围内
 *
 * 空间复杂度分析：
 * - 记忆化数组：log(r) * 2 * 2 * 2520 * 50 ≈ 40MB
 *
 * 最优解分析：
 * 这是标准的最优解，利用了 LCM 的数学性质和数位 DP 的记忆化
 */

class BeautifulNumbersCF {
private:
    static const int MOD = 2520; // 1-9 的 LCM
    vector<int> digits;          // 存储数位
    vector<int> lcm_map;         // LCM 映射表

    // 计算两个数的最大公约数
    int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    // 计算两个数的最小公倍数
    int lcm(int a, int b) {
        return a / gcd(a, b) * b;
    }
```

```
    // 预计算 1-9 所有子集的 LCM
    void precomputeLCM() {
        lcm_map.resize(1 << 9, 1);
        for (int mask = 1; mask < (1 << 9); mask++) {
            int lcm_val = 1;
            for (int i = 1; i <= 9; i++) {
                if (mask & (1 << (i-1))) {
                    lcm_val = lcm(lcm_val, i);
                }
            }
            lcm_map[mask] = lcm_val;
        }
    }

public:
    /**
     * 计算区间[l, r]中美丽数字的个数
     * 时间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
     * 空间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
     */
    long long countBeautifulNumbers(long long l, long long r) {
        precomputeLCM();
        return countUpTo(r) - countUpTo(l - 1);
    }

private:
    /**
     * 计算[0, n]中美丽数字的个数
     */
    long long countUpTo(long long n) {
        if (n < 0) return 0;
        if (n == 0) return 1; // 0 是美丽数字（特殊情况）

        // 将数字转换为字符串
        string s = to_string(n);
        digits.resize(s.length());
        for (int i = 0; i < s.length(); i++) {
            digits[i] = s[i] - '0';
        }

        int len = digits.size();
```

```
        // 记忆化数组：dp[pos][isLimit][isNum][mod][mask]
        vector<vector<vector<vector<vector<long long>>>>> dp(
            len, vector<vector<vector<vector<long long>>>>(
                2, vector<vector<vector<long long>>>(
                    2, vector<vector<long long>>(
                        MOD, vector<long long>(1 << 9, -1)
                    )
                )
            )
        );

        // 使用 lambda 函数实现 DFS
        function<long long(int, bool, bool, int, int)> dfs = [&](int pos, bool isLimit, bool
isNum, int mod, int mask) -> long long {
            // 递归终止条件
            if (pos == len) {
                if (!isNum) return 1; // 前导零也算美丽数字（特殊情况）

                // 检查是否美丽：数字能被所有非零数字整除
                int actualLCM = lcm_map[mask];
                return (mod % actualLCM == 0) ? 1 : 0;
            }

            // 记忆化搜索
            if (!isLimit && isNum && dp[pos][0][0][mod][mask] != -1) {
                return dp[pos][0][0][mod][mask];
            }

            long long ans = 0;

            // 处理前导零
            if (!isNum) {
                ans += dfs(pos + 1, false, false, mod, mask);
            }

            // 确定当前位可选范围
            int up = isLimit ? digits[pos] : 9;
            int start = isNum ? 0 : 1;

            // 枚举当前位可选数字
            for (int d = start; d <= up; d++) {
                int newMod = (mod * 10 + d) % MOD;
                int newMask = mask;
```

```
                if (d > 0) {
                    newMask |= (1 << (d-1));
                }
                ans += dfs(pos + 1, isLimit && d == up, true, newMod, newMask);
            }

            // 记忆化存储
            if (!isLimit && isNum) {
                dp[pos][0][0][mod][mask] = ans;
            }

            return ans;
        };

        return dfs(0, true, false, 0, 0);
    }
};

/**
 * 单元测试函数
 */
void testBeautifulNumbers() {
    cout << "=== 测试 Beautiful Numbers ===" << endl;

    BeautifulNumbersCF bn;

    // 测试用例 1: 小范围
    cout << "测试区间[1, 9]:" << endl;
    long long result1 = bn.countBeautifulNumbers(1, 9);
    cout << "结果: " << result1 << endl;
    cout << "预期: 9 (所有 1-9 的数字都美丽)" << endl;
    cout << endl;

    // 测试用例 2: 包含不美丽数字
    cout << "测试区间[1, 20]:" << endl;
    long long result2 = bn.countBeautifulNumbers(1, 20);
    cout << "结果: " << result2 << endl;
    cout << "预期: 14 (1,2,3,4,5,6,7,8,9,10,11,12,15,18,20)" << endl;
    cout << endl;

    // 测试用例 3: 较大范围
    cout << "测试区间[1, 100]:" << endl;
    long long result3 = bn.countBeautifulNumbers(1, 100);
```

```
    cout << "结果: " << result3 << endl;
    cout << endl;
}

int main() {
    // 运行功能测试
    testBeautifulNumbers();

    return 0;
}
```

==========

文件: Code18_BeautifulNumbersCF_Enhanced.java

==========

```
package class084;

/**
 * Codeforces 55D. Beautiful Numbers (增强版)
 * 题目链接：https://codeforces.com/problemset/problem/55/D
 *
 * 题目描述：
 * 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。
 * 给定区间 [l, r]，求其中美丽数字的个数。
 *
 * 解题思路：
 * 1. 数位DP方法：使用数位DP框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 是否已开始填数字
 *    - 当前数字对LCM(1-9)的余数
 *    - 已使用数字的LCM
 * 3. 关键优化：1-9的LCM是2520，所有数字的LCM都是2520的约数
 *
 * 时间复杂度分析：
 * - 状态数：log(r) * 2 * 2 * 2520 * 50 ≈ 10^7
 * - 每个状态处理10种选择
 * - 总复杂度：O(10^8) 在可接受范围内
 *
 * 空间复杂度分析：
 * - 记忆化数组：log(r) * 2 * 2 * 2520 * 50 ≈ 40MB
 *
```

```
 * 最优解分析：
 * 这是标准的最优解，利用了 LCM 的数学性质和数位 DP 的记忆化
 */
public class Code18_BeautifulNumbersCF_Enhanced {

    private static final int MOD = 2520; // 1-9 的 LCM
    private static int[] digits;         // 存储数位
    private static int[] lcmMap;        // LCM 映射表

