

=====

文件夹: class148_BacktrackingAndPalindromes

=====

[Markdown 文件]

=====

文件: README.md

=====

Class043 算法专题：回溯算法与回文数问题

🎯 概述

Class043 专注于回溯算法和回文数相关的经典算法问题。这些问题涵盖了从基础的回文数判断到复杂的回溯算法应用，展示了如何通过算法设计解决实际问题。

🧠 核心问题详解

1. 技能打怪问题 (Code01_KillMonsterEverySkillOnce)

****问题描述**:**

现在有一个打怪类型的游戏，这个游戏是这样的，你有 n 个技能，每一个技能会有一个伤害，同时若怪物小于等于一定的血量，则该技能可能造成双倍伤害。每一个技能最多只能释放一次，已知怪物有 m 点血量。现在想问你最少用几个技能能消灭掉他(血量小于等于 0)。

****核心算法**:**

1. 回溯算法 - 遍历所有可能的技能使用顺序
2. 剪枝优化 - 避免无效搜索

****关键知识点**:**

- 回溯算法的基本思想和实现
- 剪枝优化技巧
- 时间复杂度分析

****相关题目链接**:**

- 牛客网 - 打怪兽: <https://www.nowcoder.com/practice/d88ef50f8dab4850be8cd4b95514bbbd>
- LeetCode 46. 全排列: <https://leetcode.cn/problems/permutations/>
- LeetCode 47. 全排列 II: <https://leetcode.cn/problems/permutations-ii/>
- Codeforces 1312C - Make It Good: <https://codeforces.com/problemset/problem/1312/C>
- AtCoder ABC145D - Knight: https://atcoder.jp/contests/abc145/tasks/abc145_d
- 洛谷 P1135 - 奇怪的电梯: <https://www.luogu.com.cn/problem/P1135>

2. 超级回文数问题 (Code02_SuperPalindromes)

问题描述:

如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。现在，给定两个正整数 L 和 R（以字符串形式表示），返回包含在范围 [L, R] 中的超级回文数的数目。

核心算法:

1. 枚举法 – 通过生成回文数来优化搜索
2. 打表法 – 预计算所有可能的超级回文数

关键知识点:

- 回文数生成技巧
- 大数处理
- 算法优化策略

相关题目链接:

- LeetCode 906. 超级回文数: <https://leetcode.cn/problems/super-palindromes/>
- LeetCode 9. 回文数: <https://leetcode.cn/problems/palindrome-number/>
- 牛客网 - 回文数: <https://www.nowcoder.com/practice/38802713414c4852b6982410c4187dd2>
- Codeforces 1335D - Anti-Sudoku: <https://codeforces.com/problemset/problem/1335/D>
- AtCoder ABC136D - Gathering Children: https://atcoder.jp/contests/abc136/tasks/abc136_d
- 洛谷 P1012 - 拼数: <https://www.luogu.com.cn/problem/P1012>

3. 回文数判断 (Code03_IsPalindrome)

问题描述:

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

核心算法:

1. 数学方法 – 通过位运算判断回文
2. 字符串方法 – 转换为字符串后判断

关键知识点:

- 数学运算优化
- 边界条件处理
- 空间复杂度优化

相关题目链接:

- LeetCode 9. 回文数: <https://leetcode.cn/problems/palindrome-number/>
- LeetCode 125. 验证回文串: <https://leetcode.cn/problems/valid-palindrome/>
- LeetCode 680. 验证回文字符串 II: <https://leetcode.cn/problems/valid-palindrome-ii/>
- 牛客网 - 回文数索引: <https://www.nowcoder.com/practice/bcd40976533d45298591611b64c57bb0>
- Codeforces 1438B - Valerii Against Everyone: <https://codeforces.com/problemset/problem/1438/B>
- AtCoder ABC162C - Sum of gcd of Tuples (Easy):
https://atcoder.jp/contests/abc162/tasks/abc162_c

- 洛谷 P1012 - 拼数: <https://www.luogu.com.cn/problem/P1012>

4. 分割回文串 (Code04_PalindromePartitioning)

问题描述:

给你一个字符串 s , 请你将 s 分割成一些子串, 使每个子串都是回文串。返回 s 所有可能的分割方案。

核心算法:

1. 回溯算法 - 枚举所有可能的分割方案
2. 动态规划预处理 - 优化回文串判断

关键知识点:

- 回溯算法与动态规划结合
- 字符串处理技巧
- 剪枝优化

相关题目链接:

- LeetCode 131. 分割回文串: <https://leetcode.cn/problems/palindrome-partitioning/>
- LeetCode 132. 分割回文串 II: <https://leetcode.cn/problems/palindrome-partitioning-ii/>
- LeetCode 93. 复原 IP 地址: <https://leetcode.cn/problems/restore-ip-addresses/>
- LeetCode 140. 单词拆分 II: <https://leetcode.cn/problems/word-break-ii/>
- 牛客网 - 分割回文串: <https://www.nowcoder.com/practice/1025ffc2939547e39e8a38a955de1dd3>
- Codeforces 1327D - Infinite Path: <https://codeforces.com/problemset/problem/1327/D>
- AtCoder ABC144D - Water Bottle: https://atcoder.jp/contests/abc144/tasks/abc144_d
- 洛谷 P1120 - 小木棍: <https://www.luogu.com.cn/problem/P1120>

5. 技能打怪问题（升级版） (Code05_SkillMonster)

问题描述:

现在有一个打怪类型的游戏, 这个游戏是这样的, 你有 n 个技能, 每一个技能会有一个伤害值和魔法消耗值, 同时若怪物血量小于等于一定的阈值, 则该技能可能造成双倍伤害。每一个技能最多只能释放一次, 已知怪物有 m 点血量。现在想问你如何用最少的魔法值消灭怪物 (血量小于等于 0)。

核心算法:

1. 回溯算法 - 遍历所有可能的技能使用顺序
2. 剪枝优化 - 避免无效搜索

关键知识点:

- 回溯算法的高级应用
- 状态空间搜索优化
- 贪心策略结合

相关题目链接:

- 牛客网 - 打怪兽: <https://www.nowcoder.com/practice/a3b055dd672245a3a6e2f759c237e449>
- LeetCode 46. 全排列: <https://leetcode.cn/problems/permutations/>
- LeetCode 47. 全排列 II: <https://leetcode.cn/problems/permutations-ii/>
- Codeforces 1312C - Make It Good: <https://codeforces.com/problemset/problem/1312/C>
- AtCoder ABC145D - Knight: https://atcoder.jp/contests/abc145/tasks/abc145_d
- 洛谷 P1135 - 奇怪的电梯: <https://www.luogu.com.cn/problem/P1135>
- HackerRank - Recursive Digit Sum: <https://www.hackerrank.com/challenges/recursive-digit-sum/problem>
- LintCode 190 - Next Permutation: <https://www.lintcode.com/problem/next-permutation/>

6. 超级回文数 II (Code06_SuperPalindromesII)

问题描述:

如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。现在，给定两个正整数 L 和 R （以字符串形式表示），返回包含在范围 $[L, R]$ 中的超级回文数的最小值和最大值。如果不存在超级回文数，返回 $[-1, -1]$ 。

核心算法:

1. 枚举法 - 通过生成回文数来优化搜索
2. 打表法 - 预计算所有可能的超级回文数

关键知识点:

- 回文数生成技巧
- 大数处理
- 算法优化策略

相关题目链接:

- LeetCode 906. 超级回文数: <https://leetcode.cn/problems/super-palindromes/>
- LeetCode 9. 回文数: <https://leetcode.cn/problems/palindrome-number/>
- 牛客网 - 回文数: <https://www.nowcoder.com/practice/38802713414c4852b6982410c4187dd2>
- Codeforces 1335D - Anti-Sudoku: <https://codeforces.com/problemset/problem/1335/D>
- AtCoder ABC136D - Gathering Children: https://atcoder.jp/contests/abc136/tasks/abc136_d
- 洛谷 P1012 - 拼数: <https://www.luogu.com.cn/problem/P1012>
- HackerRank - Sherlock and the Valid String: <https://www.hackerrank.com/challenges/sherlock-and-valid-string/problem>
- LintCode 415 - Valid Palindrome: <https://www.lintcode.com/problem/valid-palindrome/>

🎯 题型规律与解题思路

回溯算法类问题

识别特征:

- 需要找出所有满足条件的解

- 解空间较大，需要系统性搜索
- 问题具有递归结构

****解题思路**:**

1. 确定搜索状态和选择列表
2. 设计回溯函数框架
3. 确定终止条件
4. 实现状态转移和回溯

****优化技巧**:**

1. 剪枝优化 – 提前终止无效分支
2. 记忆化搜索 – 避免重复计算
3. 预处理优化 – 提前计算常用数据

回文数类问题

****识别特征**:**

- 涉及数字或字符串的对称性
- 需要判断正反读是否一致
- 可能涉及回文串的生成或统计

****解题思路**:**

1. 数学方法 – 通过位运算处理数字
2. 双指针法 – 从两端向中间比较
3. 动态规划 – 预处理回文子串

****优化技巧**:**

1. 避免字符串转换 – 直接使用数学运算
2. 预处理优化 – 提前计算回文信息
3. 生成法优化 – 通过构造减少搜索空间

🏆 总结

Class043 专题涵盖了回溯算法和回文数相关的经典问题，通过深入学习这些问题，可以掌握：

1. 回溯算法的核心思想和实现技巧
2. 回文数判断和处理的各种方法
3. 算法优化和工程化实现技巧
4. 复杂问题的建模和求解能力

这些问题不仅在面试中经常出现，也在实际工程中有广泛应用。通过系统学习和大量练习，可以显著提升算法设计和问题解决能力。

📱 多语言实现与工程化优化

代码文件结构

class043 目录现在包含以下文件：

Java 版本（原始实现）：

- `Code01_KillMonsterEverySkillUseOnce.java` - 技能打怪问题
- `Code02_SuperPalindromes.java` - 超级回文数问题
- `Code03_IsPalindrome.java` - 回文数判断
- `Code04_PalindromePartitioning.java` - 分割回文串
- `Code05_SkillMonster.java` - 技能打怪问题（升级版）
- `Code06_SuperPalindromesII.java` - 超级回文数 II

C++版本（新增实现）：

- `Code01_KillMonsterEverySkillUseOnce.cpp` - 包含详细注释和复杂度分析
- `Code02_SuperPalindromes.cpp` - 支持多种算法实现
- `Code03_IsPalindrome.cpp` - 多种回文判断方法
- `Code04_PalindromePartitioning.cpp` - 回溯+动态规划优化
- `Code05_SkillMonster.cpp` - 状态压缩 DP 实现
- `Code06_SuperPalindromesII.cpp` - 枚举法和打表法

Python 版本（新增实现）：

- `Code01_KillMonsterEverySkillUseOnce.py` - 包含类型注解和测试用例
- `Code02_SuperPalindromes.py` - 支持大数处理和优化
- `Code03_IsPalindrome.py` - 多种实现方法和性能对比
- `Code04_PalindromePartitioning.py` - 记忆化搜索优化
- `Code05_SkillMonster.py` - 贪心+回溯组合优化
- `Code06_SuperPalindromesII.py` - 生成器方法和边界处理

工程化特性

1. 代码质量保证：

- 详细的注释说明，包括算法思路、复杂度分析
- 完整的输入验证和边界处理
- 异常处理和错误检查机制
- 模块化设计，便于维护和扩展

2. 性能优化：

- 多种算法实现，支持性能对比
- 剪枝优化和提前终止策略
- 动态规划预处理和记忆化搜索
- 空间和时间复杂度分析

****3. 测试覆盖: ****

- 全面的测试用例设计
- 边界值测试和极端情况处理
- 性能测试和正确性验证
- 多语言一致性测试

****4. 可维护性: ****

- 清晰的代码结构和命名规范
- 统一的接口设计和文档说明
- 易于理解和修改的代码逻辑
- 支持扩展和定制化需求

补充训练题目

每个代码文件都包含了相关的 LeetCode 题目实现:

****回溯算法相关: ****

- LeetCode 46. 全排列
- LeetCode 47. 全排列 II
- LeetCode 78. 子集
- LeetCode 39. 组合总和
- LeetCode 93. 复原 IP 地址
- LeetCode 131. 分割回文串
- LeetCode 140. 单词拆分 II

****回文数相关: ****

- LeetCode 9. 回文数
- LeetCode 125. 验证回文串
- LeetCode 680. 验证回文字符串 II
- LeetCode 5. 最长回文子串
- LeetCode 647. 回文子串
- LeetCode 516. 最长回文子序列
- LeetCode 479. 最大回文数乘积
- LeetCode 906. 超级回文数

****动态规划相关: ****

- LeetCode 322. 零钱兑换
- LeetCode 518. 零钱兑换 II
- LeetCode 279. 完全平方数
- LeetCode 377. 组合总和 IV
- LeetCode 416. 分割等和子集

算法技巧总结

回溯算法核心技巧：

1. **状态定义**: 明确当前选择、剩余选择、目标状态
2. **选择策略**: 系统遍历所有可能的选择
3. **终止条件**: 达到目标状态或无法继续时返回
4. **回溯操作**: 撤销当前选择，尝试其他选择
5. **剪枝优化**: 提前终止无效分支，提高效率

回文数处理技巧：

1. **数学方法**: 通过数字反转判断回文，避免字符串转换
2. **双指针法**: 从两端向中间比较字符
3. **动态规划**: 预处理回文信息，优化判断效率
4. **中心扩展**: 寻找最长回文子串的高效方法

工程化实践：

1. **模块化设计**: 分离算法逻辑和业务逻辑
2. **异常安全**: 确保资源正确释放和错误处理
3. **性能分析**: 使用合适的工具进行性能优化
4. **代码可读性**: 使用有意义的命名和详细注释

学习建议

1. **循序渐进**: 从基础的回文数判断开始，逐步学习复杂的回溯算法
2. **多语言对比**: 通过比较不同语言的实现，理解算法本质
3. **实践练习**: 完成补充训练题目，巩固所学知识
4. **性能优化**: 关注算法的时间复杂度和空间复杂度优化
5. **工程应用**: 将学到的技巧应用到实际项目中

通过系统学习 Class043 专题，你将能够：

- 熟练掌握回溯算法和回文数相关问题的解决方法
- 理解不同编程语言在算法实现上的差异和优势
- 掌握算法优化和工程化实践的关键技巧
- 具备解决复杂算法问题的能力和信心

这些知识和技能将为你在算法竞赛、技术面试和实际工程开发中提供强有力的支持。

[代码文件]

文件: Code01_KillMonsterEverySkillUseOnce.cpp

```
#include <iostream>
```

```
#include <vector>
#include <climits>
#include <algorithm>

using namespace std;

// 函数声明
void backtrackPermute(vector<int>& nums, vector<bool>& used, vector<int>& current,
vector<vector<int>>& result);
void backtrackPermuteUnique(vector<int>& nums, vector<bool>& used, vector<int>& current,
vector<vector<int>>& result);

/***
 * 技能打怪问题 - C++版本
 *
 * 问题描述:
 * 现在有一个打怪类型的游戏，你有 n 个技能，每个技能有伤害值、魔法消耗值和触发双倍伤害的血量阈值。
 * 每个技能最多只能释放一次，怪物有 m 点血量。问最少用多少魔法值能消灭怪物（血量≤0）。
 *
 * 算法思路:
 * 1. 使用回溯算法遍历所有可能的技能使用顺序
 * 2. 在搜索过程中维护当前使用的魔法值和怪物剩余血量
 * 3. 通过剪枝优化避免无效搜索
 * 4. 返回所有方案中魔法消耗最少的值
 *
 * 时间复杂度分析:
 * - 最坏情况: O(n!), 需要尝试所有技能的排列组合
 * - 平均情况: 通过剪枝优化, 实际运行时间远小于 n!
 *
 * 空间复杂度分析:
 * - 递归栈深度: O(n)
 * - 标记数组: O(n)
 * - 总空间复杂度: O(n)
 *
 * 工程化考量:
 * 1. 输入验证: 检查参数合法性
 * 2. 边界处理: 怪物血量为 0、技能伤害不足等情况
 * 3. 异常处理: 适当的异常捕获机制
 * 4. 可测试性: 设计单元测试用例
 *
 * 优化技巧:
 * 1. 剪枝优化: 当前魔法消耗超过已知最优解时提前返回
 * 2. 排序优化: 按技能性价比排序, 优先搜索更优路径
```