    /**
     * 预计算 1-9 所有子集的 LCM
     * 时间复杂度: O(2^9 * 9) = O(4608)
     * 空间复杂度: O(2^9) = O(512)
     */
    private static void precomputeLCM() {
        lcmMap = new int[1 << 9];
        // 初始化为 1
        for (int i = 0; i < (1 << 9); i++) {
            lcmMap[i] = 1;
        }

        for (int mask = 1; mask < (1 << 9); mask++) {
            int lcmVal = 1;
            for (int i = 1; i <= 9; i++) {
                if ((mask & (1 << (i-1))) != 0) {
                    lcmVal = lcm(lcmVal, i);
                }
            }
            lcmMap[mask] = lcmVal;
        }
    }

    /**
     * 计算两个数的最大公约数
     * 时间复杂度: O(log(min(a,b)))
     */
    private static int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }

    /**
     * 计算两个数的最小公倍数
     * 时间复杂度: O(log(min(a,b)))
```

```
     */
    private static int lcm(int a, int b) {
        return a / gcd(a, b) * b;
    }

    /**
     * 计算区间[l, r]中美丽数字的个数
     * 时间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
     * 空间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
     */
    public static long countBeautifulNumbers(long l, long r) {
        precomputeLCM();
        return countUpTo(r) - countUpTo(l - 1);
    }

    /**
     * 计算[0, n]中美丽数字的个数
     * 使用记忆化搜索实现数位DP
     */
    private static long countUpTo(long n) {
        if (n < 0) return 0;
        if (n == 0) return 1; // 0是美丽数字（特殊情况）

        // 将数字转换为数位数组
        String s = String.valueOf(n);
        digits = new int[s.length()];
        for (int i = 0; i < s.length(); i++) {
            digits[i] = s.charAt(i) - '0';
        }

        int len = digits.length;

        // 记忆化数组：dp[pos][isLimit][isNum][mod][mask]
        // 使用Long数组支持null值，避免稀疏数组浪费空间
        Long[][][][][] dp = new Long[len][2][2][MOD][1 << 9];

        // 使用DFS进行数位DP
        return dfs(0, true, false, 0, 0, dp);
    }

    /**
     * 数位DP递归函数
     *
```

```
     * @param pos 当前处理的位置
     * @param isLimit 是否受到上界限制
     * @param isNum 是否已开始填数字
     * @param mod 当前数字对 MOD 的余数
     * @param mask 已使用数字的位掩码
     * @param dp 记忆化数组
     * @return 满足条件的数字个数
     */
    private static long dfs(int pos, boolean isLimit, boolean isNum,
                            int mod, int mask, Long[][][][][] dp) {
        // 递归终止条件：处理完所有数位
        if (pos == digits.length) {
            if (!isNum) return 1; // 前导零也算美丽数字（特殊情况）

            // 检查是否美丽：数字能被所有非零数字整除
            int actualLCM = lcmMap[mask];
            return (mod % actualLCM == 0) ? 1 : 0;
        }

        // 记忆化搜索：如果已计算过且不受限制且已开始填数字
        if (!isLimit && isNum && dp[pos][0][0][mod][mask] != null) {
            return dp[pos][0][0][mod][mask];
        }

        long ans = 0;

        // 处理前导零：可以选择跳过当前位
        if (!isNum) {
            ans += dfs(pos + 1, false, false, mod, mask, dp);
        }

        // 确定当前位可选数字范围
        int up = isLimit ? digits[pos] : 9;
        int start = isNum ? 0 : 1; // 处理前导零

        // 枚举当前位可选数字
        for (int d = start; d <= up; d++) {
            int newMod = (mod * 10 + d) % MOD;
            int newMask = mask;
            if (d > 0) {
                newMask |= (1 << (d-1));
            }
            ans += dfs(pos + 1, isLimit && d == up, true, newMod, newMask, dp);
```

```
        }

        // 记忆化存储：只存储不受限制且已开始填数字的状态
        if (!isLimit && isNum) {
            dp[pos][0][0][mod][mask] = ans;
        }

        return ans;
    }

    /**
     * 单元测试函数
     */
    public static void main(String[] args) {
        System.out.println("=== 测试 Beautiful Numbers ===");

        // 测试用例 1: 小范围
        System.out.println("测试区间[1, 9]:");
        long result1 = countBeautifulNumbers(1, 9);
        System.out.println("结果: " + result1);
        System.out.println("预期: 9 (所有 1-9 的数字都美丽)");
        System.out.println();

        // 测试用例 2: 包含不美丽数字
        System.out.println("测试区间[1, 20]:");
        long result2 = countBeautifulNumbers(1, 20);
        System.out.println("结果: " + result2);
        System.out.println("预期: 14 (1,2,3,4,5,6,7,8,9,10,11,12,15,18,20)");
        System.out.println();

        // 测试用例 3: 较大范围
        System.out.println("测试区间[1, 100]:");
        long result3 = countBeautifulNumbers(1, 100);
        System.out.println("结果: " + result3);
        System.out.println();
    }
}

==========================================

文件: Code18_BeautifulNumbersCF_Enhanced.py
==========================================
"""
```

Codeforces 55D. Beautiful Numbers (增强版)
题目链接：https://codeforces.com/problemset/problem/55/D

题目描述：
如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。
给定区间 [l, r]，求其中美丽数字的个数。

解题思路：
1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
2. 状态设计需要记录：
   - 当前处理位置
   - 是否受上界限制
   - 是否已开始填数字
   - 当前数字对 LCM(1-9)的余数
   - 已使用数字的 LCM
3. 关键优化：1-9 的 LCM 是 2520，所有数字的 LCM 都是 2520 的约数

时间复杂度分析：
- 状态数：log(r) * 2 * 2 * 2520 * 50 ≈ 10^7
- 每个状态处理 10 种选择
- 总复杂度：O(10^8) 在可接受范围内

空间复杂度分析：
- 记忆化数组：log(r) * 2 * 2 * 2520 * 50 ≈ 40MB

最优解分析：
这是标准的最优解，利用了 LCM 的数学性质和数位 DP 的记忆化
"""

```
import math
from functools import lru_cache

class BeautifulNumbersCF:
    def __init__(self):
        self.MOD = 2520  # 1-9 的 LCM
        self.lcm_map = self.precompute_lcm()

    def precompute_lcm(self):
        """
        预计算 1-9 所有子集的 LCM
        时间复杂度: O(2^9 * 9) = O(4608)
        空间复杂度: O(2^9) = O(512)
        """
```

```
    lcm_map = [1] * (1 << 9)

    for mask in range(1, 1 << 9):
        lcm_val = 1
        for i in range(1, 10):
            if mask & (1 << (i-1)):
                lcm_val = self.lcm(lcm_val, i)
        lcm_map[mask] = lcm_val

    return lcm_map

def gcd(self, a, b):
    """
    计算两个数的最大公约数
    时间复杂度: O(log(min(a,b)))
    """
    return math.gcd(a, b)

def lcm(self, a, b):
    """
    计算两个数的最小公倍数
    时间复杂度: O(log(min(a,b)))
    """
    return a // self.gcd(a, b) * b

def count_beautiful_numbers(self, l, r):
    """
    计算区间[l, r]中美丽数字的个数
    时间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
    空间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
    """
    return self.count_up_to(r) - self.count_up_to(l - 1)

def count_up_to(self, n):
    """
    计算[0, n]中美丽数字的个数
    使用记忆化搜索实现数位DP
    """
    if n < 0:
        return 0
    if n == 0:
        return 1  # 0是美丽数字（特殊情况）
```

```
# 将数字转换为字符串
s = str(n)

@lru_cache(maxsize=None)
def dfs(pos, is_limit, is_num, mod, mask):
    """
    数位 DP 递归函数

    Args:
        pos: 当前处理的位置
        is_limit: 是否受到上界限制
        is_num: 是否已开始填数字
        mod: 当前数字对 MOD 的余数
        mask: 已使用数字的位掩码

    Returns:
        满足条件的数字个数
    """
    # 递归终止条件：处理完所有数位
    if pos == len(s):
        if not is_num:
            return 1  # 前导零也算美丽数字（特殊情况）

        # 检查是否美丽：数字能被所有非零数字整除
        actual_lcm = self.lcm_map[mask]
        return 1 if mod % actual_lcm == 0 else 0

    ans = 0

    # 处理前导零：可以选择跳过当前位
    if not is_num:
        ans += dfs(pos + 1, False, False, mod, mask)

    # 确定当前位可选数字范围
    up = int(s[pos]) if is_limit else 9
    start = 0 if is_num else 1  # 处理前导零

    # 枚举当前位可选数字
    for d in range(start, up + 1):
        new_mod = (mod * 10 + d) % self.MOD
        new_mask = mask
        if d > 0:
            new_mask |= (1 << (d-1))
```

```
                ans += dfs(pos + 1, is_limit and d == up, True, new_mod, new_mask)

            return ans

        return dfs(0, True, False, 0, 0)

def main():
    """
    单元测试函数
    """
    print("=== 测试 Beautiful Numbers ===")

    bn = BeautifulNumbersCF()

    # 测试用例 1: 小范围
    print("测试区间[1, 9]:")
    result1 = bn.count_beautiful_numbers(1, 9)
    print(f"结果: {result1}")
    print("预期: 9 (所有 1-9 的数字都美丽)")
    print()

    # 测试用例 2: 包含不美丽数字
    print("测试区间[1, 20]:")
    result2 = bn.count_beautiful_numbers(1, 20)
    print(f"结果: {result2}")
    print("预期: 14 (1,2,3,4,5,6,7,8,9,10,11,12,15,18,20)")
    print()

    # 测试用例 3: 较大范围
    print("测试区间[1, 100]:")
    result3 = bn.count_beautiful_numbers(1, 100)
    print(f"结果: {result3}")
    print()

if __name__ == "__main__":
    main()

==============================================

文件: Code19_CountDigitOneLC233.java
==============================================
package class084;
```

```
/**
 * LeetCode 233. 数字 1 的个数
 * 题目链接: https://leetcode.cn/problems/number-of-digit-one/
 *
 * 题目描述:
 * 给定一个整数 n，计算所有小于等于 n 的非负整数中数字 1 出现的个数。
 *
 * 解题思路:
 * 1. 数位DP方法：使用数位DP框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 当前已出现的1的个数
 * 3. 关键点：统计1出现的次数而不是数字个数
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(log n)
 */
public class Code19_CountDigitOneLC233 {

    /**
     * 数位DP解法
     * 时间复杂度: O(log n) 每个数位最多计算两次(受限/不受限)
     * 空间复杂度: O(log n) 递归栈深度
     */
    public static int countDigitOne(int n) {
        if (n <= 0) {
            return 0;
        }
        // 将数字n转换为字符数组，方便按位处理
        char[] s = String.valueOf(n).toCharArray();
        int len = s.length;
        // dp[i][count][isLimit] 表示处理到第i位，已经出现了count个1，当前是否受限制时的方案数
        // -1表示未计算过
        int[][][] dp = new int[len][len][2];
        for (int i = 0; i < len; i++) {
            for (int j = 0; j < len; j++) {
                dp[i][j][0] = -1;
                dp[i][j][1] = -1;
            }
        }
        return f(s, 0, 0, true, dp);
    }
```

```
// s: 数字的字符数组表示
// i: 当前处理到第几位
// count: 当前已经统计到的 1 的个数
// isLimit: 当前位是否受到上限限制
// dp: 记忆化数组
private static int f(char[] s, int i, int count, boolean isLimit, int[][][] dp) {
    // 递归终止条件：已经处理完所有数位
    if (i == s.length) {
        return count;
    }

    // 记忆化：如果已经计算过该状态，直接返回结果
    if (!isLimit && dp[i][count][0] != -1) {
        return dp[i][count][0];
    }

    // 确定当前位可以填入的数字范围
    // 如果受限制，最大只能填入 s[i]对应的数字，否则可以填入 0-9
    int up = isLimit ? s[i] - '0' : 9;
    int ans = 0;

    // 枚举当前位可以填入的数字
    for (int d = 0; d <= up; d++) {
        // 递归处理下一位
        // 如果当前位填入 1，则 count+1
        // 下一位是否受限制：当前位受限制且填入了上限值
        ans += f(s, i + 1, count + (d == 1 ? 1 : 0), isLimit && d == up, dp);
    }

    // 记忆化存储结果
    if (!isLimit) {
        dp[i][count][0] = ans;
    }
    return ans;
}

/**
 * 数学方法解法（更高效）
 * 时间复杂度: O(log n)
 * 空间复杂度: O(1)
 *
 * 解题思路:
```

```
     * 1. 逐位分析每一位上 1 出现的次数
     * 2. 对于每一位，考虑高位、当前位和低位的影响
     * 3. 根据当前位与 1 的大小关系，分情况计算
     */
    public static int countDigitOneMath(int n) {
        if (n <= 0) return 0;

        int count = 0;
        long divisor = 1; // 当前位的权重

        while (n / divisor > 0) {
            long high = n / (divisor * 10); // 高位部分
            long current = (n / divisor) % 10; // 当前位
            long low = n % divisor; // 低位部分