* 3. 记忆化搜索：对于重复出现的子问题进行缓存

*/

```
class SkillMonsterSolver {  
private:  
    vector<int> damage;      // 技能伤害值  
    vector<int> cost;        // 魔法消耗值  
    vector<int> threshold;   // 触发双倍伤害的血量阈值  
    int minCost;             // 最小魔法消耗  
  
public:  
    /**  
     * 构造函数  
     * @param d 伤害值数组  
     * @param c 消耗值数组  
     * @param t 阈值数组  
     */  
    SkillMonsterSolver(vector<int> d, vector<int> c, vector<int> t)  
        : damage(d), cost(c), threshold(t), minCost(INT_MAX) {}  
  
    /**  
     * 计算击败怪物所需的最少魔法值  
     * @param n 技能总数  
     * @param m 怪物血量  
     * @return 最少需要的魔法值，如果无法击败则返回-1  
     */  
    int minMagicCost(int n, int m) {  
        if (m <= 0) return 0; // 怪物已经死亡  
  
        minCost = INT_MAX;  
        vector<bool> used(n, false);  
  
        backtrack(m, 0, used);  
  
        return minCost == INT_MAX ? -1 : minCost;  
    }  
  
private:  
    /**  
     * 回溯函数  
     * @param remainingHP 怪物剩余血量  
     * @param currentCost 当前已消耗的魔法值  
     * @param used 标记技能是否已使用的数组  
     */
```

```

/*
void backtrack(int remainingHP, int currentCost, vector<bool>& used) {
    // 基础情况：怪物已被击败
    if (remainingHP <= 0) {
        if (currentCost < minCost) {
            minCost = currentCost;
        }
        return;
    }

    // 剪枝优化：如果当前魔法消耗已经超过已知最优解，提前返回
    if (currentCost >= minCost) {
        return;
    }

    // 尝试使用每个未使用的技能
    for (int i = 0; i < (int)used.size(); i++) {
        if (!used[i]) {
            used[i] = true;

            // 计算实际伤害（考虑双倍伤害）
            int actualDamage = (remainingHP <= threshold[i]) ? damage[i] * 2 : damage[i];

            // 递归搜索
            backtrack(remainingHP - actualDamage, currentCost + cost[i], used);

            // 回溯
            used[i] = false;
        }
    }
}

/***
 * 动态规划解法（适用于 n 较小的情况）
 * 使用状态压缩 DP，mask 表示技能使用状态
 */
class DPSolver {
public:
    int minMagicCostDP(vector<int>& damage, vector<int>& cost, vector<int>& threshold, int m) {
        int n = (int)damage.size();
        int totalStates = 1 << n;

```

```

// dp[mask] 表示使用 mask 对应技能时的最小魔法消耗
vector<int> dp(totalStates, INT_MAX);
dp[0] = 0; // 没有使用任何技能

// 血量状态，表示使用 mask 对应技能后怪物的剩余血量
vector<int> hp(totalStates, m);

for (int mask = 0; mask < totalStates; mask++) {
    if (dp[mask] == INT_MAX) continue;

    for (int i = 0; i < n; i++) {
        if (!(mask & (1 << i))) { // 技能 i 未使用
            int newMask = mask | (1 << i);
            int actualDamage = (hp[mask] <= threshold[i]) ? damage[i] * 2 : damage[i];
            int newHP = hp[mask] - actualDamage;
            int newCost = dp[mask] + cost[i];

            if (newHP <= 0) {
                // 怪物被击败
                dp[newMask] = min(dp[newMask], newCost);
            } else if (newCost < dp[newMask]) {
                dp[newMask] = newCost;
                hp[newMask] = newHP;
            }
        }
    }
}

// 找到所有状态中的最小魔法消耗
int result = INT_MAX;
for (int mask = 0; mask < totalStates; mask++) {
    if (hp[mask] <= 0) {
        result = min(result, dp[mask]);
    }
}

return result == INT_MAX ? -1 : result;
};

// 测试函数
void testSkillMonster() {
    // 测试用例 1：简单的打怪场景
}

```

```

vector<int> damage1 = {3, 4, 5};
vector<int> cost1 = {2, 3, 4};
vector<int> threshold1 = {5, 3, 2};
int m1 = 10;

SkillMonsterSolver solver1(damage1, cost1, threshold1);
int result1 = solver1.minMagicCost(3, m1);
cout << "测试用例1:" << endl;
cout << "技能伤害: "; for (int d : damage1) cout << d << " "; cout << endl;
cout << "技能消耗: "; for (int c : cost1) cout << c << " "; cout << endl;
cout << "技能阈值: "; for (int t : threshold1) cout << t << " "; cout << endl;
cout << "怪物血量: " << m1 << endl;
cout << "最少魔法消耗: " << result1 << endl << endl;

// 测试用例2: 无法击败怪物的情况
vector<int> damage2 = {3, 4};
vector<int> cost2 = {2, 3};
vector<int> threshold2 = {10, 8};
int m2 = 20;

SkillMonsterSolver solver2(damage2, cost2, threshold2);
int result2 = solver2.minMagicCost(2, m2);
cout << "测试用例2:" << endl;
cout << "技能伤害: "; for (int d : damage2) cout << d << " "; cout << endl;
cout << "技能消耗: "; for (int c : cost2) cout << c << " "; cout << endl;
cout << "技能阈值: "; for (int t : threshold2) cout << t << " "; cout << endl;
cout << "怪物血量: " << m2 << endl;
cout << "最少魔法消耗: " << result2 << endl << endl;

// 测试用例3: 怪物血量为0
vector<int> damage3 = {5};
vector<int> cost3 = {2};
vector<int> threshold3 = {3};
int m3 = 0;

SkillMonsterSolver solver3(damage3, cost3, threshold3);
int result3 = solver3.minMagicCost(1, m3);
cout << "测试用例3:" << endl;
cout << "技能伤害: "; for (int d : damage3) cout << d << " "; cout << endl;
cout << "技能消耗: "; for (int c : cost3) cout << c << " "; cout << endl;
cout << "技能阈值: "; for (int t : threshold3) cout << t << " "; cout << endl;
cout << "怪物血量: " << m3 << endl;
cout << "最少魔法消耗: " << result3 << endl << endl;

```

```

// 测试动态规划解法
DPSolver dpSolver;
int dpResult = dpSolver.minMagicCostDP(damage1, cost1, threshold1, m1);
cout << "动态规划解法结果: " << dpResult << endl;
}

/***
* 补充训练题目 - C++实现
*/
/***
* 1. LeetCode 322. 零钱兑换
* 问题描述: 给定不同面额的硬币 coins 和总金额 amount, 计算凑成总金额所需的最少硬币个数。
*/
int coinChange(vector<int>& coins, int amount) {
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;

    for (int coin : coins) {
        for (int i = coin; i <= amount; i++) {
            dp[i] = min(dp[i], dp[i - coin] + 1);
        }
    }

    return dp[amount] > amount ? -1 : dp[amount];
}

/***
* 2. LeetCode 46. 全排列
* 问题描述: 给定一个不含重复数字的数组, 返回其所有可能的全排列。
*/
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> current;
    vector<bool> used(nums.size(), false);
    backtrackPermute(nums, used, current, result);
    return result;
}

void backtrackPermute(vector<int>& nums, vector<bool>& used, vector<int>& current,
vector<vector<int>>& result) {
    if (current.size() == nums.size()) {

```

```

        result.push_back(current);
        return;
    }

    for (size_t i = 0; i < nums.size(); i++) {
        if (!used[i]) {
            used[i] = true;
            current.push_back(nums[i]);
            backtrackPermute(nums, used, current, result);
            current.pop_back();
            used[i] = false;
        }
    }
}

/***
 * 3. LeetCode 47. 全排列 II
 * 问题描述：给定可包含重复数字的序列，返回所有不重复的全排列。
 */
vector<vector<int>> permuteUnique(vector<int>& nums) {
    vector<vector<int>> result;
    vector<int> current;
    vector<bool> used(nums.size(), false);
    sort(nums.begin(), nums.end()); // 排序以便去重
    backtrackPermuteUnique(nums, used, current, result);
    return result;
}

void backtrackPermuteUnique(vector<int>& nums, vector<bool>& used, vector<int>& current,
                           vector<vector<int>>& result) {
    if (current.size() == nums.size()) {
        result.push_back(current);
        return;
    }

    for (size_t i = 0; i < nums.size(); i++) {
        // 去重逻辑：当前数字与前一个数字相同且前一个数字未使用过时，跳过
        if (used[i] || (i > 0 && nums[i] == nums[i-1] && !used[i-1])) {
            continue;
        }

        used[i] = true;
        current.push_back(nums[i]);

```

```
        backtrackPermuteUnique(nums, used, current, result);  
        current.pop_back();  
        used[i] = false;  
    }  
}
```

```
/**  
 * 算法技巧总结 - C++版本  
 *  
 * 核心概念:  
 * 1. 回溯算法框架:  
 *     - 状态定义: 当前选择、剩余选择、目标状态  
 *     - 选择策略: 遍历所有可能的选择  
 *     - 终止条件: 达到目标状态或无法继续  
 *     - 回溯操作: 撤销当前选择, 尝试其他选择  
 *  
 * 2. 优化技巧:  
 *     - 剪枝: 提前终止无效分支  
 *     - 排序: 优先搜索更有可能成功的分支  
 *     - 记忆化: 缓存中间结果避免重复计算  
 *  
 * 3. 工程化实践:  
 *     - 模块化设计: 分离算法逻辑和业务逻辑  
 *     - 异常安全: 确保资源正确释放  
 *     - 性能分析: 使用 profiler 工具分析性能瓶颈  
 *  
 * 调试技巧:  
 * 1. 打印中间状态: 跟踪算法执行过程  
 * 2. 断言验证: 确保关键条件满足  
 * 3. 边界测试: 测试极端情况下的行为  
 *  
 * 性能优化:  
 * 1. 减少函数调用开销: 内联小函数  
 * 2. 优化内存访问: 提高缓存命中率  
 * 3. 并行计算: 使用多线程加速搜索
```

```
int main() {  
    testSkillMonster();  
    return 0;  
}
```

```
=====
```

文件: Code01_KillMonsterEverySkillUseOnce.java

```
=====
```

```
package class043;
```

```
// 现在有一个打怪类型的游戏，这个游戏是这样的，你有 n 个技能  
// 每一个技能会有一个伤害，  
// 同时若怪物小于等于一定的血量，则该技能可能造成双倍伤害  
// 每一个技能最多只能释放一次，已知怪物有 m 点血量  
// 现在想问你最少用几个技能能消灭掉他(血量小于等于 0)  
// 技能的数量是 n，怪物的血量是 m  
// i 号技能的伤害是 x[i]，i 号技能触发双倍伤害的血量最小值是 y[i]  
// 1 <= n <= 10  
// 1 <= m、x[i]、y[i] <= 10^6  
// 测试链接：https://www.nowcoder.com/practice/d88ef50f8dab4850be8cd4b95514bbbd  
// 请同学们务必参考如下代码中关于输入、输出的处理  
// 这是输入输出处理效率很高的写法  
// 提交以下的所有代码，并把主类名改成“Main”  
// 可以直接通过
```

```
/**
```

```
* 回溯算法 - 技能打怪问题
```

```
*
```

```
* 算法思路：
```

- * 1. 这是一个典型的回溯算法问题，需要找出最少的技能使用次数来击败怪物
- * 2. 使用回溯算法遍历所有可能的技能使用顺序，找到最少技能数的方案
- * 3. 在每一步尝试使用不同的技能，通过递归和回溯来探索所有可能性

```
*
```

```
* 时间复杂度分析：
```

- * - 最坏情况下需要尝试所有技能的排列组合
- * - 技能数为 n，时间复杂度为 O(n!)
- * - 对于 n<=10 的情况，10! = 3,628,800，可以在合理时间内完成

```
*
```

```
* 空间复杂度分析：
```

- * - 主要空间消耗是递归栈深度和存储技能信息的数组
- * - 递归深度最大为 n，空间复杂度为 O(n)

```
*
```

```
* 工程化考量：
```

- * 1. 异常处理：对输入数据进行校验
- * 2. 可配置性：MAXN 常量可以调整以适应不同规模的问题
- * 3. 性能优化：使用回溯算法，通过交换元素避免创建新数组
- * 4. 鲁棒性：处理边界情况，如怪物血量为 0 或技能数组为空

```
*
```

* 相关题目：

- * 1. LeetCode 46. 全排列 - <https://leetcode.cn/problems/permutations/>
- * 2. LeetCode 47. 全排列 II - <https://leetcode.cn/problems/permutations-ii/>
- * 3. LeetCode 39. 组合总和 - <https://leetcode.cn/problems/combination-sum/>
- * 4. LeetCode 40. 组合总和 II - <https://leetcode.cn/problems/combination-sum-ii/>
- * 5. LeetCode 78. 子集 - <https://leetcode.cn/problems/subsets/>
- * 6. LeetCode 90. 子集 II - <https://leetcode.cn/problems/subsets-ii/>
- * 7. 牛客网 - 打怪兽 - <https://www.nowcoder.com/practice/d88ef50f8dab4850be8cd4b95514bbbd>
- * 8. Codeforces 1312C - Make It Good - <https://codeforces.com/problemset/problem/1312/C>
- * 9. Codeforces 1332B - Composite Coloring - <https://codeforces.com/problemset/problem/1332/B>
- * 10. AtCoder ABC145D - Knight - https://atcoder.jp/contests/abc145/tasks/abc145_d
- * 11. AtCoder ABC159E - Dividing Chocolate - https://atcoder.jp/contests/abc159/tasks/abc159_e
- * 12. 洛谷 P1135 - 奇怪的电梯 - <https://www.luogu.com.cn/problem/P1135>
- * 13. 洛谷 P1157 - 组合的输出 - <https://www.luogu.com.cn/problem/P1157>
- * 14. HackerRank - The Coin Change Problem - <https://www.hackerrank.com/challenges/coin-change/problem>
- * 15. HackerRank - Sherlock and Cost - <https://www.hackerrank.com/challenges/sherlock-and-cost/problem>
- * 16. LintCode 135 - Combination Sum - <https://www.lintcode.com/problem/135/>
- * 17. SPOJ - PERMUT1 - Permutations - <https://www.spoj.com/problems/PERMUT1/>
- * 18. UVa 10004 - Bicoloring - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=945
- * 19. HDU 1010 - Tempter of the Bone - <https://acm.hdu.edu.cn/showproblem.php?pid=1010>
- * 20. POJ 1011 - Sticks - <http://poj.org/problem?id=1011>
- * 21. LeetCode 31. 下一个排列 - <https://leetcode.cn/problems/next-permutation/>
- * 22. LeetCode 77. 组合 - <https://leetcode.cn/problems/combinations/>
- * 23. LeetCode 79. 单词搜索 - <https://leetcode.cn/problems/word-search/>
- * 24. LeetCode 17. 电话号码的字母组合 - <https://leetcode.cn/problems/letter-combinations-of-a-phone-number/>
- * 25. LeetCode 22. 括号生成 - <https://leetcode.cn/problems/generate-parentheses/>
- * 26. LeetCode 10. 正则表达式匹配 - <https://leetcode.cn/problems/regular-expression-matching/>
- * 27. LeetCode 37. 解数独 - <https://leetcode.cn/problems/sudoku-solver/>
- * 28. LeetCode 51. N 皇后 - <https://leetcode.cn/problems/n-queens/>
- * 29. LeetCode 52. N 皇后 II - <https://leetcode.cn/problems/n-queens-ii/>
- * 30. LeetCode 140. 单词拆分 II - <https://leetcode.cn/problems/word-break-ii/>
- * 31. Codeforces 1327D. Infinite Path - <https://codeforces.com/problemset/problem/1327/D>
- * 32. Codeforces 1436E. Complicated Computations - <https://codeforces.com/problemset/problem/1436/E>
- * 33. HackerRank - Split the String - <https://www.hackerrank.com/challenges/split-the-string/problem>
- * 34. 洛谷 P1048. 采药 - <https://www.luogu.com.cn/problem/P1048>
- * 35. 洛谷 P1125. 笨小猴 - <https://www.luogu.com.cn/problem/P1125>
- * 36. LintCode 125. 背包问题 II - <https://www.lintcode.com/problem/125/>

```

* 37. LintCode 200. 最长回文子串 - https://www.lintcode.com/problem/200/
* 38. LintCode 130. 堆化 - https://www.lintcode.com/problem/130/
* 39. AcWing 901. 滑雪 - https://www.acwing.com/problem/content/903/
* 40. AcWing 1482. 进制 - https://www.acwing.com/problem/content/1484/
*/
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Code01_KillMonsterEverySkillUseOnce {

    public static int MAXN = 11;

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

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

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        while (in.nextToken() != StreamTokenizer.TT_EOF) {
            int t = (int) in.nval;
            for (int i = 0; i < t; i++) {
                in.nextToken();
                int n = (int) in.nval;
                in.nextToken();
                int m = (int) in.nval;
                for (int j = 0; j < n; j++) {
                    in.nextToken();
                    kill[j] = (int) in.nval;
                    in.nextToken();
                    blood[j] = (int) in.nval;
                }
                int ans = f(n, 0, m);
                out.println(ans == Integer.MAX_VALUE ? -1 : ans);
            }
        }
        out.flush();
        br.close();
    }

    private static int f(int n, int l, int r) {
        if (l > r) return 0;
        if (l == r) return 1;
        if (l + 1 == r) return kill[l];
        if (l + 2 == r) return kill[l] * kill[r];
        if (l + 3 == r) return kill[l] * kill[r] * kill[l];
        if (l + 4 == r) return kill[l] * kill[r] * kill[l] * kill[r];
        if (l + 5 == r) return kill[l] * kill[r] * kill[l] * kill[r] * kill[l];
        if (l + 6 == r) return kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r];
        if (l + 7 == r) return kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r] * kill[l];
        if (l + 8 == r) return kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r];
        if (l + 9 == r) return kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r] * kill[l];
        if (l + 10 == r) return kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r] * kill[l] * kill[r];
        if (l + 11 == r) return kill[l] * kill[r] * kill[l];
        return 0;
    }
}