            // 分情况讨论当前位上 1 的个数
            if (current > 1) {
                count += (high + 1) * divisor;
            } else if (current == 1) {
                count += high * divisor + low + 1;
            } else {
                count += high * divisor;
            }

            divisor *= 10;
        }

        return count;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int n1 = 13;
        System.out.println("n = " + n1 + ", 数字 1 出现的次数: " + countDigitOne(n1));
        System.out.println("数学方法结果: " + countDigitOneMath(n1));
        // 预期输出: 6 (数字 1, 10, 11, 12, 13 中 1 出现了 6 次)

        // 测试用例 2
        int n2 = 0;
        System.out.println("n = " + n2 + ", 数字 1 出现的次数: " + countDigitOne(n2));
        System.out.println("数学方法结果: " + countDigitOneMath(n2));
        // 预期输出: 0
```

```
        // 测试用例 3
        int n3 = 100;
        System.out.println("n = " + n3 + ", 数字 1 出现的次数: " + countDigitOne(n3));
        System.out.println("数学方法结果: " + countDigitOneMath(n3));
        // 预期输出: 21

        // 测试用例 4
        int n4 = 1000;
        System.out.println("n = " + n4 + ", 数字 1 出现的次数: " + countDigitOne(n4));
        System.out.println("数学方法结果: " + countDigitOneMath(n4));
        // 预期输出: 301
    }
}

==========================================

文件: Code20_NonNegativeIntegersWithoutConsecutiveOnesLC600.java
==========================================
package class084;

/**
 * LeetCode 600. 不含连续 1 的非负整数
 * 题目链接: https://leetcode.cn/problems/non-negative-integers-without-consecutive-ones/
 *
 * 题目描述:
 * 给定一个正整数 n，返回在 [0, n] 范围内不含连续 1 的非负整数的个数。
 *
 * 解题思路:
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定二进制数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 前一位是否为 1
 * 3. 关键点：当前位不能与前一位同时为 1
 *
 * 时间复杂度: O(log n)
 * 空间复杂度: O(log n)
 */
public class Code20_NonNegativeIntegersWithoutConsecutiveOnesLC600 {

    /**
     * 数位 DP 解法
```

```
 * 时间复杂度: O(log n) 每个数位最多计算几次(受限/不受限, 前一位是0/1)
 * 空间复杂度: O(log n) 递归栈深度
 */
public static int findIntegers(int n) {
    // 将数字n转换为二进制字符数组，方便按位处理
    char[] s = Integer.toBinaryString(n).toCharArray();
    int len = s.length;
    // dp[i][prev][isLimit]
    // 表示处理到第i位，前一位是prev，当前是否受限制时的方案数
    // -1表示未计算过
    int[][][] dp = new int[len][2][2];
    for (int i = 0; i < len; i++) {
        for (int j = 0; j < 2; j++) {
            for (int k = 0; k < 2; k++) {
                dp[i][j][k] = -1;
            }
        }
    }
    return f(s, 0, 0, true, dp);
}

// s: 数字的二进制字符数组表示
// i: 当前处理到第几位
// prev: 前一位填的数字
// isLimit: 当前位是否受到上限限制
// dp: 记忆化数组
private static int f(char[] s, int i, int prev, boolean isLimit, int[][][] dp) {
    // 递归终止条件：已经处理完所有数位
    if (i == s.length) {
        // 成功构造一个有效数字
        return 1;
    }

    // 记忆化：如果已经计算过该状态，直接返回结果
    if (!isLimit && dp[i][prev][0] != -1) {
        return dp[i][prev][0];
    }

    // 确定当前位可以填入的数字范围
    // 如果受限制，最大只能填入s[i]对应的数字，否则可以填入0-1
    int up = isLimit ? s[i] - '0' : 1;
    int ans = 0;
```

```
        // 枚举当前位可以填入的数字
        for (int d = 0; d <= up; d++) {
            // 不能有连续的1
            if (prev == 1 && d == 1) {
                continue;
            }
            // 递归处理下一位
            // 下一位是否受限制：当前位受限制且填入了上限值
            ans += f(s, i + 1, d, isLimit && d == up, dp);
        }

        // 记忆化存储结果
        if (!isLimit) {
            dp[i][prev][0] = ans;
        }
        return ans;
    }

    /**
     * 数学方法解法（斐波那契数列）- 更高效的解法
     * 时间复杂度: O(log n)
     * 空间复杂度: O(1)
     *
     * 解题思路:
     * 1. 观察发现，不含连续1的二进制数个数满足斐波那契数列
     * 2. 对于k位二进制数，有效数字个数为fib(k+2)
     * 3. 利用这个性质可以快速计算
     */
    public static int findIntegersMath(int n) {
        if (n == 0) return 1;
        if (n == 1) return 2;

        // 预处理斐波那契数列
        int[] fib = new int[32];
        fib[0] = 1;
        fib[1] = 2;
        for (int i = 2; i < 32; i++) {
            fib[i] = fib[i-1] + fib[i-2];
        }

        String binary = Integer.toBinaryString(n);
        int len = binary.length();
        int ans = 0;
```

```
    boolean prevBit = false; // 前一位是否为1

    for (int i = 0; i < len; i++) {
        if (binary.charAt(i) == '1') {
            // 如果当前位为1，可以选择填0，后面位可以任意填
            ans += fib[len - i - 1];

            // 如果前一位也是1，说明出现了连续1，后面的数字都不满足条件
            if (prevBit) {
                return ans;
            }
            prevBit = true;
        } else {
            prevBit = false;
        }
    }

    // 加上n本身（如果n本身满足条件）
    return ans + 1;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例1
    int n1 = 5;
    System.out.println("n = " + n1 + ", 不含连续1的非负整数个数: " + findIntegers(n1));
    System.out.println("数学方法结果: " + findIntegersMath(n1));
    // 预期输出: 5 (0, 1, 2, 4, 5 满足条件，3的二进制是11，不满足)

    // 测试用例2
    int n2 = 1;
    System.out.println("n = " + n2 + ", 不含连续1的非负整数个数: " + findIntegers(n2));
    System.out.println("数学方法结果: " + findIntegersMath(n2));
    // 预期输出: 2 (0, 1 满足条件)

    // 测试用例3
    int n3 = 2;
    System.out.println("n = " + n3 + ", 不含连续1的非负整数个数: " + findIntegers(n3));
    System.out.println("数学方法结果: " + findIntegersMath(n3));
    // 预期输出: 3 (0, 1, 2 满足条件)