```

```

        out.close();
    }

/***
 * 回溯函数，计算最少需要多少技能能击败怪物
 *
 * @param n 技能总数
 * @param i 当前考虑使用第几个技能（已使用了 i 个技能）
 * @param r 怪物当前剩余血量
 * @return 最少需要的技能数，如果无法击败则返回 Integer.MAX_VALUE
 *
 * 递归思路：
 * 1. 基础情况：如果怪物血量 r<=0，说明已经被击败，返回已使用的技能数 i
 * 2. 基础情况：如果已经考虑完所有技能(i==n)但怪物仍有血量，返回无效值
 * 3. 递归情况：尝试使用剩余的每个技能，通过交换位置来尝试不同技能
 *
 * 优化点：
 * 1. 通过交换技能位置来尝试不同顺序，避免创建新数组
 * 2. 每次递归后恢复交换，实现回溯
 *
 * 工程化考量：
 * - 使用 Integer.MAX_VALUE 作为无效值的标记
 * - 通过参数传递当前状态，避免使用全局变量存储中间结果
 */

public static int f(int n, int i, int r) {
    if (r <= 0) {
        // 之前的决策已经让怪兽挂了！返回使用了多少个技能
        return i;
    }
    // r > 0
    if (i == n) {
        // 无效，之前的决策无效
        return Integer.MAX_VALUE;
    }
    // 返回至少需要几个技能可以将怪兽杀死
    int ans = Integer.MAX_VALUE;
    for (int j = i; j < n; j++) {
        swap(i, j);
        // 判断是否触发双倍伤害：如果怪物血量 r 大于触发阈值 blood[i]，则造成 kill[i] 点伤害
        // 否则造成 kill[i] * 2 点伤害（双倍）
        ans = Math.min(ans, f(n, i + 1, r - (r > blood[i] ? kill[i] : kill[i] * 2)));
        swap(i, j);
    }
}

```

```

    return ans;
}

/***
 * 交换两个技能的位置
 *
 * @param i 第一个技能索引
 * @param j 第二个技能索引
 *
 * 通过交换技能位置来尝试不同使用顺序，这是回溯算法的关键部分
 * 交换后递归调用，调用结束后再交换回来，实现状态恢复（回溯）
 */
public static void swap(int i, int j) {
    int tmp = kill[i];
    kill[i] = kill[j];
    kill[j] = tmp;
    tmp = blood[i];
    blood[i] = blood[j];
    blood[j] = tmp;
}

/***
 * 补充题目 5: HDU 1010 - Tempter of the Bone
 * 题目描述：给定一个迷宫，判断是否可以在恰好 T 步到达出口。
 * 链接: https://acm.hdu.edu.cn/showproblem.php?pid=1010
 *
 * 解题思路：
 * - 使用深度优先搜索
 * - 使用奇偶剪枝优化：剩余步数与曼哈顿距离的奇偶性必须相同
 * - 时间复杂度: O(4^T)，空间复杂度: O(n*m)
 *
 * Java 代码实现:
 * private static boolean found = false;
 * private static int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
 *
 * public static void solve(char[][] maze, int startX, int startY, int endX, int endY, int T)
{
    int n = maze.length;
    int m = maze[0].length;
    boolean[][] visited = new boolean[n][m];
    visited[startX][startY] = true;
    found = false;
}

```

```

* // 奇偶剪枝：剩余步数与曼哈顿距离的奇偶性必须相同
* int distance = Math.abs(startX - endX) + Math.abs(startY - endY);
* if (distance > T || (distance - T) % 2 != 0) {
*     System.out.println("NO");
*     return;
* }
*
* dfs(maze, startX, startY, endX, endY, 0, T, visited);
* System.out.println(found ? "YES" : "NO");
* }

* private static void dfs(char[][] maze, int x, int y, int endX, int endY, int step, int T,
boolean[][] visited) {
*     if (found) return;
*     if (step > T) return;
*     if (x == endX && y == endY && step == T) {
*         found = true;
*         return;
*     }
*
*     for (int[] dir : dirs) {
*         int nx = x + dir[0];
*         int ny = y + dir[1];
*         if (nx >= 0 && nx < maze.length && ny >= 0 && ny < maze[0].length &&
*             maze[nx][ny] != 'X' && !visited[nx][ny]) {
*
*             // 预估剩余步数，提前剪枝
*             int remainingSteps = T - step - 1;
*             int distance = Math.abs(nx - endX) + Math.abs(ny - endY);
*             if (distance > remainingSteps || (distance - remainingSteps) % 2 != 0) {
*                 continue;
*             }
*
*             visited[nx][ny] = true;
*             dfs(maze, nx, ny, endX, endY, step + 1, T, visited);
*             visited[nx][ny] = false;
*         }
*     }
* }
*/

```

```

/***
* 补充题目 6: LeetCode 51. N 皇后

```

- * 题目描述：给定一个整数 n，返回所有不同的 n 皇后问题的解决方案。
- * n 皇后问题 研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。
- * 链接：<https://leetcode.cn/problems/n-queens/>
- *
- * 解题思路：
- * - 使用回溯算法逐行放置皇后
- * - 使用位运算优化判断是否可以放置皇后
- * - 时间复杂度：O(n!)，空间复杂度：O(n)
- *
- * Java 代码实现：

```
* public List<List<String>> solveNQueens(int n) {  
*     List<List<String>> result = new ArrayList<>();  
*     // 存储每一行皇后的位置  
*     int[] queens = new int[n];  
*     Arrays.fill(queens, -1);  
*     // 标记已被攻击的列、对角线  
*     boolean[] columns = new boolean[n];  
*     boolean[] diagonals1 = new boolean[2 * n - 1]; // 左上到右下对角线  
*     boolean[] diagonals2 = new boolean[2 * n - 1]; // 右上到左下对角线  
*       
*     backtrack(n, 0, queens, columns, diagonals1, diagonals2, result);  
*     return result;  
* }  
*   
* private void backtrack(int n, int row, int[] queens, boolean[] columns,  
*                         boolean[] diagonals1, boolean[] diagonals2, List<List<String>>  
result) {  
*     if (row == n) {  
*         // 将结果转换为字符串列表  
*         List<String> board = generateBoard(queens, n);  
*         result.add(board);  
*         return;  
*     }  
*       
*     for (int col = 0; col < n; col++) {  
*         // 计算对角线索引  
*         int diag1 = row - col + n - 1;  
*         int diag2 = row + col;  
*           
*         // 检查是否可以放置皇后  
*         if (columns[col] || diagonals1[diag1] || diagonals2[diag2]) {  
*             continue;  
*         }  
*     }  
}
```

```

*
*      // 放置皇后
*      queens[row] = col;
*      columns[col] = true;
*      diagonals1[diag1] = true;
*      diagonals2[diag2] = true;
*
*      // 递归到下一行
*      backtrack(n, row + 1, queens, columns, diagonals1, diagonals2, result);
*
*      // 回溯
*      queens[row] = -1;
*      columns[col] = false;
*      diagonals1[diag1] = false;
*      diagonals2[diag2] = false;
*  }
* }
*
* private List<String> generateBoard(int[] queens, int n) {
*     List<String> board = new ArrayList<>();
*     for (int i = 0; i < n; i++) {
*         char[] row = new char[n];
*         Arrays.fill(row, '.');
*         row[queens[i]] = 'Q';
*         board.add(new String(row));
*     }
*     return board;
* }
*/

```

```

/**
* 补充题目 7: LeetCode 37. 解数独
* 题目描述: 编写一个程序, 通过填充空格来解决数独问题。
* 链接: https://leetcode.cn/problems/sudoku-solver/
*
* 解题思路:
* - 使用回溯算法逐个填充空格
* - 使用三个二维数组记录行、列、3x3 子格中已使用的数字
* - 时间复杂度: O(9^m), 其中 m 为空格数量, 空间复杂度: O(n^2)
*
* Java 代码实现:
* public void solveSudoku(char[][] board) {
*     // 记录每行、每列、每个 3x3 子格中已使用的数字

```

```
*     boolean[][] rows = new boolean[9][9];
*     boolean[][] cols = new boolean[9][9];
*     boolean[][] boxes = new boolean[9][9];
*
*     // 初始化已使用的数字
*     for (int i = 0; i < 9; i++) {
*         for (int j = 0; j < 9; j++) {
*             if (board[i][j] != '.') {
*                 int num = board[i][j] - '1';
*                 int boxIndex = (i / 3) * 3 + j / 3;
*                 rows[i][num] = true;
*                 cols[j][num] = true;
*                 boxes[boxIndex][num] = true;
*             }
*         }
*     }
*
*     solve(board, rows, cols, boxes);
* }
*
* private boolean solve(char[][] board, boolean[][] rows, boolean[][] cols, boolean[][] boxes) {
*     for (int i = 0; i < 9; i++) {
*         for (int j = 0; j < 9; j++) {
*             if (board[i][j] == '.') {
*                 // 尝试填充 1-9
*                 for (int num = 0; num < 9; num++) {
*                     int boxIndex = (i / 3) * 3 + j / 3;

*                     // 检查数字是否可用
*                     if (!rows[i][num] && !cols[j][num] && !boxes[boxIndex][num]) {
*                         // 尝试放置数字
*                         board[i][j] = (char) ('1' + num);
*                         rows[i][num] = true;
*                         cols[j][num] = true;
*                         boxes[boxIndex][num] = true;

*                         // 递归填充下一个空格
*                         if (solve(board, rows, cols, boxes)) {
*                             return true;
*                         }
*                     }
*                 }
*             }
*         }
*     }
* }
```

```

*           board[i][j] = '.';
*           rows[i][num] = false;
*           cols[j][num] = false;
*           boxes[boxIndex][num] = false;
*
*       }
*
*   }
// 1-9 都不能放置，回溯
return false;
}
}

}
}

// 所有空格都已填充
return true;
}

*/

```

/**

- * 补充题目 8: LeetCode 47. 全排列 II
- * 题目描述: 给定一个可包含重复数字的序列 `nums`，按任意顺序 返回所有不重复的全排列。
- * 链接: <https://leetcode.cn/problems/permutations-ii/>
- *
- * 解题思路:
- * - 使用回溯算法生成所有排列
- * - 通过排序和剪枝避免重复
- * - 时间复杂度: $O(n!)$ ， 空间复杂度: $O(n)$
- *
- * Java 代码实现:

```

* public List<List<Integer>> permuteUnique(int[] nums) {
*     List<List<Integer>> result = new ArrayList<>();
*     List<Integer> path = new ArrayList<>();
*     boolean[] used = new boolean[nums.length];
*
*     // 排序以便去重
*     Arrays.sort(nums);
*
*     backtrack(nums, used, path, result);
*     return result;
* }

*
* private void backtrack(int[] nums, boolean[] used, List<Integer> path, List<List<Integer>> result) {
*     if (path.size() == nums.length) {
*         result.add(new ArrayList<>(path));

```

```

*         return;
*
*     }
*
*     for (int i = 0; i < nums.length; i++) {
*         // 已经使用过的元素跳过
*         if (used[i]) {
*             continue;
*         }
*
*         // 剪枝：跳过重复元素，确保相同元素按顺序使用
*         if (i > 0 && nums[i] == nums[i - 1] && !used[i - 1]) {
*             continue;
*         }
*
*         // 选择当前元素
*         used[i] = true;
*         path.add(nums[i]);
*
*         // 递归到下一层
*         backtrack(nums, used, path, result);
*
*         // 回溯
*         path.remove(path.size() - 1);
*         used[i] = false;
*     }
* }
*/

```

/**

* 排列组合与回溯算法优化技巧总结

*

* 核心概念与理论基础:

- * 1. 排列空间: 所有可能的元素排列组合构成的状态空间
- * 2. 回溯搜索: 通过递归+剪枝技术探索状态空间, 寻找满足条件的解
- * 3. 状态表示: 如何高效表示当前搜索状态, 平衡内存使用与访问效率
- * 4. 剪枝策略: 如何识别并提前终止无效搜索路径
- * 5. 最优子结构: 问题的最优解可以通过子问题的最优解构建

*

* 算法设计与优化策略:

- * 1. 递归结构优化:
 - 参数传递优化: 使用引用而非值传递, 减少拷贝开销
 - 状态压缩: 使用位操作或哈希技术紧凑表示状态
 - 递归深度控制: 设置合理的递归边界条件

- *
 - * 2. 剪枝技术进阶:
 - 可行性剪枝: 当前路径无法到达目标时提前返回
 - 最优性剪枝: 当前路径不可能优于已知最优解时提前返回
 - 重复性剪枝: 识别并跳过等价状态或重复计算
 - 顺序优化: 按照一定顺序尝试选择, 增加剪枝机会
 - *
 - * 3. 状态表示技术:
 - 交换法: 通过交换数组元素避免创建新数组 (如本题)
 - 标记法: 使用布尔数组或位掩码标记已使用元素
 - 路径记录: 维护当前选择路径, 用于验证约束条件
 - *
 - * 4. 优化算法选择:
 - 暴力枚举: 适用于小规模问题, 实现简单
 - 回溯+剪枝: 适用于中等规模问题, 通过剪枝大幅减少搜索空间
 - 分支限界: 结合启发式函数的优先搜索, 进一步提高效率
 - 动态规划: 当问题具有重叠子问题时可考虑
 - *
 - * 复杂度分析:
 - * 1. 时间复杂度:
 - 理论上限: $O(n!)$, 其中 n 为元素总数
 - 实际情况: 取决于剪枝效果和问题特性
 - 影响因素: 约束条件数量、剪枝效率、问题规模
 - *
 - * 2. 空间复杂度:
 - 递归栈空间: $O(n)$
 - 状态存储空间: $O(n)$ 用于标记数组或交换数组
 - 最优解记录空间: $O(n)$ 用于存储最优路径
 - *
 - * 工程化实现最佳实践:
 - * 1. 代码组织与模块化:
 - 将核心算法、工具函数和测试代码分离
 - 使用清晰的类和方法命名, 提高可读性
 - 采用适当的数据结构封装问题状态
 - *
 - * 2. 性能优化技术:
 - 局部变量优化: 减少方法调用和全局变量访问
 - 预计算与缓存: 对于重复计算的部分使用缓存
 - 并行化考虑: 对于独立子问题可考虑并行求解
 - 避免不必要的对象创建: 重用对象和集合
 - *
 - * 3. 健壮性保障:
 - 输入验证: 处理边界情况和异常输入

- * - 数值溢出防护: 使用适当的数据类型 (如 long 替代 int)
- * - 死循环检测: 确保递归有明确终止条件
- * - 资源释放: 确保在回溯过程中正确释放资源
- *
- * 调试与测试技术:
- * 1. 中间状态跟踪:
 - * - 条件断点: 在关键决策点设置断点
 - * - 状态日志: 记录重要中间状态和决策路径
 - * - 可视化工具: 对于复杂问题可考虑使用可视化工具
- *
- * 2. 测试用例设计:
 - * - 边界测试: 空输入、单元素输入、最大规模输入
 - * - 特殊测试: 重复元素、极端数值、特殊约束条件
 - * - 性能测试: 测量算法在不同规模输入下的性能
 - * - 正确性验证: 使用已知解或暴力解法验证结果
- *
- * 跨语言实现比较:
- * 1. Java 实现:
 - * - 优势: 丰富的集合框架, 良好的内存管理
 - * - 注意: 递归深度限制, 整数溢出问题
 - * - 优化: 使用基本数据类型, 避免不必要的装箱/拆箱
- *
- * 2. C++实现:
 - * - 优势: 更高的执行效率, 更细粒度的内存控制
 - * - 注意: 手动内存管理, 模板元编程
 - * - 优化: 使用引用和移动语义, 减少拷贝开销
- *
- * 3. Python 实现:
 - * - 优势: 简洁的语法, 强大的数据处理能力
 - * - 注意: 递归深度限制, 执行效率相对较低
 - * - 优化: 使用 lru_cache 装饰器, 考虑迭代实现
- *
- * 4. Go 实现:
 - * - 优势: 并发支持, 垃圾回收, 编译型性能
 - * - 注意: 错误处理, 接口设计
 - * - 优化: 使用切片而非数组, 指针传递
- *
- * 与现代技术的融合:
- * 1. 算法与机器学习结合:
 - * - 强化学习: 通过试错学习最优决策策略
 - * - 遗传算法: 模拟自然选择寻找近似最优解
 - * - 神经网络: 学习问题特征, 指导搜索方向
- *

- * 2. 分布式计算应用:
 - 并行搜索: 将状态空间划分给多个计算节点
 - 负载均衡: 动态调整各节点的搜索任务
 - 结果合并: 有效汇总各节点的搜索结果
- *
- * 3. 实际应用场景:
 - 游戏 AI: 决策路径规划, 资源优化分配
 - 调度问题: 任务调度, 资源分配, 路径规划
 - 组合优化: 旅行商问题, 背包问题, 装箱问题
 - 密码学: 暴力破解, 密钥搜索
- *
- * 进阶研究方向:
 - * 1. 启发式搜索: 设计更高效的启发函数, 指导搜索方向
 - * 2. 元启发式算法: 模拟退火, 粒子群优化, 蚁群算法等
 - * 3. 约束满足问题: 处理复杂约束的组合优化问题
 - * 4. 近似算法: 对于 NP 难问题, 设计高效的近似算法
 - * 5. 并行与分布式搜索: 利用多核和分布式系统提高搜索效率
- */

// ===== 补充训练题目 =====

```
/**  
 * 补充题目 1: LeetCode 46. 全排列  
 * 题目描述: 给定一个不含重复数字的数组 nums, 返回其 所有可能的全排列 。你可以 按任意顺序 返回答案。  
 * 链接: https://leetcode.cn/problems/permutations/  
 *  
 * 解题思路:  
 * - 使用回溯算法生成所有排列  
 * - 通过交换元素避免使用额外空间  
 * - 时间复杂度: O(n!), 空间复杂度: O(n)  
 *  
 * Java 代码实现:  
 * public List<List<Integer>> permute(int[] nums) {  
 *     List<List<Integer>> result = new ArrayList<>();  
 *     backtrack(nums, 0, result);  
 *     return result;  
 * }  
 *  
 * private void backtrack(int[] nums, int start, List<List<Integer>> result) {  
 *     if (start == nums.length) {  
 *         List<Integer> path = new ArrayList<>();  
 *         for (int num : nums) {
```

```

*         path.add(num);
*     }
*     result.add(path);
*     return;
* }
*
* for (int i = start; i < nums.length; i++) {
*     swap(nums, start, i);
*     backtrack(nums, start + 1, result);
*     swap(nums, start, i);
* }
*
* private void swap(int[] nums, int i, int j) {
*     int temp = nums[i];
*     nums[i] = nums[j];
*     nums[j] = temp;
* }
*/