    // 测试用例4
    int n4 = 10;
```

```
        System.out.println("n = " + n4 + ", 不含连续1的非负整数个数: " + findIntegers(n4));
        System.out.println("数学方法结果: " + findIntegersMath(n4));
        // 预期输出: 8
    }
}
```

==========

文件: Code21_NumbersWithRepeatedDigitsLC1012.java

==========

```
package class084;

/**
 * LeetCode 1012. 至少有 1 位重复的数字
 * 题目链接: https://leetcode.cn/problems/numbers-with-repeated-digits/
 *
 * 题目描述:
 * 给定正整数 n，返回在 [1, n] 范围内具有至少 1 位重复数字的正整数的个数。
 *
 * 解题思路:
 * 1. 补集思想：统计没有重复数字的数字个数，然后用总数减去它
 * 2. 数位DP方法：使用数位DP框架，逐位确定数字
 * 3. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 是否已开始填数字
 *    - 已使用数字的位掩码
 * 4. 关键点：用位掩码记录已使用的数字
 *
 * 时间复杂度: O(10 * 2^10 * log n)
 * 空间复杂度: O(2^10 * log n)
 */
public class Code21_NumbersWithRepeatedDigitsLC1012 {

    /**
     * 数位DP解法
     * 时间复杂度: O(10 * 2^10 * log n)
     * 空间复杂度: O(2^10 * log n)
     */
    public static int numDupDigitsAtMostN(int n) {
        // 总数减去没有重复数字的数字个数
        return n - countUniqueDigits(n);
    }
```

```
/**
 * 计算[0, n]中没有重复数字的数字个数
 */
private static int countUniqueDigits(int n) {
    if (n <= 0) return 1; // 0

    // 将数字 n 转换为字符数组
    char[] s = String.valueOf(n).toCharArray();
    int len = s.length;

    // 记忆化数组：dp[pos][isLimit][isNum][mask]
    Integer[][][][] dp = new Integer[len][2][2][1 << 10];

    return dfs(s, 0, true, false, 0, dp);
}

/**
 * 数位 DP 递归函数
 *
 * @param s 数字的字符数组表示
 * @param pos 当前处理位置
 * @param isLimit 是否受到上界限制
 * @param isNum 是否已开始填数字
 * @param mask 已使用数字的位掩码
 * @param dp 记忆化数组
 * @return 满足条件的数字个数
 */
private static int dfs(char[] s, int pos, boolean isLimit, boolean isNum,
                       int mask, Integer[][][][] dp) {
    // 递归终止条件：处理完所有数位
    if (pos == s.length) {
        return isNum ? 1 : 0; // 已填数字才计数
    }

    // 记忆化搜索
    if (!isLimit && isNum && dp[pos][0][0][mask] != null) {
        return dp[pos][0][0][mask];
    }

    int ans = 0;

    // 处理前导零：可以选择跳过当前位
```

```
    if (!isNum) {
        ans += dfs(s, pos + 1, false, false, mask, dp);
    }

    // 确定当前位可选数字范围
    int up = isLimit ? (s[pos] - '0') : 9;
    int start = isNum ? 0 : 1; // 处理前导零

    // 枚举当前位可选数字
    for (int d = start; d <= up; d++) {
        // 检查数字 d 是否已被使用
        if ((mask & (1 << d)) != 0) {
            continue; // 数字重复，跳过
        }

        // 递归处理下一位
        ans += dfs(s, pos + 1, isLimit && (d == up), true,
                   mask | (1 << d), dp);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][0][0][mask] = ans;
    }

    return ans;
}

/**
 * 数学方法解法（排列组合）- 更高效的解法
 * 时间复杂度: O(10^2)
 * 空间复杂度: O(1)
 *
 * 解题思路:
 * 1. 对于 n 位数，没有重复数字的数字个数可以用排列组合计算
 * 2. 总个数 = 1 位数个数 + 2 位数个数 + ... + n 位数个数
 * 3. k 位数个数 = 9 × 9 × 8 × ... × (11-k)
 */
public static int numDupDigitsAtMostNMath(int n) {
    if (n <= 10) return 0;

    // 计算没有重复数字的数字个数
    int uniqueCount = countUniqueDigitsMath(n);
```

```
    return n - uniqueCount;
}

private static int countUniqueDigitsMath(int n) {
    if (n == 0) return 1;

    // 将 n 转换为字符串获取位数
    String s = String.valueOf(n);
    int len = s.length();

    // 计算位数小于 len 的数字个数
    int count = 0;
    for (int i = 1; i < len; i++) {
        count += countUniqueDigitsOfLength(i);
    }

    // 计算位数等于 len 且小于等于 n 的数字个数
    count += countUniqueDigitsEqualToLength(s);

    return count;
}

/**
 * 计算恰好 k 位数且没有重复数字的数字个数
 */
private static int countUniqueDigitsOfLength(int k) {
    if (k == 0) return 1;
    if (k == 1) return 9;

    int count = 9; // 第一位有 9 种选择（1-9）
    int available = 9; // 剩余可用数字个数

    for (int i = 2; i <= k; i++) {
        count *= available;
        available--;
    }

    return count;
}

/**
 * 计算位数等于给定字符串长度且没有重复数字的数字个数
 */
```

```
private static int countUniqueDigitsEqualToLength(String s) {
    int len = s.length();
    boolean[] used = new boolean[10];
    int count = 0;

    // 逐位处理
    for (int i = 0; i < len; i++) {
        int currentDigit = s.charAt(i) - '0';

        // 确定当前位可以选择的数字范围
        int start = (i == 0) ? 1 : 0; // 第一位不能为0
        for (int d = start; d < currentDigit; d++) {
            if (!used[d]) {
                // 计算剩余位数的排列数
                count += countPermutations(10 - countUsed(used) - 1, len - i - 1);
            }
        }

        // 如果当前数字已被使用，后面的数字都不满足条件
        if (used[currentDigit]) {
            break;
        }

        used[currentDigit] = true;

        // 如果是最后一位且所有数字都不重复，计数加1
        if (i == len - 1) {
            count++;
        }
    }

    return count;
}

/**
 * 计算排列数：从m个元素中取n个的排列数
 */
private static int countPermutations(int m, int n) {
    if (n == 0) return 1;
    if (n > m) return 0;

    int result = 1;
    for (int i = 0; i < n; i++) {
```

```
            result *= (m - i);
        }
        return result;
    }

    /**
     * 统计已使用的数字个数
     */
    private static int countUsed(boolean[] used) {
        int count = 0;
        for (boolean u : used) {
            if (u) count++;
        }
        return count;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1
        int n1 = 20;
        System.out.println("n = " + n1);
        System.out.println("至少有 1 位重复数字的正整数个数: " + numDupDigitsAtMostN(n1));
        System.out.println("数学方法结果: " + numDupDigitsAtMostNMath(n1));
        // 预期输出: 1 (只有 11 满足条件)