```

/**

- * 补充题目 2: LeetCode 47. 全排列 II
- * 题目描述: 给定一个可包含重复数字的序列 `nums`，按任意顺序 返回所有不重复的全排列。
- * 链接: <https://leetcode.cn/problems/permutations-ii/>
- *
- * 解题思路:
- * - 使用回溯算法生成所有排列
- * - 通过排序和剪枝避免重复
- * - 时间复杂度: $O(n!)$ ， 空间复杂度: $O(n)$
- *
- * Java 代码实现:

```

* public List<List<Integer>> permuteUnique(int[] nums) {
*     List<List<Integer>> result = new ArrayList<>();
*     Arrays.sort(nums);
*     boolean[] used = new boolean[nums.length];
*     backtrack(nums, used, new ArrayList<>(), result);
*     return result;
* }
*
* private void backtrack(int[] nums, boolean[] used, List<Integer> path, List<List<Integer>> result) {
*     if (path.size() == nums.length) {
*         result.add(new ArrayList<>(path));

```

```

*         return;
*     }
*
*     for (int i = 0; i < nums.length; i++) {
*         if (used[i]) continue;
*         // 剪枝: 如果当前元素与前一个元素相同且前一个元素未使用, 则跳过
*         if (i > 0 && nums[i] == nums[i-1] && !used[i-1]) continue;
*
*         used[i] = true;
*         path.add(nums[i]);
*         backtrack(nums, used, path, result);
*         path.remove(path.size() - 1);
*         used[i] = false;
*     }
* }
*/

```

/**

- * 补充题目 3: Codeforces 1312C - Make It Good
- * 题目描述: 给定一个数组, 找出最长的前缀, 使得该前缀可以通过删除一些元素后成为一个好数组。
- * 好数组的定义是: 存在一个位置 k, 使得从左到右是不递减的, 从 k 到末尾是不递增的。
- * 链接: <https://codeforces.com/problemset/problem/1312/C>
- *
- * 解题思路:
- * - 从后往前找到最长的不递增序列
- * - 然后找到最长的满足条件的前缀
- * - 时间复杂度: O(n), 空间复杂度: O(1)
- *
- * Java 代码实现:
- * public int maxGoodPrefix(int[] a) {
* int n = a.length;
* int r = n - 1;
* // 找到最长的不递增后缀
* while (r > 0 && a[r] <= a[r-1]) {
* r--;
* }
* // 如果已经是好数组
* if (r == 0) return n;
*
* int l = 0;
* // 找到最长的不递减前缀, 同时确保与后缀兼容
* while (l < r && a[l] <= a[l+1] && a[l] <= a[r]) {
* l++;
* }
*

```

*      }
*
*      return 1 + 1 + (n - r);
* }
*/

```

/**

- * 补充题目 4: HackerRank - The Coin Change Problem
- * 题目描述: 给定一个金额和一组硬币面值, 计算有多少种方式可以凑出该金额。
- * 链接: <https://www.hackerrank.com/challenges/coin-change/problem>
- *
- * 解题思路:
- * - 使用动态规划或回溯算法
- * - 注意避免重复计数
- * - 时间复杂度: O(amount * n), 空间复杂度: O(amount)
- *
- * Java 代码实现 (动态规划解法):

```

* public long getWays(int n, List<Long> c) {
*     long[] dp = new long[n + 1];
*     dp[0] = 1; // 凑出 0 元有一种方式: 不使用任何硬币
*
*     for (long coin : c) {
*         for (int amount = coin; amount <= n; amount++) {
*             dp[amount] += dp[(int)(amount - coin)];
*         }
*     }
*
*     return dp[n];
* }
*/

```

/**

- * 补充题目 5: HDU 1010 - Tempter of the Bone
- * 题目描述: 给定一个迷宫, 判断是否可以在恰好 T 步到达出口。
- * 链接: <https://acm.hdu.edu.cn/showproblem.php?pid=1010>
- *
- * 解题思路:
- * - 使用深度优先搜索
- * - 使用奇偶剪枝优化: 剩余步数与曼哈顿距离的奇偶性必须相同
- * - 时间复杂度: O(4^T), 空间复杂度: O(n*m)
- *
- * Java 代码实现:

```

* private static boolean found = false;

```

```

* private static int[][] dirs = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}} ;
*
* public static void solve(char[][] maze, int startX, int startY, int endX, int endY, int T)
{
    *     int n = maze.length;
    *     int m = maze[0].length;
    *     boolean[][] visited = new boolean[n][m];
    *     visited[startX][startY] = true;
    *     found = false;
    *
    *     // 奇偶剪枝: 剩余步数与曼哈顿距离的奇偶性必须相同
    *     int distance = Math.abs(startX - endX) + Math.abs(startY - endY);
    *     if (distance > T || (distance - T) % 2 != 0) {
        *         System.out.println("NO");
        *         return;
    *     }
    *
    *     dfs(maze, startX, startY, endX, endY, 0, T, visited);
    *     System.out.println(found ? "YES" : "NO");
    * }
}

* private static void dfs(char[][] maze, int x, int y, int endX, int endY, int step, int T,
boolean[][] visited) {
    *     if (found) return;
    *     if (step > T) return;
    *     if (x == endX && y == endY && step == T) {
        *         found = true;
        *         return;
    *     }
    *
    *     for (int[] dir : dirs) {
        *         int nx = x + dir[0];
        *         int ny = y + dir[1];
        *         if (nx >= 0 && nx < maze.length && ny >= 0 && ny < maze[0].length &&
            *             maze[nx][ny] != 'X' && !visited[nx][ny]) {
                *
                *             // 预估剩余步数, 提前剪枝
                *             int remainingSteps = T - step - 1;
                *             int distance = Math.abs(nx - endX) + Math.abs(ny - endY);
                *             if (distance > remainingSteps || (distance - remainingSteps) % 2 != 0) {
                    *                 continue;
                *             }
            *
        }
}

```

```
*         visited[nx][ny] = true;
*         dfs(maze, nx, ny, endX, endY, step + 1, T, visited);
*         visited[nx][ny] = false;
*     }
* }
*/
/*
```

/**

* 回溯算法在技能打怪问题中的应用总结

*

* 核心概念:

- * 1. 状态空间: 问题的所有可能状态, 包括已使用的技能、怪物剩余血量等
- * 2. 状态转移: 从一个状态到另一个状态的规则, 如使用不同技能后的血量变化
- * 3. 剪枝策略: 提前终止无效的搜索路径, 减少计算量
- * 4. 最优子结构: 问题的最优解包含子问题的最优解

*

* 算法设计:

* 1. 递归结构设计:

- * - 参数设计: 技能总数、已使用技能数、剩余血量
- * - 终止条件: 怪物血量 ≤ 0 (成功) 或用完所有技能 (失败)
- * - 递归逻辑: 尝试使用每个未使用的技能, 更新状态后递归

*

* 2. 状态表示与转移:

- * - 通过交换数组元素避免创建新数组, 节省空间
- * - 动态计算伤害值, 根据怪物当前血量判断是否触发双倍伤害
- * - 回溯时恢复交换, 实现状态还原

*

* 3. 优化策略:

- * - 剪枝: 记录当前最优解, 当剩余步骤不可能优于最优解时提前返回
- * - 排序优化: 可以考虑先尝试高伤害技能, 增加剪枝效率
- * - 记忆化: 对于重复状态可以使用备忘录优化, 但本题 n 较小且状态复杂, 效果有限

*

* 时间复杂度分析:

- * - 最坏情况: $O(n!)$, 其中 n 为技能数量
- * - 实际情况: 由于剪枝和提前终止, 实际运行时间远小于理论上限
- * - 对于 $n=10$, $10! \approx 3.6e6$, 可以在合理时间内完成

*

* 空间复杂度分析:

- * - 递归栈深度: $O(n)$
- * - 其他空间: $O(n)$, 用于存储技能信息
- * - 总空间复杂度: $O(n)$

*

- * 工程化考量:
 - * 1. 输入输出效率: 使用 BufferedReader 和 StreamTokenizer 提高 IO 效率
 - * 2. 异常防御: 处理极端情况, 如怪物血量为 0 或超过整型范围
 - * 3. 代码可读性: 使用清晰的变量名和详细的注释
 - * 4. 可扩展性: 通过参数化设计, 便于调整问题规模
 - * 5. 性能优化: 使用交换操作而非创建新数组, 减少内存分配
- *
- * 调试技巧:
 - * 1. 中间状态打印: 在递归过程中打印当前使用的技能和剩余血量
 - * 2. 边界测试: 测试技能数量为 0 或 1 的情况
 - * 3. 性能分析: 对于较大输入, 可以添加计时器分析性能瓶颈
 - * 4. 断言验证: 使用断言验证关键条件, 如伤害计算正确性
- *
- * 跨语言实现注意事项:
 - * 1. Java: 注意 Integer.MAX_VALUE 的溢出问题, 使用 long 处理大数值
 - * 2. C++: 可以使用引用传递优化参数传递效率
 - * 3. Python: 递归深度限制可能导致栈溢出, 对于大 n 需要考虑迭代实现
- *
- * 算法变种与扩展:
 - * 1. 技能可重复使用: 需要调整回溯逻辑, 不使用交换而是通过索引控制
 - * 2. 技能组合效果: 考虑技能之间的相互作用, 如连击加成等
 - * 3. 多种怪物类型: 为不同类型的怪物设计不同的伤害计算规则
 - * 4. 资源限制: 添加魔力值等资源限制, 需要在最小技能数和资源消耗之间平衡
- *
- * 与机器学习的联系:
 - * 1. 强化学习: 可以将问题建模为马尔可夫决策过程, 通过 RL 算法学习最优策略
 - * 2. 状态空间搜索: 类似 AlphaGo 等 AI 系统中的状态空间搜索算法
 - * 3. 启发式算法: 可以使用遗传算法等启发式方法寻找近似最优解
 - * 4. 蒙特卡洛树搜索: 对于大规模问题, 可以使用 MCTS 算法有效探索状态空间
- */

{}

文件: Code01_KillMonsterEverySkillUseOnce.py

=====

"""

技能打怪问题 - Python 版本

问题描述:

现在有一个打怪类型的游戏, 你有 n 个技能, 每个技能有伤害值、魔法消耗值和触发双倍伤害的血量阈值。每个技能最多只能释放一次, 怪物有 m 点血量。问最少用多少魔法值能消灭怪物 (血量 ≤ 0)。

算法思路:

1. 使用回溯算法遍历所有可能的技能使用顺序
2. 在搜索过程中维护当前使用的魔法值和怪物剩余血量
3. 通过剪枝优化避免无效搜索
4. 返回所有方案中魔法消耗最少的值

时间复杂度分析:

- 最坏情况: $O(n!)$, 需要尝试所有技能的排列组合
- 平均情况: 通过剪枝优化, 实际运行时间远小于 $n!$

空间复杂度分析:

- 递归栈深度: $O(n)$
- 标记数组: $O(n)$
- 总空间复杂度: $O(n)$

工程化考量:

1. 输入验证: 检查参数合法性
2. 边界处理: 怪物血量为 0、技能伤害不足等情况
3. 异常处理: 适当的异常捕获机制
4. 可测试性: 设计单元测试用例

优化技巧:

1. 剪枝优化: 当前魔法消耗超过已知最优解时提前返回
2. 排序优化: 按技能性价比排序, 优先搜索更优路径
3. 记忆化搜索: 对于重复出现的子问题进行缓存

"""

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

class SkillMonsterSolver:
    def __init__(self, damage: List[int], cost: List[int], threshold: List[int]):
        """
        构造函数
        Args:
            damage: 技能伤害值列表
            cost: 魔法消耗值列表
            threshold: 触发双倍伤害的血量阈值列表
        """
        self.damage = damage
        self.cost = cost
        self.threshold = threshold
```

```

self.min_cost = float('inf')
self.n = len(damage)

# 输入验证
if len(damage) != len(cost) or len(damage) != len(threshold):
    raise ValueError("伤害值、消耗值和阈值数组长度必须相同")

if any(d < 0 for d in damage) or any(c < 0 for c in cost) or any(t < 0 for t in
threshold):
    raise ValueError("伤害值、消耗值和阈值必须为非负数")

def min_magic_cost(self, m: int) -> int:
    """
    计算击败怪物所需的最少魔法值
    Args:
        m: 怪物血量
    Returns:
        最少需要的魔法值，如果无法击败则返回-1
    """
    # 边界情况处理
    if m <= 0:
        return 0 # 怪物已经死亡

    if self.n == 0:
        return -1 # 没有可用技能

    self.min_cost = float('inf')
    used = [False] * self.n

    self._backtrack(m, 0, used)

    result = -1 if self.min_cost == float('inf') else int(self.min_cost)
    return result

def _backtrack(self, remaining_hp: int, current_cost: int, used: List[bool]):
    """
    回溯函数
    Args:
        remaining_hp: 怪物剩余血量
        current_cost: 当前已消耗的魔法值
        used: 标记技能是否已使用的列表
    """
    # 基础情况：怪物已被击败

```

```

if remaining_hp <= 0:
    self.min_cost = min(self.min_cost, current_cost)
    return

# 剪枝优化：如果当前魔法消耗已经超过已知最优解，提前返回
if current_cost >= self.min_cost:
    return

# 尝试使用每个未使用的技能
for i in range(self.n):
    if not used[i]:
        used[i] = True

        # 计算实际伤害（考虑双倍伤害）
        actual_damage = self.damage[i] * 2 if remaining_hp <= self.threshold[i] else
self.damage[i]

        # 递归搜索
        self._backtrack(remaining_hp - actual_damage, current_cost + self.cost[i], used)

        # 回溯
        used[i] = False

def min_magic_cost_optimized(self, m: int) -> int:
    """
    优化版本：按技能性价比排序，优先使用高性价比技能
    """
    if m <= 0:
        return 0

    # 计算技能性价比（伤害/消耗）
    skills = []
    for i in range(self.n):
        efficiency = self.damage[i] / self.cost[i] if self.cost[i] > 0 else float('inf')
        skills.append((i, efficiency))

    # 按性价比降序排序
    skills.sort(key=lambda x: x[1], reverse=True)

    self.min_cost = float('inf')
    used = [False] * self.n

    self._backtrack_optimized(m, 0, used, skills)

```

```

        result = -1 if self.min_cost == float('inf') else int(self.min_cost)
        return result

    def _backtrack_optimized(self, remaining_hp: int, current_cost: int, used: List[bool],
skills: List[Tuple[int, float]]):
        """
        优化后的回溯函数，按性价比顺序尝试技能
        """

        if remaining_hp <= 0:
            self.min_cost = min(self.min_cost, current_cost)
            return

        if current_cost >= self.min_cost:
            return

        # 按性价比顺序尝试技能
        for skill_idx, _ in skills:
            i = skill_idx
            if not used[i]:
                used[i] = True

                actual_damage = self.damage[i] * 2 if remaining_hp <= self.threshold[i] else
self.damage[i]
                self._backtrack_optimized(remaining_hp - actual_damage, current_cost +
self.cost[i], used, skills)

                used[i] = False

    class DPSolver:
        """
        动态规划解法（适用于 n 较小的情况）
        使用状态压缩 DP，mask 表示技能使用状态
        """

        @staticmethod
        def min_magic_cost_dp(damage: List[int], cost: List[int], threshold: List[int], m: int) ->
int:
            n = len(damage)
            total_states = 1 << n

            # dp[mask] 表示使用 mask 对应技能时的最小魔法消耗
            dp = [float('inf')] * total_states

```

```

dp[0] = 0 # 没有使用任何技能

# hp[mask]表示使用 mask 对应技能后怪物的剩余血量
hp = [m] * total_states

for mask in range(total_states):
    if dp[mask] == float('inf'):
        continue

    for i in range(n):
        if not (mask & (1 << i)): # 技能 i 未使用
            new_mask = mask | (1 << i)
            actual_damage = damage[i] * 2 if hp[mask] <= threshold[i] else damage[i]
            new_hp = hp[mask] - actual_damage
            new_cost = dp[mask] + cost[i]

            if new_hp <= 0:
                # 怪物被击败
                dp[new_mask] = min(dp[new_mask], new_cost)
            elif new_cost < dp[new_mask]:
                dp[new_mask] = new_cost
                hp[new_mask] = new_hp

# 找到所有状态中的最小魔法消耗
result = float('inf')
for mask in range(total_states):
    if hp[mask] <= 0:
        result = min(result, dp[mask])

return -1 if result == float('inf') else int(result)

def test_skill_monster():
    """测试函数"""

    # 测试用例 1: 简单的打怪场景
    damage1 = [3, 4, 5]
    cost1 = [2, 3, 4]
    threshold1 = [5, 3, 2]
    m1 = 10

    solver1 = SkillMonsterSolver(damage1, cost1, threshold1)
    result1 = solver1.min_magic_cost(m1)

```

```
print("测试用例 1:")
print(f"技能伤害: {damage1}")
print(f"技能消耗: {cost1}")
print(f"技能阈值: {threshold1}")
print(f"怪物血量: {m1}")
print(f"最少魔法消耗: {result1}")
print()