        // 测试用例 2
        int n2 = 100;
        System.out.println("n = " + n2);
        System.out.println("至少有 1 位重复数字的正整数个数: " + numDupDigitsAtMostN(n2));
        System.out.println("数学方法结果: " + numDupDigitsAtMostNMath(n2));
        // 预期输出: 10

        // 测试用例 3
        int n3 = 10;
        System.out.println("n = " + n3);
        System.out.println("至少有 1 位重复数字的正整数个数: " + numDupDigitsAtMostN(n3));
        System.out.println("数学方法结果: " + numDupDigitsAtMostNMath(n3));
        // 预期输出: 0 (1-10 中没有重复数字)

        // 测试用例 4
        int n4 = 1000;
        System.out.println("n = " + n4);
        System.out.println("至少有 1 位重复数字的正整数个数: " + numDupDigitsAtMostN(n4));
```

```
        System.out.println("数学方法结果: " + numDupDigitsAtMostNMath(n4));
        // 预期输出: 262
    }
}

==========================================

文件: Code22_BeautifulNumbersCF55D.java
==========================================
package class084;

/**
 * Codeforces 55D. Beautiful Numbers
 * 题目链接: https://codeforces.com/problemset/problem/55/D
 *
 * 题目描述:
 * 如果一个正整数能被它的所有非零数字整除，那么这个数就是美丽的。
 * 给定区间 [l, r]，求其中美丽数字的个数。
 *
 * 解题思路:
 * 1. 数位DP方法：使用数位DP框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 是否已开始填数字
 *    - 当前数字对LCM(1-9)的余数
 *    - 已使用数字的LCM
 * 3. 关键优化：1-9的LCM是2520，所有数字的LCM都是2520的约数
 * 4. 关键点：一个数能被其所有非零数字整除等价于这个数能被这些数字的最小公倍数(LCM)整除
 *
 * 时间复杂度: O(log r * 2520 * 50)
 * 空间复杂度: O(log r * 2520 * 50)
 */
public class Code22_BeautifulNumbersCF55D {

    private static final int MOD = 2520; // 1-9的LCM
    private static int[] digits;         // 存储数位
    private static int[] lcmMap;        // LCM映射表

    /**
     * 预计算1-9所有子集的LCM
     * 时间复杂度: O(2^9 * 9) = O(4608)
     * 空间复杂度: O(2^9) = O(512)
```

```
 */
private static void precomputeLCM() {
    lcmMap = new int[1 << 9];
    for (int mask = 0; mask < (1 << 9); mask++) {
        lcmMap[mask] = 1;
    }

    for (int mask = 1; mask < (1 << 9); mask++) {
        int lcmVal = 1;
        for (int i = 1; i <= 9; i++) {
            if ((mask & (1 << (i-1))) != 0) {
                lcmVal = lcm(lcmVal, i);
            }
        }
        lcmMap[mask] = lcmVal;
    }
}

/**
 * 计算两个数的最大公约数
 * 时间复杂度: O(log(min(a,b)))
 */
private static int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

/**
 * 计算两个数的最小公倍数
 * 时间复杂度: O(log(min(a,b)))
 */
private static int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}

/**
 * 计算区间[l, r]中美丽数字的个数
 * 时间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
 * 空间复杂度: O(log(r) * 2 * 2 * 2520 * 50)
 */
public static long countBeautifulNumbers(long l, long r) {
    precomputeLCM();
    return countUpTo(r) - countUpTo(l - 1);
}
```

```
/**
 * 计算[0, n]中美丽数字的个数
 * 使用记忆化搜索实现数位 DP
 */
private static long countUpTo(long n) {
    if (n < 0) return 0;
    if (n == 0) return 1; // 0 是美丽数字（特殊情况）

    // 将数字转换为数位数组
    String s = String.valueOf(n);
    digits = new int[s.length()];
    for (int i = 0; i < s.length(); i++) {
        digits[i] = s.charAt(i) - '0';
    }

    int len = digits.length;

    // 记忆化数组：dp[pos][isLimit][isNum][mod][mask]
    // 使用 Long 数组支持 null 值，避免稀疏数组浪费空间
    Long[][][][][] dp = new Long[len][2][2][MOD][1 << 9];

    // 使用 DFS 进行数位 DP
    return dfs(0, true, false, 0, 0, dp);
}

/**
 * 数位 DP 递归函数
 *
 * @param pos 当前处理的位置
 * @param isLimit 是否受到上界限制
 * @param isNum 是否已开始填数字
 * @param mod 当前数字对 MOD 的余数
 * @param mask 已使用数字的位掩码
 * @param dp 记忆化数组
 * @return 满足条件的数字个数
 */
private static long dfs(int pos, boolean isLimit, boolean isNum,
                        int mod, int mask, Long[][][][][] dp) {
    // 递归终止条件：处理完所有数位
    if (pos == digits.length) {
        if (!isNum) return 1; // 前导零也算美丽数字（特殊情况）
```

```
        // 检查是否美丽：数字能被所有非零数字整除
        int actualLCM = lcmMap[mask];
        return (mod % actualLCM == 0) ? 1 : 0;
    }

    // 记忆化搜索：如果已计算过且不受限制且已开始填数字
    if (!isLimit && isNum && dp[pos][0][0][mod][mask] != null) {
        return dp[pos][0][0][mod][mask];
    }

    long ans = 0;

    // 处理前导零：可以选择跳过当前位
    if (!isNum) {
        ans += dfs(pos + 1, false, false, mod, mask, dp);
    }

    // 确定当前位可选数字范围
    int up = isLimit ? digits[pos] : 9;
    int start = isNum ? 0 : 1; // 处理前导零

    // 枚举当前位可选数字
    for (int d = start; d <= up; d++) {
        int newMod = (mod * 10 + d) % MOD;
        int newMask = mask;
        if (d > 0) {
            newMask |= (1 << (d-1));
        }
        ans += dfs(pos + 1, isLimit && d == up, true, newMod, newMask, dp);
    }

    // 记忆化存储：只存储不受限制且已开始填数字的状态
    if (!isLimit && isNum) {
        dp[pos][0][0][mod][mask] = ans;
    }

    return ans;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: 小范围
    System.out.println("测试区间[1, 9]:");
```