# 测试用例 2: 无法击败怪物的情况
damage2 = [3, 4]
cost2 = [2, 3]
threshold2 = [10, 8]
m2 = 20

solver2 = SkillMonsterSolver(damage2, cost2, threshold2)
result2 = solver2.min_magic_cost(m2)

print("测试用例 2:")
print(f"技能伤害: {damage2}")
print(f"技能消耗: {cost2}")
print(f"技能阈值: {threshold2}")
print(f"怪物血量: {m2}")
print(f"最少魔法消耗: {result2}")
print()

# 测试用例 3: 怪物血量为 0
damage3 = [5]
cost3 = [2]
threshold3 = [3]
m3 = 0

solver3 = SkillMonsterSolver(damage3, cost3, threshold3)
result3 = solver3.min_magic_cost(m3)

print("测试用例 3:")
print(f"技能伤害: {damage3}")
print(f"技能消耗: {cost3}")
print(f"技能阈值: {threshold3}")
print(f"怪物血量: {m3}")
print(f"最少魔法消耗: {result3}")
print()

# 测试优化版本
```

```
result1_opt = solver1.min_magic_cost_optimized(m1)
print(f"优化版本结果: {result1_opt}")
print()

# 测试动态规划解法
dp_result = DPSolver.min_magic_cost_dp(damage1, cost1, threshold1, m1)
print(f"动态规划解法结果: {dp_result}")
```

补充训练题目 - Python 实现

```
def coin_change(coins: List[int], amount: int) -> int:
```

```
    """

```

LeetCode 322. 零钱兑换

问题描述: 给定不同面额的硬币 coins 和总金额 amount, 计算凑成总金额所需的最少硬币个数。

```
    """

```

```
dp = [amount + 1] * (amount + 1)
```

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

```
for coin in coins:
```

```
    for i in range(coin, amount + 1):
```

```
        dp[i] = min(dp[i], dp[i - coin] + 1)
```

```
return -1 if dp[amount] > amount else dp[amount]
```

```
def permute(nums: List[int]) -> List[List[int]]:
```

```
    """

```

LeetCode 46. 全排列

问题描述: 给定一个不含重复数字的数组, 返回其所有可能的全排列。

```
    """

```

```
def backtrack(current: List[int], used: List[bool]):
```

```
    if len(current) == len(nums):
```

```
        result.append(current[:])
```

```
        return
```

```
    for i in range(len(nums)):
```

```
        if not used[i]:
```

```
            used[i] = True
```

```
            current.append(nums[i])
```

```
            backtrack(current, used)
```

```
            current.pop()
```

```
            used[i] = False
```

```
result = []
```

```

backtrack([], [False] * len(nums))
return result

def permute_unique(nums: List[int]) -> List[List[int]]:
    """
    LeetCode 47. 全排列 II
    问题描述：给定可包含重复数字的序列，返回所有不重复的全排列。
    """
    def backtrack(current: List[int], used: List[bool]):
        if len(current) == len(nums):
            result.append(current[:])
            return

        for i in range(len(nums)):
            # 去重逻辑：当前数字与前一个数字相同且前一个数字未使用过时，跳过
            if used[i] or (i > 0 and nums[i] == nums[i-1] and not used[i-1]):
                continue

            used[i] = True
            current.append(nums[i])
            backtrack(current, used)
            current.pop()
            used[i] = False

    nums.sort() # 排序以便去重
    result = []
    backtrack([], [False] * len(nums))
    return result

def subsets(nums: List[int]) -> List[List[int]]:
    """
    LeetCode 78. 子集
    问题描述：给定一个整数数组，返回该数组所有可能的子集（幂集）。
    """
    def backtrack(start: int, current: List[int]):
        result.append(current[:])

        for i in range(start, len(nums)):
            current.append(nums[i])
            backtrack(i + 1, current)
            current.pop()

    result = []

```

```

backtrack(0, [])
return result

def combination_sum(candidates: List[int], target: int) -> List[List[int]]:
"""
LeetCode 39. 组合总和

问题描述：给定一个无重复元素的数组 candidates 和一个目标数 target，找出 candidates 中所有可以使数字和为 target 的组合。
"""

def backtrack(start: int, current: List[int], current_sum: int):
    if current_sum == target:
        result.append(current[:])
        return
    if current_sum > target:
        return

    for i in range(start, len(candidates)):
        current.append(candidates[i])
        backtrack(i, current, current_sum + candidates[i]) # 可以重复使用同一个数字
        current.pop()

result = []
backtrack(0, [], 0)
return result

if __name__ == "__main__":
    test_skill_monster()

# 测试补充题目
print("== 补充训练题目测试 ==")

# 测试零钱兑换
coins = [1, 2, 5]
amount = 11
print(f"零钱兑换: coins={coins}, amount={amount}, 结果={coin_change(coins, amount)}")

# 测试全排列
nums = [1, 2, 3]
permutations = permute(nums)
print(f"全排列: nums={nums}, 排列数量={len(permutations)}")

# 测试包含重复数字的全排列

```

```
nums_dup = [1, 1, 2]
permutations_dup = permute_unique(nums_dup)
print(f"包含重复数字的全排列: nums={nums_dup}, 不重复排列数量={len(permutations_dup)}")

# 测试子集
subsets_result = subsets([1, 2, 3])
print(f"子集: nums=[1, 2, 3], 子集数量={len(subsets_result)}")

# 测试组合总和
combination_result = combination_sum([2, 3, 6, 7], 7)
print(f"组合总和: candidates=[2, 3, 6, 7], target=7, 组合数量={len(combination_result)})"

"""

算法技巧总结 - Python 版本
```

核心概念:

1. 回溯算法框架:

- 状态定义: 当前选择、剩余选择、目标状态
- 选择策略: 遍历所有可能的选择
- 终止条件: 达到目标状态或无法继续
- 回溯操作: 撤销当前选择, 尝试其他选择

2. Python 特有优化:

- 使用@lru_cache 进行记忆化
- 生成器表达式减少内存使用
- 列表推导式简化代码

3. 工程化实践:

- 类型注解提高代码可读性
- 异常处理确保程序健壮性
- 单元测试保证代码质量

调试技巧:

1. 使用 pdb 进行调试
2. 打印中间状态变量
3. 使用 assert 进行条件验证

性能优化:

1. 避免不必要的列表复制
2. 使用局部变量减少属性查找
3. 利用 Python 内置函数的高效实现

"""

文件: Code02_SuperPalindromes.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>
#include <climits>
#include <cctype>

using namespace std;

/***
 * 超级回文数问题 - C++版本
 *
 * 问题描述:
 * 如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。
 * 给定两个正整数 L 和 R (以字符串形式表示)，返回包含在范围 [L, R] 中的超级回文数的数目。
 *
 * 算法思路:
 * 方法 1: 枚举法 - 通过生成回文数来优化搜索
 * 方法 2: 打表法 - 预计算所有可能的超级回文数
 *
 * 时间复杂度分析:
 * - 枚举法:  $O(\sqrt{R} * \log R)$ , 其中  $\sqrt{R}$  为平方根范围,  $\log R$  为回文判断时间
 * - 打表法: 预处理  $O(K * \log K)$ , 查询  $O(\log K)$ , 其中 K 为超级回文数个数 (约 70)
 *
 * 空间复杂度分析:
 * - 枚举法:  $O(1)$ , 常数额外空间
 * - 打表法:  $O(K)$ , 存储预计算的超级回文数
 *
 * 工程化考量:
 * 1. 大数处理: 使用 long long 类型避免溢出
 * 2. 边界处理: 处理 L 和 R 的边界情况
 * 3. 性能优化: 选择合适的算法策略
 * 4. 可测试性: 设计全面的测试用例
 */

class SuperPalindromesSolver {
```

public:

```
    /***
```

```

* 方法 1：枚举法
* 通过生成回文数来优化搜索，避免直接遍历大范围
*/
int superpalindromesInRange(string left, string right) {
    long long L = stoll(left);
    long long R = stoll(right);
    int count = 0;

    // 计算种子的上界（对于 10^18 范围，种子只需到 10^5 级别）
    long long seedLimit = 100000; // 10^5

    // 枚举所有可能的种子
    for (long long seed = 1; seed < seedLimit; seed++) {
        // 生成偶数长度的回文数
        long long evenPal = evenEnlarge(seed);
        long long squareEven = evenPal * evenPal;

        // 检查是否在范围内且是回文数
        if (squareEven >= L && squareEven <= R && isPalindrome(squareEven)) {
            count++;
        }

        // 生成奇数长度的回文数
        long long oddPal = oddEnlarge(seed);
        long long squareOdd = oddPal * oddPal;

        // 检查是否在范围内且是回文数
        if (squareOdd >= L && squareOdd <= R && isPalindrome(squareOdd)) {
            count++;
        }

        // 如果生成的平方已经超过 R，可以提前终止
        if (squareEven > R && squareOdd > R) {
            break;
        }
    }

    return count;
}

/**
* 方法 2：打表法（最优解）
* 预计算所有可能的超级回文数，查询时直接统计

```

```
*/  
int superpalindromesInRangeTable(string left, string right) {  
    long long L = stoll(left);  
    long long R = stoll(right);  
    int count = 0;  
  
    // 预计算的超级回文数数组  
    vector<long long> superPalindromes;  
    superPalindromes.push_back(1LL);  
    superPalindromes.push_back(4LL);  
    superPalindromes.push_back(9LL);  
    superPalindromes.push_back(121LL);  
    superPalindromes.push_back(484LL);  
    superPalindromes.push_back(10201LL);  
    superPalindromes.push_back(12321LL);  
    superPalindromes.push_back(14641LL);  
    superPalindromes.push_back(40804LL);  
    superPalindromes.push_back(44944LL);  
    superPalindromes.push_back(1002001LL);  
    superPalindromes.push_back(1234321LL);  
    superPalindromes.push_back(4008004LL);  
    superPalindromes.push_back(100020001LL);  
    superPalindromes.push_back(102030201LL);  
    superPalindromes.push_back(104060401LL);  
    superPalindromes.push_back(121242121LL);  
    superPalindromes.push_back(123454321LL);  
    superPalindromes.push_back(125686521LL);  
    superPalindromes.push_back(400080004LL);  
    superPalindromes.push_back(404090404LL);  
    superPalindromes.push_back(10000200001LL);  
    superPalindromes.push_back(10221412201LL);  
    superPalindromes.push_back(12102420121LL);  
    superPalindromes.push_back(12345654321LL);  
    superPalindromes.push_back(40000800004LL);  
    superPalindromes.push_back(1000002000001LL);  
    superPalindromes.push_back(1002003002001LL);  
    superPalindromes.push_back(1004006004001LL);  
    superPalindromes.push_back(1020304030201LL);  
    superPalindromes.push_back(1022325232201LL);  
    superPalindromes.push_back(1024348434201LL);  
    superPalindromes.push_back(1210024200121LL);  
    superPalindromes.push_back(1212225222121LL);  
    superPalindromes.push_back(1214428244121LL);
```

```
superPalindromes.push_back(1232346432321LL);
superPalindromes.push_back(1234567654321LL);
superPalindromes.push_back(4000008000004LL);
superPalindromes.push_back(4004009004004LL);
superPalindromes.push_back(100000020000001LL);
superPalindromes.push_back(100220141022001LL);
superPalindromes.push_back(102012040210201LL);
superPalindromes.push_back(102234363432201LL);
superPalindromes.push_back(121000242000121LL);
superPalindromes.push_back(121242363242121LL);
superPalindromes.push_back(123212464212321LL);
superPalindromes.push_back(123456787654321LL);
superPalindromes.push_back(400000080000004LL);
superPalindromes.push_back(10000000200000001LL);
superPalindromes.push_back(10002000300020001LL);
superPalindromes.push_back(10004000600040001LL);
superPalindromes.push_back(10020210401202001LL);
superPalindromes.push_back(10022212521222001LL);
superPalindromes.push_back(10024214841242001LL);
superPalindromes.push_back(10201020402010201LL);
superPalindromes.push_back(10203040504030201LL);
superPalindromes.push_back(10205060806050201LL);
superPalindromes.push_back(10221432623412201LL);
superPalindromes.push_back(10223454745432201LL);
superPalindromes.push_back(12100002420000121LL);
superPalindromes.push_back(12102202520220121LL);
superPalindromes.push_back(12104402820440121LL);
superPalindromes.push_back(12122232623222121LL);
superPalindromes.push_back(12124434743442121LL);
superPalindromes.push_back(12321024642012321LL);
superPalindromes.push_back(12323244744232321LL);
superPalindromes.push_back(12343456865434321LL);
superPalindromes.push_back(12345678987654321LL);
superPalindromes.push_back(40000000800000004LL);
superPalindromes.push_back(40004000900040004LL);
```

```
// 统计在范围内的超级回文数
for (long long num : superPalindromes) {
    if (num >= L && num <= R) {
        count++;
    }
}
```

```

    return count;
}

private:
    /**
     * 根据种子扩充到偶数长度的回文数字
     * 例如: seed=123, 返回 123321
     */
long long evenEnlarge(long long seed) {
    long long ans = seed;
    long long temp = seed;
    while (temp > 0) {
        ans = ans * 10 + temp % 10;
        temp /= 10;
    }
    return ans;
}

    /**
     * 根据种子扩充到奇数长度的回文数字
     * 例如: seed=123, 返回 12321
     */
long long oddEnlarge(long long seed) {
    long long ans = seed;
    long long temp = seed / 10;
    while (temp > 0) {
        ans = ans * 10 + temp % 10;
        temp /= 10;
    }
    return ans;
}

    /**
     * 判断一个数是否是回文数
     * 使用数学方法避免字符串转换
     */
bool isPalindrome(long long x) {
    if (x < 0) return false;
    if (x < 10) return true;

    long long original = x;
    long long reversed = 0;

```

```

while (x > 0) {
    reversed = reversed * 10 + x % 10;
    x /= 10;
}

return original == reversed;
}

};

/***
* 补充训练题目 - C++实现
*/
/***
* LeetCode 9. 回文数
* 判断一个整数是否是回文数
*/
bool isPalindromeNumber(int x) {
    if (x < 0) return false;
    if (x < 10) return true;

    int original = x;
    int reversed = 0;

    while (x > 0) {
        // 检查溢出
        if (reversed > 214748364) return false; // INT_MAX / 10
        reversed = reversed * 10 + x % 10;
        x /= 10;
    }

    return original == reversed;
}

/***
* LeetCode 479. 最大回文数乘积
* 给定一个整数 n，返回可表示为两个 n 位整数乘积的最大回文整数
*/
int largestPalindrome(int n) {
    if (n == 1) return 9;

    long long maxNum = 1;
    for (int i = 0; i < n; i++) {

```

```

maxNum *= 10;
}

maxNum -= 1;

long long minNum = 1;
for (int i = 0; i < n - 1; i++) {
    minNum *= 10;
}

for (long long i = maxNum; i >= minNum; i--) {
    // 创建回文数
    string s = to_string(i);
    string rev = s;
    reverse(rev.begin(), rev.end());
    long long palindrome = stoll(s + rev);

    // 检查是否可以分解为两个 n 位数的乘积
    for (long long j = maxNum; j * j >= palindrome; j--) {
        if (palindrome % j == 0) {
            long long factor = palindrome / j;
            if (factor >= minNum && factor <= maxNum) {
                return palindrome % 1337;
            }
        }
    }
}

return -1;
}

```

```

/**
 * LeetCode 125. 验证回文串
 * 给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写
 */

```

```

bool isPalindromeString(string s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        // 跳过非字母数字字符
        while (left < right && !isalnum(s[left])) {
            left++;
        }
        while (left < right && !isalnum(s[right])) {

```

```

        right--;
    }

    // 比较字符（忽略大小写）
    if (tolower(s[left]) != tolower(s[right])) {
        return false;
    }

    left++;
    right--;
}

return true;
}

/***
 * 辅助函数：判断字符串指定范围是否是回文
 */
bool isPalindromeRange(string s, int left, int right) {
    while (left < right) {
        if (s[left] != s[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}

/***
 * LeetCode 680. 验证回文字符串 II
 * 给定一个非空字符串 s，最多删除一个字符，判断是否能成为回文字符串
 */
bool validPalindrome(string s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        if (s[left] != s[right]) {
            // 尝试删除左边或右边的字符
            return isPalindromeRange(s, left + 1, right) ||
                   isPalindromeRange(s, left, right - 1);
        }
        left++;
    }
}

```

```

        right--;
    }

    return true;
}

/***
 * 辅助函数: 中心扩展法
 */
int expandAroundCenter(string s, int left, int right) {
    while (left >= 0 && right < s.length() && s[left] == s[right]) {
        left--;
        right++;
    }
    return right - left - 1;
}

/***
 * LeetCode 5. 最长回文子串
 * 使用中心扩展法找到最长回文子串
 */
string longestPalindrome(string s) {
    if (s.empty()) return "";

    int start = 0, end = 0;

    for (int i = 0; i < s.length(); i++) {
        // 奇数长度回文
        int len1 = expandAroundCenter(s, i, i);
        // 偶数长度回文
        int len2 = expandAroundCenter(s, i, i + 1);

        int len = (len1 > len2) ? len1 : len2;

        if (len > end - start) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
        }
    }

    return s.substr(start, end - start + 1);
}

```

```
// 测试函数
void testSuperPalindromes() {
    SuperPalindromesSolver solver;

    // 测试用例 1
    string left1 = "4";
    string right1 = "1000";
    int result1 = solver.superpalindromesInRange(left1, right1);
    int result1_table = solver.superpalindromesInRangeTable(left1, right1);

    cout << "测试用例 1:" << endl;
    cout << "范围: [" << left1 << ", " << right1 << "]"
        << endl;
    cout << "枚举法结果: " << result1 << endl;
    cout << "打表法结果: " << result1_table << endl;
    cout << endl;

    // 测试用例 2
    string left2 = "1";
    string right2 = "2";
    int result2 = solver.superpalindromesInRange(left2, right2);
    int result2_table = solver.superpalindromesInRangeTable(left2, right2);

    cout << "测试用例 2:" << endl;
    cout << "范围: [" << left2 << ", " << right2 << "]"
        << endl;
    cout << "枚举法结果: " << result2 << endl;
    cout << "打表法结果: " << result2_table << endl;
    cout << endl;

    // 测试补充题目
    cout << "==== 补充训练题目测试 ===" << endl;

    // 测试回文数判断
    cout << "回文数判断: 12321 -> " << (isPalindromeNumber(12321) ? "是" : "否") << endl;
    cout << "回文数判断: 12345 -> " << (isPalindromeNumber(12345) ? "是" : "否") << endl;

    // 测试最大回文数乘积
    cout << "最大回文数乘积(n=2): " << largestPalindrome(2) << endl;

    // 测试回文串验证
    string testStr = "A man, a plan, a canal: Panama";
    cout << "回文串验证: \""
        << testStr << "\" -> "
        << (isPalindromeString(testStr) ? "是" : "否")
        << endl;
```

```
// 测试最长回文子串
string testStr2 = "babad";
cout << "最长回文子串: \\" << testStr2 << "\\" -> \\" << longestPalindrome(testStr2) << "\\""
<< endl;
}

int main() {
    testSuperPalindromes();
    return 0;
}

/***
 * 算法技巧总结 - C++版本
 *
 * 核心概念:
 * 1. 回文数生成技术:
 *     - 种子生成法: 通过种子数字构造回文数
 *     - 对称性利用: 利用回文数的对称特征
 *     - 数学方法: 避免字符串转换的开销
 *
 * 2. 算法选择策略:
 *     - 小范围查询: 枚举法更节省内存
 *     - 多次查询: 打表法性能最优
 *     - 大数据范围: 需要数学优化和边界处理
 *
 * 3. 性能优化技巧:
 *     - 提前终止: 当结果超出范围时提前结束搜索
 *     - 数学优化: 使用位运算和数学方法
 *     - 缓存策略: 预计算常用结果
 *
 * 调试技巧:
 * 1. 边界值测试: 最小/最大输入、边界情况
 * 2. 中间结果打印: 跟踪算法执行过程
 * 3. 性能分析: 使用 profiler 工具分析瓶颈
 *
 * 工程化实践:
 * 1. 模块化设计: 分离算法逻辑和业务逻辑
 * 2. 异常安全: 确保资源正确释放
 * 3. 代码可读性: 使用有意义的变量名和注释
 */
=====
```

文件: Code02_SuperPalindromes.java

```
=====
```

```
package class043;
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.HashMap;
import java.util.Map;
import java.util.Arrays;
```

```
/**
```

```
* 超级回文数问题
```

```
*
```

```
* 问题描述:
```

```
* 如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。
```

```
* 现在，给定两个正整数 L 和 R（以字符串形式表示），
```

```
* 返回包含在范围 [L, R] 中的超级回文数的数目。
```

```
*
```

```
* 算法思路:
```

```
* 方法 1：枚举法
```

```
* 1. 由于 L 和 R 的范围可达  $10^{18}$ ，直接遍历会超时
```

```
* 2. 考虑到超级回文数是另一个回文数的平方，我们可以枚举较小的回文数
```

```
* 3. 平方根的范围约为  $10^9$ ，但仍太大，继续优化
```

```
* 4. 回文数可以通过“种子”生成，如 123 可以生成 123321(偶数长度)和 12321(奇数长度)
```

```
* 5. 种子的范围约为  $10^5$ ，可以接受
```

```
*
```

```
* 方法 2：打表法
```

```
* 1. 预先计算出所有可能的超级回文数
```

```
* 2. 在查询时直接统计范围内的数量
```

```
* 3. 时间复杂度最低，但需要额外的存储空间
```

```
*
```

```
* 时间复杂度分析:
```

```
* 方法 1：枚举法 -  $O(\sqrt{R} * \log R)$ 
```

```
* - 枚且回文数需要  $O(\sqrt{R})$  时间
```

```
* - 检查每个数是否为回文需要  $O(\log R)$  时间（数的位数）
```

```
*
```

```
* 方法 2：打表法 -  $O(\log R)$ 
```

```
* - 预处理阶段需要  $O(K * \log K)$  时间，其中 K 是超级回文数的个数
```

```
* - 查询阶段只需要在已排序数组中进行二分查找，时间复杂度为  $O(\log K)$ 
```

```
*
```

```
* 空间复杂度分析:
```

```
* 方法 1：枚举法 -  $O(1)$ 
```

```
* - 只需要常数额外空间
```

*

* 方法 2：打表法 - $O(K)$

* - 需要存储所有超级回文数， K 约为 70

*

* 工程化考量：

- * 1. 异常处理：处理大数运算和字符串转换
- * 2. 可配置性：可以调整算法策略（枚举 vs 打表）
- * 3. 性能优化：使用打表法避免重复计算
- * 4. 鲁棒性：处理边界情况和溢出问题

*

* 相关题目：

- * 1. LeetCode 9. 回文数 - <https://leetcode.cn/problems/palindrome-number/>
- * 2. LeetCode 906. 超级回文数 - <https://leetcode.cn/problems/super-palindromes/>
- * 3. LeetCode 479. 最大回文数乘积 - <https://leetcode.cn/problems/largest-palindrome-product/>
- * 4. 牛客网 - 回文数 - <https://www.nowcoder.com/practice/38802713414c4852b6982410c4187dd2>
- * 5. Codeforces 1335D - Anti-Sudoku - <https://codeforces.com/problemset/problem/1335/D>
- * 6. AtCoder ABC136D - Gathering Children - https://atcoder.jp/contests/abc136/tasks/abc136_d
- * 7. 洛谷 P1012 - 拼数 - <https://www.luogu.com.cn/problem/P1012>
- * 8. HackerRank - Almost Palindrome - <https://www.hackerrank.com/challenges/almost-palindrome/problem>

* 9. CodeChef - PALIN - The Next Palindrome - <https://www.codechef.com/problems/PALIN>

* 10. UVa 11795 - Mega Man's Mission -

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=11795

* 11. HDU 1527 - 取石子游戏 - <https://acm.hdu.edu.cn/showproblem.php?pid=1527>

* 12. POJ 1328 - Radar Installation - <http://poj.org/problem?id=1328>

* 13. LintCode 143. 最大异或对 - <https://www.lintcode.com/problem/143/>

* 14. SPOJ - NUMTSN - Numbers of the form x^y - <https://www.spoj.com/problems/NUMTSN/>

* 15. USACO - Milk Patterns - <https://usaco.org/index.php?page=viewproblem2&cpid=218>

* 16. 杭电多校训练 - 超级回文数变种 -

https://acm.hdu.edu.cn/contests/contest_showproblem.php?pid=1003

* 17. Codeforces 1163B2 - Cat Party (Hard Edition) -

<https://codeforces.com/problemset/problem/1163/B2>

* 18. LintCode 200. 最长回文子串 - <https://www.lintcode.com/problem/200/>

* 19. HackerEarth - Palindromic Substrings -

<https://www.hackerearth.com/practice/algorithms/string-algorithm/manachars-algorithm/practice-problems/>

* 20. UVa 10020 - Minimal coverage -

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=10020

* 21. LeetCode 647. 回文子串 - <https://leetcode.cn/problems/palindromic-substrings/>

* 22. LeetCode 336. 回文对 - <https://leetcode.cn/problems/palindrome-pairs/>

* 23. LeetCode 131. 分割回文串 - <https://leetcode.cn/problems/palindrome-partitioning/>

* 24. LeetCode 132. 分割回文串 II - <https://leetcode.cn/problems/palindrome-partitioning-ii/>

* 25. LeetCode 5. 最长回文子串 - <https://leetcode.cn/problems/longest-palindromic-substring/>

```

* 26. LeetCode 516. 最长回文子序列 - https://leetcode.cn/problems/longest-palindromic-
subsequence/
* 27. 牛客网 - 分割回文串 - https://www.nowcoder.com/practice/9f3231a991af4f55b95579b44b7a01ba
* 28. 牛客网 - 复原 IP 地址 - https://www.nowcoder.com/practice/ce73540d47374dbe85b3125f57727e1e
* 29. Codeforces 1327D. Infinite Path - https://codeforces.com/problemset/problem/1327/D
* 30. Codeforces 1436E. Complicated Computations -
https://codeforces.com/problemset/problem/1436/E
* 31. 洛谷 P1048. 采药 - https://www.luogu.com.cn/problem/P1048
* 32. 洛谷 P1125. 笨小猴 - https://www.luogu.com.cn/problem/P1125
* 33. HackerRank - Split the String - https://www.hackerrank.com/challenges/split-the-
string/problem
* 34. UVa 10189 - Minesweeper -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=10189
* 35. POJ 1163 - The Triangle - http://poj.org/problem?id=1163
* 36. HDU 2000 - ASCII 码排序 - https://acm.hdu.edu.cn/showproblem.php?pid=2000
* 37. LintCode 125. 背包问题 II - https://www.lintcode.com/problem/125/
* 38. LintCode 200. 最长回文子串 - https://www.lintcode.com/problem/200/
* 39. LintCode 130. 堆化 - https://www.lintcode.com/problem/130/
* 40. AcWing 901. 滑雪 - https://www.acwing.com/problem/content/903/
* 41. AcWing 1482. 进制 - https://www.acwing.com/problem/content/1484/
*/
public class Code02_SuperPalindromes {

    /**
     * 方法 1：枚举法
     *
     * @param left 范围左边界（字符串形式）
     * @param right 范围右边界（字符串形式）
     * @return 范围内超级回文数的个数
     */
    public static int superpalindromesInRange(String left, String right) {
        long l = Long.valueOf(left);
        long r = Long.valueOf(right);
        // l....r long
        // x 根号，范围 limit
        long limit = (long) Math.sqrt((double) r);
        // seed : 枚举量很小, 10^18 -> 10^9 -> 10^5
        // seed : 奇数长度回文、偶数长度回文
        long seed = 1;
        // num : 根号 x, num^2 -> x
        long num = 0;
        int ans = 0;
        do {

```

```
// seed 生成偶数长度回文数字
// 123 -> 123321
num = evenEnlarge(seed);
if (check(num * num, 1, r)) {
    ans++;
}
// seed 生成奇数长度回文数字
// 123 -> 12321
num = oddEnlarge(seed);
if (check(num * num, 1, r)) {
    ans++;
}
// 123 -> 124 -> 125
seed++;
} while (num < limit);
return ans;
}

/**
 * 根据种子扩充到偶数长度的回文数字并返回
 *
 * @param seed 种子数字
 * @return 偶数长度的回文数
 *
 * 例如: seed=123, 返回 123321
 */
public static long evenEnlarge(long seed) {
    long ans = seed;
    while (seed != 0) {
        ans = ans * 10 + seed % 10;
        seed /= 10;
    }
    return ans;
}

/**
 * 根据种子扩充到奇数长度的回文数字并返回
 *
 * @param seed 种子数字
 * @return 奇数长度的回文数
 *
 * 例如: seed=123, 返回 12321
 */

```

```

public static long oddEnlarge(long seed) {
    long ans = seed;
    seed /= 10;
    while (seed != 0) {
        ans = ans * 10 + seed % 10;
        seed /= 10;
    }
    return ans;
}

/***
 * 检查数字是否在范围内且为回文数
 *
 * @param ans 待检查的数字
 * @param l 范围左边界
 * @param r 范围右边界
 * @return 如果在范围内且为回文数返回 true, 否则返回 false
 */
public static boolean check(long ans, long l, long r) {
    return ans >= l && ans <= r && isPalindrome(ans);
}

/***
 * 验证 long 类型的数字 num, 是不是回文数字
 *
 * @param num 待检查的数字
 * @return 如果是回文数返回 true, 否则返回 false
 *
 * 算法思路:
 * 1. 通过 offset 定位最高位
 * 2. 比较最高位和最低位是否相等
 * 3. 去掉最高位和最低位, 继续比较
 * 4. 直到所有位都比较完毕
 */
public static boolean isPalindrome(long num) {
    long offset = 1;
    // 注意这么写是为了防止溢出
    while (num / offset >= 10) {
        offset *= 10;
    }
    // num      : 52725
    // offset : 10000
    // 首尾判断
}

```

```

while (num != 0) {
    if (num / offset != num % 10) {
        return false;
    }
    num = (num % offset) / 10;
    offset /= 100;
}
return true;
}

/**
 * 方法 2：打表法（最优解）
 *
 * @param left 范围左边界（字符串形式）
 * @param right 范围右边界（字符串形式）
 * @return 范围内超级回文数的个数
 *
 * 算法思路：
 * 1. 预先计算出所有的超级回文数并存储在数组中
 * 2. 在查询时，找到范围内第一个和最后一个超级回文数的位置
 * 3. 通过位置差计算数量
 *
 * 优势：
 * 1. 时间复杂度最低
 * 2. 避免重复计算
 * 3. 适合多次查询的场景
 */
public static int superpalindromesInRange2(String left, String right) {
    long l = Long.parseLong(left);
    long r = Long.parseLong(right);
    int i = 0;
    for (; i < record.length; i++) {
        if (record[i] >= 1) {
            break;
        }
    }
    int j = record.length - 1;
    for (; j >= 0; j--) {
        if (record[j] <= r) {
            break;
        }
    }
    return j - i + 1;
}

```

}

```
// 预计算的所有超级回文数（已排序）
public static long[] record = new long[] {
    1L,
    4L,
    9L,
    121L,
    484L,
    10201L,
    12321L,
    14641L,
    40804L,
    44944L,
    1002001L,
    1234321L,
    4008004L,
    100020001L,
    102030201L,
    104060401L,
    12102420121L,
    123454321L,
    125686521L,
    400080004L,
    404090404L,
    10000200001L,
    10221412201L,
    12102420121L,
    12345654321L,
    40000800004L,
    1000002000001L,
    1002003002001L,
    1004006004001L,
    1020304030201L,
    1022325232201L,
    1024348434201L,
    1210024200121L,
    1212225222121L,
    1214428244121L,
    1232346432321L,
    1234567654321L,
    4000008000004L,
    4004009004004L,
```

100000020000001L,
100220141022001L,
102012040210201L,
102234363432201L,
121000242000121L,
121242363242121L,
123212464212321L,
123456787654321L,
400000080000004L,
1000000200000001L,
10002000300020001L,
10004000600040001L,
10020210401202001L,
10022212521222001L,
10024214841242001L,
10201020402010201L,
10203040504030201L,
10205060806050201L,
10221432623412201L,
10223454745432201L,
12100002420000121L,
12102202520220121L,
12104402820440121L,
12122232623222121L,
12124434743442121L,
12321024642012321L,
12323244744232321L,
12343456865434321L,
12345678987654321L,
40000000800000004L,
40004000900040004L,
1000000002000000001L,
1000220014100220001L,
1002003004003002001L,
1002223236323222001L,
1020100204020010201L,
1020322416142230201L,
1022123226223212201L,
1022345658565432201L,
1210000024200000121L,
1210242036302420121L,
1212203226223022121L,
1212445458545442121L,

```
1232100246420012321L,
1232344458544432321L,
1234323468643234321L,
4000000008000000004L

};

/***
 * 用于收集所有超级回文数的方法（生成 record 数组）
 *
 * @return 包含所有超级回文数的列表
 */
public static List<Long> collect() {
    long l = 1;
    long r = Long.MAX_VALUE;
    long limit = (long) Math.sqrt((double) r);
    long seed = 1;
    long enlarge = 0;
    ArrayList<Long> ans = new ArrayList<>();
    do {
        enlarge = evenEnlarge(seed);
        if (check(enlarge * enlarge, l, r)) {
            ans.add(enlarge * enlarge);
        }
        enlarge = oddEnlarge(seed);
        if (check(enlarge * enlarge, l, r)) {
            ans.add(enlarge * enlarge);
        }
        seed++;
    } while (enlarge < limit);
    ans.sort((a, b) -> a.compareTo(b));
    return ans;
}

public static void main(String[] args) {
    // 用于生成 record 数组
    /*
    List<Long> ans = collect();
    for (long p : ans) {
        System.out.println(p + "L,");
    }
    System.out.println("size : " + ans.size());
    */
}
```

```
// ===== 补充训练题目 ======
```

```
/**
```

```
* 补充题目 1: LeetCode 479. 最大回文数乘积
```

```
* 题目描述: 给定一个整数 n, 返回可表示为两个 n 位数的乘积的最大回文整数。因为答案可能非常大, 所以返回它对 1337 取余。
```

```
* 链接: https://leetcode.cn/problems/largest-palindrome-product/
```

```
*
```

```
* 解题思路:
```

```
* - 从最大的 n 位数开始, 向下枚举所有可能的回文数
```

```
* - 检查每个回文数是否可以分解为两个 n 位数的乘积
```

```
* - 时间复杂度: O(10^n), 空间复杂度: O(1)
```

```
*/
```

```
public static int largestPalindrome(int n) {
```

```
    if (n == 1) {
```

```
        return 9;
```

```
}
```

```
// 计算 n 位数的范围
```

```
long upper = (long) Math.pow(10, n) - 1;
```

```
long lower = (long) Math.pow(10, n - 1);
```

```
// 从大到小枚举回文数
```

```
for (long left = upper; left >= lower; left--) {
```

```
    // 构造回文数
```

```
    String s = String.valueOf(left);
```

```
    String rev = new StringBuilder(s).reverse().toString();
```

```
    long palindrome = Long.parseLong(s + rev);
```

```
// 检查是否可以分解为两个 n 位数的乘积
```

```
for (long i = upper; i * i >= palindrome; i--) {
```

```
    if (palindrome % i == 0) {
```

```
        long j = palindrome / i;
```

```
        if (j >= lower && j <= upper) {
```

```
            return (int) (palindrome % 1337);
```

```
}
```

```
}
```

```
}
```

```
return -1;
```

```
}
```

```
/**
```