```
        long result1 = countBeautifulNumbers(1, 9);
        System.out.println("结果: " + result1);
        System.out.println("预期: 9 (所有 1-9 的数字都美丽)");
        System.out.println();

        // 测试用例 2: 包含不美丽数字
        System.out.println("测试区间[1, 20]:");
        long result2 = countBeautifulNumbers(1, 20);
        System.out.println("结果: " + result2);
        System.out.println("预期: 14 (1,2,3,4,5,6,7,8,9,10,11,12,15,18,20)");
        System.out.println();

        // 测试用例 3: 较大范围
        System.out.println("测试区间[1, 100]:");
        long result3 = countBeautifulNumbers(1, 100);
        System.out.println("结果: " + result3);
        System.out.println();

        // 测试用例 4: 更大范围
        System.out.println("测试区间[1, 1000]:");
        long result4 = countBeautifulNumbers(1, 1000);
        System.out.println("结果: " + result4);
        System.out.println();
    }
}
```

==========

文件: Code23_DigitsParadeABC135D.java

==========

```
package class084;

/**
 * AtCoder ABC135 D - Digits Parade
 * 题目链接: https://atcoder.jp/contests/abc135/tasks/abc135_d
 *
 * 题目描述:
 * 给定一个由数字和'?'组成的字符串 S，'?'可以替换成 0-9 的任意数字。
 * 求有多少种替换方案使得结果能被 13 整除，结果对 10^9+7 取模。
 *
 * 解题思路:
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
```

```
 *     - 当前处理位置
 *     - 当前数字对 13 的余数
 * 3. 关键点：'?'可以替换为 0-9 的任意数字
 *
 * 时间复杂度: O(n * 13)
 * 空间复杂度: O(n * 13)
 */
public class Code23_DigitsParadeABC135D {
    private static final int MOD = 1000000007;
    private static final int DIVISOR = 13;

    /**
     * 计算有多少种替换方案使得结果能被 13 整除
     * 时间复杂度: O(n * 13)
     * 空间复杂度: O(n * 13)
     */
    public static int countDivisibleBy13(String s) {
        int n = s.length();
        char[] chars = s.toCharArray();

        // dp[i][r] 表示处理到第 i 位，当前余数为 r 的方案数
        long[][] dp = new long[n + 1][DIVISOR];
        dp[0][0] = 1;  // 初始状态：余数为 0 有 1 种方案（空数字）

        // 从高位到低位动态规划
        for (int i = 0; i < n; i++) {
            for (int r = 0; r < DIVISOR; r++) {
                if (dp[i][r] == 0) continue;

                if (chars[i] == '?') {
                    // '?'可以替换为 0-9 的任意数字
                    for (int d = 0; d <= 9; d++) {
                        int newR = (r * 10 + d) % DIVISOR;
                        dp[i + 1][newR] = (dp[i + 1][newR] + dp[i][r]) % MOD;
                    }
                } else {
                    // 固定数字
                    int d = chars[i] - '0';
                    int newR = (r * 10 + d) % DIVISOR;
                    dp[i + 1][newR] = (dp[i + 1][newR] + dp[i][r]) % MOD;
                }
            }
        }
```

```
    return (int) dp[n][0];
}

/**
 * 使用记忆化 DFS 的替代解法（更符合数位 DP 传统风格）
 * 时间复杂度: O(n * 13)
 * 空间复杂度: O(n * 13)
 */
public static int countDivisibleBy13DFS(String s) {
    char[] chars = s.toCharArray();
    int n = chars.length;

    // 记忆化数组
    Long[][] memo = new Long[n][DIVISOR];

    return (int) dfs(chars, 0, 0, memo);
}

private static long dfs(char[] chars, int pos, int remainder, Long[][] memo) {
    // 递归终止条件：处理完所有字符
    if (pos == chars.length) {
        return (remainder == 0) ? 1 : 0;
    }

    // 记忆化搜索
    if (memo[pos][remainder] != null) {
        return memo[pos][remainder];
    }

    long count = 0;

    if (chars[pos] == '?') {
        // '?'可以替换为 0-9 的任意数字
        for (int d = 0; d <= 9; d++) {
            int newRemainder = (remainder * 10 + d) % DIVISOR;
            count = (count + dfs(chars, pos + 1, newRemainder, memo)) % MOD;
        }
    } else {
        // 固定数字
        int d = chars[pos] - '0';
        int newRemainder = (remainder * 10 + d) % DIVISOR;
        count = (count + dfs(chars, pos + 1, newRemainder, memo)) % MOD;
```

```
        }

        // 记忆化存储
        memo[pos][remainder] = count;
        return count;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1: 简单情况
        String s1 = "??";
        int result1 = countDivisibleBy13(s1);
        int result1DFS = countDivisibleBy13DFS(s1);
        System.out.println("输入: " + s1);
        System.out.println("DP 结果: " + result1);
        System.out.println("DFS 结果: " + result1DFS);
        System.out.println("结果一致: " + (result1 == result1DFS));
        System.out.println("预期: 100 种组合中有几个能被 13 整除");
        System.out.println();

        // 测试用例 2: 固定数字
        String s2 = "13";
        int result2 = countDivisibleBy13(s2);
        int result2DFS = countDivisibleBy13DFS(s2);
        System.out.println("输入: " + s2);
        System.out.println("DP 结果: " + result2);
        System.out.println("DFS 结果: " + result2DFS);
        System.out.println("结果一致: " + (result2 == result2DFS));
        System.out.println("预期: 13 能被 13 整除，所以为 1");
        System.out.println();

        // 测试用例 3: 混合情况
        String s3 = "1?2";
        int result3 = countDivisibleBy13(s3);
        int result3DFS = countDivisibleBy13DFS(s3);
        System.out.println("输入: " + s3);
        System.out.println("DP 结果: " + result3);
        System.out.println("DFS 结果: " + result3DFS);
        System.out.println("结果一致: " + (result3 == result3DFS));
        System.out.println();

        // 测试用例 4: 边界情况
        System.out.println("=== 边界测试 ===");
```

```
        System.out.println("空字符串: " + countDivisibleBy13(""));
        System.out.println("单个'?': " + countDivisibleBy13("?"));
        System.out.println("全'?': " + countDivisibleBy13("???"));
    }
}
```

==========

文件: Code24_SimilarDistributionP4127.java

==========

```
package class084;