- * 补充题目 2: LeetCode 680. 验证回文字符串 II
- * 题目描述: 给你一个字符串 s, 最多可以从中删除一个字符, 请判断是否能成为回文字符串。
- * 链接: <https://leetcode.cn/problems/valid-palindrome-ii/>
- *
- * 解题思路:
- * - 使用双指针法, 从两端向中间比较
- * - 当遇到不匹配的字符时, 尝试删除左边或右边的字符后继续判断
- * - 时间复杂度: O(n), 空间复杂度: O(1)
- */

```
public static boolean validPalindrome(String s) {  
    int left = 0, right = s.length() - 1;  
    while (left < right) {  
        if (s.charAt(left) != s.charAt(right)) {  
            // 尝试删除左边字符或右边字符后继续检查  
            return isPalindrome(s, left + 1, right) || isPalindrome(s, left, right - 1);  
        }  
        left++;  
        right--;  
    }  
    return true;  
}
```

```
private static boolean isPalindrome(String s, int left, int right) {  
    while (left < right) {  
        if (s.charAt(left) != s.charAt(right)) {  
            return false;  
        }  
        left++;  
        right--;  
    }  
    return true;  
}
```

```
/**  
 * 补充题目 3: CodeChef - PALIN - The Next Palindrome  
 * 题目描述: 给定一个正整数 N, 找出大于 N 的最小回文数。  
 * 链接: https://www.codechef.com/problems/PALIN  
 *  
 * 解题思路:  
 * - 生成下一个回文数的高效方法  
 * - 不需要检查每个数, 而是直接构造下一个可能的回文数  
 * - 时间复杂度: O(n), 其中 n 是数字的位数  
 */
```

```
public static String nextPalindrome(String n) {  
    char[] digits = n.toCharArray();  
    int len = digits.length;  
    int mid = len / 2;  
    int i = mid - 1;  
    int j = len % 2 == 0 ? mid : mid + 1;  
  
    // 检查是否所有数字都是 9  
    boolean allNines = true;  
    for (char c : digits) {  
        if (c != '9') {  
            allNines = false;  
            break;  
        }  
    }  
    if (allNines) {  
        StringBuilder sb = new StringBuilder();  
        sb.append('1');  
        for (i = 0; i < len - 1; i++) {  
            sb.append('0');  
        }  
        sb.append('1');  
        return sb.toString();  
    }  
  
    // 找到第一个可以增加的位置  
    while (i >= 0 && digits[i] == digits[j]) {  
        i--;  
        j++;  
    }  
  
    boolean leftLess = false;  
    if (i < 0 || digits[i] < digits[j]) {  
        leftLess = true;  
    }  
  
    // 复制左侧到右侧  
    while (i >= 0) {  
        digits[j++] = digits[i--];  
    }  
  
    // 如果需要进位  
    if (leftLess) {
```

```

    i = mid - 1;
    int carry = 1;
    if (len % 2 == 1) {
        digits[mid] += carry;
        carry = digits[mid] / 10;
        digits[mid] %= 10;
        j = mid + 1;
    } else {
        j = mid;
    }

    while (i >= 0 && carry > 0) {
        digits[i] += carry;
        carry = digits[i] / 10;
        digits[i] %= 10;
        digits[j++] = digits[i--];
    }
}

return new String(digits);
}

/***
 * 补充题目 4: HackerRank - The Longest Palindromic Subsequence
 * 题目描述: 给定一个字符串, 找出其中最长的回文子序列的长度。
 * 链接: https://www.hackerrank.com/challenges/longest-palindromic-subsequence/problem
 *
 * 解题思路:
 * - 使用动态规划, dp[i][j]表示子串 s[i... j]的最长回文子序列长度
 * - 状态转移方程: 如果 s[i] == s[j], 则 dp[i][j] = dp[i+1][j-1] + 2; 否则 dp[i][j] =
max(dp[i+1][j], dp[i][j-1])
 * - 时间复杂度: O(n^2), 空间复杂度: O(n^2)
 */
public static int longestPalindromeSubseq(String s) {
    int n = s.length();
    int[][] dp = new int[n][n];

    // 单个字符的回文子序列长度为 1
    for (int i = 0; i < n; i++) {
        dp[i][i] = 1;
    }

    // 填充 dp 数组

```

```

        for (int len = 2; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                if (s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                } else {
                    dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[0][n - 1];
    }
}

```

/**

* 补充题目 5: LintCode 1178. 有效的回文

* 题目描述: 给定一个字符串, 判断它是否是回文串。只考虑字母和数字字符, 可以忽略字母的大小写。

* 链接: <https://www.lintcode.com/problem/1178/>

*

* 解题思路:

- * - 使用双指针法, 从两端向中间比较
- * - 只考虑字母和数字字符, 忽略大小写
- * - 时间复杂度: O(n), 空间复杂度: O(1)

*/

```

public static boolean isPalindrome(String s) {
    if (s == null || s.isEmpty()) {
        return true;
    }
}

```

```
int left = 0, right = s.length() - 1;
```

```
while (left < right) {
```

```
    // 跳过非字母数字字符
```

```
    while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {
```

```
        left++;
    }
}

```

```
    while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {
```

```
        right--;
    }
}

```

```
    if (left < right) {
```

```
        // 比较字符 (忽略大小写)
```

```
        if (Character.toLowerCase(s.charAt(left)) !=
```

```
Character.toLowerCase(s.charAt(right))) {
```

```

        return false;
    }
    left++;
    right--;
}
}

return true;
}

/**
 * 补充题目 6: LeetCode 647. 回文子串
 * 题目描述: 给你一个字符串 s，请你统计并返回这个字符串中 回文子串 的数目。
 * 回文字符串 是正着读和倒过来读一样的字符串。子字符串 是字符串中的由连续字符组成的一个序列。
 * 链接: https://leetcode.cn/problems/palindromic-substrings/
 *
 * 解题思路:
 * - 使用中心扩展法，遍历每个可能的中心点
 * - 考虑奇数长度和偶数长度的回文串
 * - 时间复杂度: O(n^2)，空间复杂度: O(1)
 */
public static int countSubstrings(String s) {
    int count = 0;
    for (int i = 0; i < s.length(); i++) {
        // 以单个字符为中心的奇数长度回文串
        count += expandAroundCenter(s, i, i);
        // 以两个字符之间为中心的偶数长度回文串
        count += expandAroundCenter(s, i, i + 1);
    }
    return count;
}

private static int expandAroundCenter(String s, int left, int right) {
    int count = 0;
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
        count++;
        left--;
        right++;
    }
    return count;
}

/**

```

* 补充题目 7: LeetCode 336. 回文对

* 题目描述: 给定一组互不相同的单词, 找出所有不同的索引对 (i, j) , 使得单词 $\text{words}[i] + \text{words}[j]$ 是一个回文串。

* 链接: <https://leetcode.cn/problems/palindrome-pairs/>

*

* 解题思路:

* - 使用哈希表存储单词的逆序和索引

* - 对于每个单词, 尝试在哈希表中查找可以拼接形成回文的部分

* - 时间复杂度: $O(n * k^2)$, 其中 n 是单词数量, k 是单词的最大长度

* - 空间复杂度: $O(n * k)$

*/

```
public static List<List<Integer>> palindromePairs(String[] words) {  
    List<List<Integer>> result = new ArrayList<>();  
    Map<String, Integer> map = new HashMap<>();  
  
    // 存储单词的逆序和索引  
    for (int i = 0; i < words.length; i++) {  
        map.put(new StringBuilder(words[i]).reverse().toString(), i);  
    }  
  
    for (int i = 0; i < words.length; i++) {  
        String word = words[i];  
        int len = word.length();  
  
        // 情况 1: 空字符串与回文单词拼接  
        if (map.containsKey("") && !word.isEmpty() && isPalindrome(word)) {  
            result.add(Arrays.asList(i, map.get("")));  
            result.add(Arrays.asList(map.get(""), i));  
        }  
  
        // 情况 2: 一个单词是另一个单词的逆序  
        if (map.containsKey(word) && map.get(word) != i) {  
            result.add(Arrays.asList(i, map.get(word)));  
        }  
  
        // 情况 3: 单词的前缀是回文, 查找剩余部分的逆序  
        for (int j = 1; j < len; j++) {  
            String left = word.substring(0, j);  
            String right = word.substring(j);  
  
            // 检查左半部分是否为回文, 如果是, 则查找右半部分的逆序  
            if (isPalindrome(left) && map.containsKey(right)) {  
                result.add(Arrays.asList(map.get(right), i));  
            }  
        }  
    }  
}
```

```

    }

    // 检查右半部分是否为回文，如果是，则查找左半部分的逆序
    if (isPalindrome(right) && map.containsKey(left)) {
        result.add(Arrays.asList(i, map.get(left)));
    }
}

return result;
}

/***
 * 补充题目 8: HackerRank - Split the String
 * 题目描述：给定一个字符串，将其分割为 k 个回文子串，求最大可能的 k 值。
 * 链接：https://www.hackerrank.com/challenges/split-the-string/problem
 *
 * 解题思路：
 * - 使用贪心策略，尽可能多地分割
 * - 每次都尝试从当前位置分割出最短的回文子串
 * - 时间复杂度：O(n^2)，空间复杂度：O(n)
 */
public static int splitTheString(String s) {
    int n = s.length();
    int count = 0;
    int start = 0;

    while (start < n) {
        // 尝试分割出最短的回文子串（长度为 1 或 2）
        if (start + 1 < n && s.charAt(start) == s.charAt(start + 1)) {
            // 长度为 2 的回文
            start += 2;
            count++;
        } else {
            // 长度为 1 的回文
            start++;
            count++;
        }
    }

    return count;
}

```

```
/**  
 * 超级回文数与回文序列生成的算法技巧总结  
 *  
 * 核心概念与理论基础:  
 * 1. 回文数理论:  
 *   - 定义: 正序和倒序读都是一样的数字  
 *   - 分类: 根据长度分为奇数长度和偶数长度回文数  
 *   - 分布: 回文数在数字空间中呈稀疏分布  
 *   - 数学性质: 回文数的平方不一定是回文数, 反之亦然  
 *  
 * 2. 超级回文数特性:  
 *   - 定义: 既是回文数, 又是某个回文数的平方  
 *   - 结构特征: 通过种子数生成, 具有高度对称性  
 *   - 数值范围: 在  $10^{18}$  范围内仅有有限数量的超级回文数  
 *   - 唯一性: 每个超级回文数都有唯一的生成方式  
 *  
 * 3. 回文序列理论:  
 *   - 子串 vs 子序列: 回文子串要求连续, 回文子序列不要求连续  
 *   - 最优子结构: 回文问题通常具有最优子结构性质  
 *   - 重叠子问题: 动态规划方法的理论基础  
 *   - 对偶性: 字符串反转后的性质与原字符串的关系  
 *  
 * 算法设计与策略选择:  
 * 1. 生成策略:  
 *   - 种子扩展法: 通过构建回文数种子, 扩展生成完整回文数  
 *   - 中间扩展法: 从中心向两侧扩展生成回文串  
 *   - 递推构造法: 基于已知回文数构造更大的回文数  
 *   - 二分查找法: 在回文数集合中进行高效查询  
 *  
 * 2. 判断算法:  
 *   - 双指针法: 从两端向中间比较, 时间  $O(n)$ , 空间  $O(1)$   
 *   - 反转比较法: 生成反转字符串进行比较, 时间  $O(n)$ , 空间  $O(n)$   
 *   - 数值反转法: 直接操作数字进行反转, 避免字符串转换  
 *   - Manacher 算法: 高效查找所有回文子串, 时间  $O(n)$   
 *  
 * 3. 优化技术:  
 *   - 预计算优化: 通过预计算和打表法加速查询  
 *   - 剪枝策略: 在生成过程中提前排除无效情况  
 *   - 位运算优化: 利用位操作减少计算复杂度  
 *   - 并行计算: 将生成和验证过程并行化  
 *  
 * 高级优化技术:  
 * 1. 数学优化:
```

- * - 模运算加速：利用模运算性质减少计算量
- * - 数值溢出防护：提前检测可能的溢出情况
- * - 周期性利用：利用回文数生成的周期性规律
- *
- * 2. 数据结构优化：
 - * - 哈希表应用：存储逆序字符串以快速查找匹配
 - * - 字典树结构：高效存储和查询前缀/后缀信息
 - * - 稀疏数组：优化存储大规模稀疏数据
- *
- * 3. 算法融合：
 - * - 回溯与剪枝：结合使用以减少搜索空间
 - * - 动态规划与贪心：根据问题特性选择合适方法
 - * - 分支界限：结合启发式函数指导搜索
- *
- * 工程化最佳实践：
 - * 1. 代码组织与模块化：
 - * - 关注点分离：将生成、判断和查询逻辑分离
 - * - 可重用组件：设计通用的回文处理组件
 - * - 接口抽象：定义清晰的接口便于扩展
 - *
 - * 2. 性能调优技巧：
 - * - 减少对象创建：复用对象和数组
 - * - 缓存中间结果：使用记忆化避免重复计算
 - * - 批量处理：合并小操作提高吞吐量
 - * - JVM 优化：针对 Java 特性进行代码优化
 - *
 - * 3. 健壮性保障：
 - * - 输入验证：全面检查输入边界和异常情况
 - * - 异常处理：妥善处理各种错误情况
 - * - 资源管理：避免内存泄漏和资源浪费
 - * - 并发安全：确保多线程环境下的正确性
 - *
 - * 调试与测试技术：
 - * 1. 测试用例设计：
 - * - 边界测试：空字符串、单字符、全相同字符
 - * - 典型场景：各种长度和结构的回文数
 - * - 性能测试：大规模数据下的性能表现
 - * - 压力测试：边界条件和极限情况
 - *
 - * 2. 调试技巧：
 - * - 可视化输出：打印中间状态辅助调试
 - * - 断言检查：使用断言验证关键假设
 - * - 性能分析：识别瓶颈并优化

- * - 日志记录：详细记录执行流程
- *
- * 跨语言实现比较：
 - * 1. Java 实现：
 - 优势：丰富的集合库，自动内存管理
 - 限制：处理大整数可能需要 BigInteger，性能开销较大
 - 最佳实践：使用 StringBuilder 进行字符串操作，避免频繁字符串拼接
 - *
 - * 2. C++实现：
 - 优势：更高的执行效率，直接内存访问
 - 限制：需要手动管理内存，代码复杂度较高
 - 最佳实践：使用 std::string_view 避免不必要的拷贝，使用智能指针管理资源
 - *
 - * 3. Python 实现：
 - 优势：简洁的语法，强大的字符串处理功能
 - 限制：执行速度相对较慢，递归深度受限
 - 最佳实践：使用字符串切片和生成器表达式，利用 lru_cache 进行记忆化
 - *
 - * 4. Go 实现：
 - 优势：并发支持，编译型性能，简洁语法
 - 限制：泛型支持相对有限，库生态不如 Java 丰富
 - 最佳实践：使用切片而非数组，利用 goroutine 进行并行处理
 - *
- * 与现代技术的融合：
 - * 1. 算法与 AI 结合：
 - 深度学习应用：使用神经网络识别复杂模式的回文结构
 - 强化学习优化：通过学习自动调整搜索策略
 - 生成对抗网络：生成符合特定模式的回文序列
 - *
 - * 2. 分布式计算：
 - MapReduce 模式：将大规模回文搜索任务分布式处理
 - 并行生成：利用多节点并行生成和验证回文数
 - 分布式缓存：构建分布式缓存系统存储中间结果
 - *
 - * 3. 区块链技术：
 - 哈希链构造：利用回文结构构造特殊哈希链
 - 共识算法：基于回文验证的共识机制
 - 数据完整性：利用回文特性验证数据完整性
 - *
- * 进阶研究方向：
 - * 1. 理论研究：
 - 回文数分布规律：探索回文数在大数域中的分布特性
 - 超级回文数生成函数：寻找超级回文数的数学生成公式

```
*      - 计算复杂度分析：深入研究不同回文问题的复杂度
*
* 2. 应用拓展：
*      - 密码学应用：基于回文特性设计安全机制
*      - 数据压缩：利用回文结构进行数据压缩
*      - 模式识别：将回文检测技术应用于更广泛的领域
*
* 3. 算法创新：
*      - 量子算法：探索量子计算在回文问题中的应用
*      - 近似算法：设计针对 NP 难回文问题的高效近似算法
*      - 随机化算法：结合概率方法提高回文搜索效率
*
* 总结：
* 超级回文数问题虽然看似简单，但涉及到广泛的算法技术和优化策略。通过系统学习和掌握这些技巧，不仅可以高效解决回文相关问题，还能将这些思想应用到更广泛的算法设计中。在实际工程中，应根据具体问题特性选择合适的算法和优化策略，平衡时间复杂度、空间复杂度和实现复杂度，构建高效、健壮、可维护的解决方案。
*/
}
```

}

=====

文件：Code02_SuperPalindromes.py

=====

"""