/**
 * 洛谷 P4127 [AHOI2009] 同类分布
 * 题目链接: https://www.luogu.com.cn/problem/P4127
 *
 * 题目描述:
 * 给出两个数 a,b，求出[a,b]中各位数字之和能整除原数的数的个数。
 *
 * 解题思路:
 * 1. 数位 DP 方法：使用数位 DP 框架，逐位确定数字
 * 2. 状态设计需要记录：
 *    - 当前处理位置
 *    - 是否受上界限制
 *    - 是否已开始填数字
 *    - 当前数位和
 *    - 当前数值对某个数的余数
 * 3. 关键点：枚举所有可能的数位和，然后对每个数位和进行数位 DP
 *
 * 时间复杂度: O(log b * 162 * 162)
 * 空间复杂度: O(log b * 162 * 162)
 */
public class Code24_SimilarDistributionP4127 {

    /**
     * 数位 DP 解法
     * 时间复杂度: O(log(b) * 162 * 162 * 2 * 2)
     * 空间复杂度: O(log(b) * 162 * 162 * 2 * 2)
     *
     * 解题思路:
     * 1. 将问题转化为统计[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
     * 2. 由于数位和 s 的最大可能值为 9*18=162（假设最多 18 位数），我们可以枚举数位和 s
     * 3. 对于每个数位和 s，使用数位 DP 统计满足以下条件的数 x 的个数：
```

```
     *     - x 的数位和等于 s
     *     - x 能被 s 整除
     * 4. 状态需要记录：当前处理到第几位、当前数位和、当前数值对 s 的余数、是否受到上界限制、是否已经开始填数字
     * 5. 通过记忆化搜索避免重复计算
     */
    public static long similarDistribution(long a, long b) {
        // 计算[0, b]中符合条件的数的个数减去[0, a-1]中符合条件的数的个数
        return countValidNumbers(b) - countValidNumbers(a - 1);
    }

    // 计算[0, n]中符合条件的数的个数
    private static long countValidNumbers(long n) {
        if (n < 1) {
            return 0; // 0 不符合条件，因为不能除以 0
        }

        String s = String.valueOf(n);
        int len = s.length();
        int maxSum = len * 9; // 最大可能的数位和
        long result = 0;

        // 枚举所有可能的数位和 s
        for (int s_sum = 1; s_sum <= maxSum; s_sum++) {
            // 对于每个数位和 s_sum，统计满足条件的数的个数
            result += countNumbersWithSumDivisibleBy(s.toCharArray(), s_sum);
        }

        return result;
    }

    // 统计数位和等于 s_sum 且能被 s_sum 整除的数的个数
    private static long countNumbersWithSumDivisibleBy(char[] digits, int s_sum) {
        int len = digits.length;
        // dp[pos][sum][mod][isLimit][isNum]
        // pos: 当前处理到第几位
        // sum: 当前数位和
        // mod: 当前数值对 s_sum 的余数
        // isLimit: 是否受到上界限制
        // isNum: 是否已开始填数字
        long[][][][][] dp = new long[len][s_sum + 1][s_sum][2][2];

        // 初始化 dp 为-1，表示未计算过
```

```
        for (int i = 0; i < len; i++) {
            for (int j = 0; j <= s_sum; j++) {
                for (int k = 0; k < s_sum; k++) {
                    for (int l = 0; l < 2; l++) {
                        for (int m = 0; m < 2; m++) {
                            dp[i][j][k][l][m] = -1;
                        }
                    }
                }
            }
        }

        return dfs(digits, 0, 0, 0, true, false, s_sum, dp);
    }

    /**
     * 数位DP递归函数
     *
     * @param digits 数字的字符数组
     * @param pos 当前处理到第几位
     * @param sum 当前数位和
     * @param mod 当前数值对s_sum的余数
     * @param isLimit 是否受到上界限制
     * @param isNum 是否已开始填数字（处理前导零）
     * @param s_sum 目标数位和
     * @param dp 记忆化数组
     * @return 从当前状态开始，符合条件的数的个数
     */
    private static long dfs(char[] digits, int pos, int sum, int mod, boolean isLimit, boolean
isNum, int s_sum, long[][][][][] dp) {
        // 递归终止条件
        if (pos == digits.length) {
            // 只有当已经填了数字，且数位和等于s_sum，且数值能被s_sum整除时才算符合条件
            return isNum && sum == s_sum && mod == 0 ? 1 : 0;
        }

        // 记忆化搜索
        if (!isLimit && isNum && dp[pos][sum][mod][0][0] != -1) {
            return dp[pos][sum][mod][0][0];
        }

        long ans = 0;
```

```
    // 如果还没开始填数字，可以选择跳过当前位（处理前导零）
    if (!isNum) {
        ans += dfs(digits, pos + 1, sum, mod, false, false, s_sum, dp);
    }

    // 确定当前位可以填入的数字范围
    int upper = isLimit ? digits[pos] - '0' : 9;

    // 枚举当前位可以填入的数字
    for (int d = isNum ? 0 : 1; d <= upper; d++) {
        int newSum = sum + d;
        // 如果新的数位和已经超过了 s_sum，可以提前剪枝
        if (newSum > s_sum) {
            continue;
        }

        // 更新当前数值对 s_sum 的余数
        int newMod = (mod * 10 + d) % s_sum;
        boolean newIsLimit = isLimit && (d == upper);
        boolean newIsNum = isNum || (d > 0);

        // 递归处理下一位
        ans += dfs(digits, pos + 1, newSum, newMod, newIsLimit, newIsNum, s_sum, dp);
    }

    // 记忆化存储
    if (!isLimit && isNum) {
        dp[pos][sum][mod][0][0] = ans;
    }

    return ans;
}

// 测试代码
public static void main(String[] args) {
    // 测试用例 1: a=1, b=20
    // 预期输出: 19 (所有数都符合条件，除了那些数位和为 0 的数)
    long a1 = 1, b1 = 20;
    long result1 = similarDistribution(a1, b1);
    System.out.println("测试用例 1: a=" + a1 + ", b=" + b1);
    System.out.println("符合条件的数的个数: " + result1);

    // 测试用例 2: a=1, b=100
```

```
        long a2 = 1, b2 = 100;
        long result2 = similarDistribution(a2, b2);
        System.out.println("\n 测试用例 2: a=" + a2 + ", b=" + b2);
        System.out.println("符合条件的数的个数: " + result2);

        // 测试用例 3: a=10, b=19
        long a3 = 10, b3 = 19;
        long result3 = similarDistribution(a3, b3);
        System.out.println("\n 测试用例 3: a=" + a3 + ", b=" + b3);
        System.out.println("符合条件的数的个数: " + result3);
        System.out.println("预期: 3 (10, 12, 18)");
    }
}
```

==========