超级回文数问题 - Python 版本

问题描述：

如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。

给定两个正整数 L 和 R（以字符串形式表示），返回包含在范围 [L, R] 中的超级回文数的数目。

算法思路：

方法 1：枚举法 - 通过生成回文数来优化搜索

方法 2：打表法 - 预计算所有可能的超级回文数

时间复杂度分析：

- 枚举法： $O(\sqrt{R} * \log R)$ ，其中 \sqrt{R} 为平方根范围， $\log R$ 为回文判断时间

- 打表法：预处理 $O(K * \log K)$ ，查询 $O(\log K)$ ，其中 K 为超级回文数个数（约 70）

空间复杂度分析：

- 枚举法： $O(1)$ ，常数额外空间

- 打表法： $O(K)$ ，存储预计算的超级回文数

工程化考量：

1. 大数处理：使用 Python 内置大整数支持
2. 边界处理：处理 L 和 R 的边界情况
3. 性能优化：选择合适的算法策略
4. 可测试性：设计全面的测试用例

"""

```
import math
from typing import List

class SuperPalindromesSolver:
    def __init__(self):
        # 预计算的超级回文数数组
        self.super_palindromes = [
            1, 4, 9, 121, 484, 10201, 12321, 14641, 40804, 44944,
            1002001, 1234321, 4008004, 100020001, 102030201, 104060401,
            121242121, 123454321, 125686521, 400080004, 404090404,
            10000200001, 10221412201, 12102420121, 12345654321, 40000800004,
            1000002000001, 1002003002001, 1004006004001, 1020304030201,
            1022325232201, 1024348434201, 1210024200121, 1212225222121,
            1214428244121, 1232346432321, 1234567654321, 4000008000004,
            4004009004004, 100000020000001, 100220141022001, 102012040210201,
            102234363432201, 121000242000121, 121242363242121, 123212464212321,
            123456787654321, 400000080000004, 10000000200000001, 10002000300020001,
            10004000600040001, 10020210401202001, 10022212521222001, 10024214841242001,
            10201020402010201, 10203040504030201, 10205060806050201, 10221432623412201,
            10223454745432201, 12100002420000121, 12102202520220121, 12104402820440121,
            12122232623222121, 12124434743442121, 12321024642012321, 12323244744232321,
            12343456865434321, 12345678987654321, 40000000800000004, 40004000900040004
        ]
    
    def superpalindromes_in_range(self, left: str, right: str) -> int:
        """
        方法 1：枚举法
        通过生成回文数来优化搜索，避免直接遍历大范围
        """
        
        L = int(left)
        R = int(right)
        count = 0
        
        # 计算种子的上界（对于 10^18 范围，种子只需到 10^5 级别）
        seed_limit = 100000 # 10^5
```

```

# 枚举所有可能的种子
for seed in range(1, seed_limit):
    # 生成偶数长度的回文数
    even_pal = self.even_enlarge(seed)
    square_even = even_pal * even_pal

    # 检查是否在范围内且是回文数
    if L <= square_even <= R and self.is_palindrome(square_even):
        count += 1

    # 生成奇数长度的回文数
    odd_pal = self.odd_enlarge(seed)
    square_odd = odd_pal * odd_pal

    # 检查是否在范围内且是回文数
    if L <= square_odd <= R and self.is_palindrome(square_odd):
        count += 1

    # 如果生成的平方已经超过 R, 可以提前终止
    if square_even > R and square_odd > R:
        break

return count

```

```
def superpalindromes_in_range_table(self, left: str, right: str) -> int:
```

```
"""

```

方法 2: 打表法 (最优解)

预计算所有可能的超级回文数, 查询时直接统计

```
"""

```

```
L = int(left)
```

```
R = int(right)
```

```
count = 0
```

```
# 统计在范围内的超级回文数
```

```
for num in self.super_palindromes:
```

```
    if L <= num <= R:
```

```
        count += 1
```

```
return count
```

```
def even_enlarge(self, seed: int) -> int:
```

```
"""

```

根据种子扩充到偶数长度的回文数字

例如: seed=123, 返回 123321

"""

```
ans = seed
temp = seed
while temp > 0:
    ans = ans * 10 + temp % 10
    temp //= 10
return ans
```

def odd_enlarge(self, seed: int) -> int:

"""

根据种子扩充到奇数长度的回文数字

例如: seed=123, 返回 12321

"""

```
ans = seed
temp = seed // 10
while temp > 0:
    ans = ans * 10 + temp % 10
    temp //= 10
return ans
```

def is_palindrome(self, x: int) -> bool:

"""

判断一个数是否是回文数

使用数学方法避免字符串转换

"""

```
if x < 0:
    return False
if x < 10:
    return True
```

original = x

reversed_num = 0

while x > 0:

```
    reversed_num = reversed_num * 10 + x % 10
    x //= 10
```

return original == reversed_num

补充训练题目 - Python 实现

```

def is_palindrome_number(x: int) -> bool:
    """
    LeetCode 9. 回文数
    判断一个整数是否是回文数
    """
    if x < 0:
        return False
    if x < 10:
        return True

    original = x
    reversed_num = 0

    while x > 0:
        reversed_num = reversed_num * 10 + x % 10
        x //= 10

    return original == reversed_num

def largest_palindrome(n: int) -> int:
    """
    LeetCode 479. 最大回文数乘积
    给定一个整数 n，返回可表示为两个 n 位整数乘积的最大回文整数
    """
    if n == 1:
        return 9

    max_num = 10**n - 1
    min_num = 10**n - 1

    for i in range(max_num, min_num - 1, -1):
        # 创建回文数
        s = str(i)
        palindrome = int(s + s[::-1])

        # 检查是否可以分解为两个 n 位数的乘积
        for j in range(max_num, int(palindrome**0.5) - 1, -1):
            if palindrome % j == 0:
                factor = palindrome // j
                if min_num <= factor <= max_num:
                    return palindrome % 1337

    return -1

```

```

def is_palindrome_string(s: str) -> bool:
    """
    LeetCode 125. 验证回文串
    给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写
    """
    left, right = 0, len(s) - 1

    while left < right:
        # 跳过非字母数字字符
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1

        # 比较字符（忽略大小写）
        if s[left].lower() != s[right].lower():
            return False

        left += 1
        right -= 1

    return True

def valid_palindrome(s: str) -> bool:
    """
    LeetCode 680. 验证回文字符串 II
    给定一个非空字符串 s，最多删除一个字符，判断是否能成为回文字符串
    """
    def is_palindrome_range(left: int, right: int) -> bool:
        while left < right:
            if s[left] != s[right]:
                return False
            left += 1
            right -= 1
        return True

    left, right = 0, len(s) - 1

    while left < right:
        if s[left] != s[right]:
            # 尝试删除左边或右边的字符
            return is_palindrome_range(left + 1, right) or is_palindrome_range(left, right - 1)
        left += 1
        right -= 1

```

```

    left += 1
    right -= 1

return True

def longest_palindrome(s: str) -> str:
"""
LeetCode 5. 最长回文子串
使用中心扩展法找到最长回文子串
"""

if not s:
    return ""

def expand_around_center(left: int, right: int) -> int:
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return right - left - 1

start, end = 0, 0

for i in range(len(s)):
    # 奇数长度回文
    len1 = expand_around_center(i, i)
    # 偶数长度回文
    len2 = expand_around_center(i, i + 1)

    max_len = max(len1, len2)

    if max_len > end - start:
        start = i - (max_len - 1) // 2
        end = i + max_len // 2

return s[start:end + 1]

def count_substrings(s: str) -> int:
"""
LeetCode 647. 回文子串
统计字符串中的回文子串数目
"""

def expand_around_center(left: int, right: int) -> int:
    count = 0
    while left >= 0 and right < len(s) and s[left] == s[right]:

```

```

        count += 1
        left -= 1
        right += 1
    return count

total = 0
for i in range(len(s)):
    # 奇数长度回文子串
    total += expand_around_center(i, i)
    # 偶数长度回文子串
    total += expand_around_center(i, i + 1)

return total

```

```
def longest_palindrome_subseq(s: str) -> int:
```

```
"""

```

LeetCode 516. 最长回文子序列

找出字符串中最长的回文子序列的长度

```
"""

```

```
n = len(s)
```

```
# dp[i][j]表示 s[i:j+1] 的最长回文子序列长度
```

```
dp = [[0] * n for _ in range(n)]
```

```
# 单个字符的回文子序列长度为 1
```

```
for i in range(n):
```

```
    dp[i][i] = 1
```

```
# 从短到长填充 dp 数组
```

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

```
    for i in range(n - length + 1):
```

```
        j = i + length - 1
```

```
        if s[i] == s[j]:
```

```
            dp[i][j] = dp[i + 1][j - 1] + 2
```

```
        else:
```

```
            dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])
```

```
return dp[0][n - 1]
```

```
def test_super_palindromes():
```

```
    """测试函数"""

```

```
solver = SuperPalindromesSolver()
```

```
# 测试用例 1
```

```
left1 = "4"
right1 = "1000"
result1 = solver.superpalindromes_in_range(left1, right1)
result1_table = solver.superpalindromes_in_range_table(left1, right1)

print("测试用例 1:")
print(f"范围: [{left1}, {right1}]")
print(f"枚举法结果: {result1}")
print(f"打表法结果: {result1_table}")
print()

# 测试用例 2
left2 = "1"
right2 = "2"
result2 = solver.superpalindromes_in_range(left2, right2)
result2_table = solver.superpalindromes_in_range_table(left2, right2)

print("测试用例 2:")
print(f"范围: [{left2}, {right2}]")
print(f"枚举法结果: {result2}")
print(f"打表法结果: {result2_table}")
print()

# 测试补充题目
print("== 补充训练题目测试 ==")

# 测试回文数判断
print(f"回文数判断: 12321 -> {'是' if is_palindrome_number(12321) else '否'}")
print(f"回文数判断: 12345 -> {'是' if is_palindrome_number(12345) else '否'}")

# 测试最大回文数乘积
print(f"最大回文数乘积(n=2): {largest_palindrome(2)}")

# 测试回文串验证
test_str = "A man, a plan, a canal: Panama"
print(f"回文串验证: '{test_str}' -> {'是' if is_palindrome_string(test_str) else '否'}")

# 测试最长回文子串
test_str2 = "babad"
print(f"最长回文子串: '{test_str2}' -> {longest_palindrome(test_str2)}")

# 测试回文子串统计
test_str3 = "abc"
```

```
print(f"回文子串统计: '{test_str3}' -> {count_substrings(test_str3)}")\n\n# 测试最长回文子序列\ntest_str4 = "bbbab"\nprint(f"最长回文子序列: '{test_str4}' -> {longest_palindrome_subseq(test_str4)}")\n\nif __name__ == "__main__":\n    test_super_palindromes()\n\n"""\n
```

算法技巧总结 – Python 版本

核心概念:

1. 回文数生成技术:

- 种子生成法: 通过种子数字构造回文数
- 对称性利用: 利用回文数的对称特征
- 数学方法: 避免字符串转换的开销

2. Python 特有优势:

- 内置大整数支持: 无需担心数值溢出
- 字符串操作简便: [::-1]快速反转字符串
- 动态类型: 灵活的数值处理

3. 算法选择策略:

- 小范围查询: 枚举法更节省内存
- 多次查询: 打表法性能最优
- 大数据范围: 需要数学优化和边界处理

调试技巧:

1. 使用 pdb 进行调试
2. 打印中间状态变量
3. 使用 assert 进行条件验证

性能优化:

1. 避免不必要的字符串转换
2. 使用局部变量减少属性查找
3. 利用 Python 内置函数的高效实现

工程化实践:

1. 类型注解: 提高代码可读性
2. 异常处理: 确保程序健壮性
3. 单元测试: 保证代码质量
4. 文档字符串: 提供清晰的接口说明

"""

=====

文件: Code02_SuperPalindromesCPP1.java

=====

```
package class043;
```

```
// 超级回文数(superpalindromesInRange1 方法 C++版本)
// 如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。
// 现在，给定两个正整数 L 和 R （以字符串形式表示），
// 返回包含在范围 [L, R] 中的超级回文数的数目。
// 1 <= len(L) <= 18
// 1 <= len(R) <= 18
// L 和 R 是表示 [1, 10^18) 范围的整数的字符串
// 测试链接 : https://leetcode.cn/problems/super-palindromes/
// 如下实现是课上讲的 superpalindromesInRange1 方法的 C++版本，提交如下代码可以直接通过
```

```
//#include <iostream>
//#include <cmath>
//#include <limits>
//
//using namespace std;
//
//class Solution {
//public:
//    int superpalindromesInRange(string left, string right) {
//        long long l = stoll(left);
//        long long r = stoll(right);
//        long long limit = static_cast<long long>(sqrt(r));
//        long long seed = 1;
//        long long num = 0;
//        int ans = 0;
//        do {
//            num = evenEnlarge(seed);
//            if (num <= limit && safeSquare(num) && check(num * num, l, r)) {
//                ans++;
//            }
//            num = oddEnlarge(seed);
//            if (num <= limit && safeSquare(num) && check(num * num, l, r)) {
//                ans++;
//            }
//            seed++;
//        }
```

```

//        } while (num < limit);
//
//        return ans;
//    }
//
//private:
//    bool safeSquare(long long num) {
//        return num <= static_cast<long long>(sqrt(numeric_limits<long long>::max()));
//    }
//
//    long long evenEnlarge(long long seed) {
//        long long ans = seed;
//        while (seed != 0) {
//            ans = ans * 10 + seed % 10;
//            seed /= 10;
//        }
//        return ans;
//    }
//
//    long long oddEnlarge(long long seed) {
//        long long ans = seed;
//        seed /= 10;
//        while (seed != 0) {
//            ans = ans * 10 + seed % 10;
//            seed /= 10;
//        }
//        return ans;
//    }
//
//    bool check(long long ans, long long l, long long r) {
//        return ans >= l && ans <= r && isPalindrome(ans);
//    }
//
//    bool isPalindrome(long long num) {
//        long long offset = 1;
//        while (num / offset >= 10) {
//            offset *= 10;
//        }
//        while (num != 0) {
//            if (num / offset != num % 10) {
//                return false;
//            }
//            num = (num % offset) / 10;
//        }
//        return true;
//    }
}

```

```
//          offset /= 100;
//      }
//      return true;
//  }
//};
```

文件: Code02_SuperPalindromesCPP2.java

```
package class043;

// 超级回文数(superpalindromesInRange2 方法 C++版本)
// 如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。
// 现在，给定两个正整数 L 和 R（以字符串形式表示），
// 返回包含在范围 [L, R] 中的超级回文数的数目。
// 1 <= len(L) <= 18
// 1 <= len(R) <= 18
// L 和 R 是表示 [1, 10^18) 范围的整数的字符串
// 测试链接：https://leetcode.cn/problems/super-palindromes/
// 如下实现是课上讲的 superpalindromesInRange2 方法的 C++版本，提交如下代码可以直接通过
```

```
//#include <string>
//#include <vector>
//
//using namespace std;
//
//class Solution {
//public:
//    int superpalindromesInRange(string left, string right) {
//        long long l = stoll(left);
//        long long r = stoll(right);
//        int i = 0;
//        while (i < record.size() && record[i] < l) {
//            i++;
//        }
//        int j = record.size() - 1;
//        while (j >= 0 && record[j] > r) {
//            j--;
//        }
//        return (j >= i) ? (j - i + 1) : 0;
//    }
//}
```

```

//private:
//    const vector<long long> record = {
//        1L, 4L, 9L, 121L, 484L, 10201L, 12321L, 14641L, 40804L, 44944L,
//        1002001L, 1234321L, 4008004L, 100020001L, 102030201L, 104060401L,
//        121242121L, 123454321L, 125686521L, 400080004L, 404090404L,
//        10000200001L, 10221412201L, 12102420121L, 12345654321L,
//        40000800004L, 1000002000001L, 1002003002001L, 1004006004001L,
//        1020304030201L, 1022325232201L, 1024348434201L, 1210024200121L,
//        1212225222121L, 1214428244121L, 1232346432321L, 1234567654321L,
//        4000008000004L, 4004009004004L, 100000020000001L, 100220141022001L,
//        102012040210201L, 102234363432201L, 121000242000121L,
//        121242363242121L, 123212464212321L, 123456787654321L,
//        400000080000004L, 10000000200000001L, 10002000300020001L,
//        10004000600040001L, 10020210401202001L, 10022212521222001L,
//        10024214841242001L, 10201020402010201L, 10203040504030201L,
//        10205060806050201L, 10221432623412201L, 10223454745432201L,
//        12100002420000121L, 12102202520220121L, 12104402820440121L,
//        12122232623222121L, 12124434743442121L, 12321024642012321L,
//        12323244744232321L, 12343456865434321L, 12345678987654321L,
//        40000000800000004L, 40004000900040004L, 1000000002000000001L,
//        1000220014100220001L, 1002003004003002001L, 1002223236323222001L,
//        1020100204020010201L, 1020322416142230201L, 1022123226223212201L,
//        1022345658565432201L, 1210000024200000121L, 1210242036302420121L,
//        1212203226223022121L, 1212445458545442121L, 1232100246420012321L,
//        1232344458544432321L, 1234323468643234321L, 4000000008000000004L
//    };
//};

=====

文件: Code03_IsPalindrome.cpp
=====
```

```
#include <iostream>
```

```
#include <string>
```

```
#include <algorithm>
```

```
#include <climits>
```

```
#include <vector>
```

```
#include <cctype>
```

```
using namespace std;
```

```
/**
```

```
* 回文数判断问题 - C++版本
```

```
*  
* 问题描述:  
* 判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。  
*  
* 算法思路:  
* 方法 1：数学方法 – 通过位运算判断回文  
* 方法 2：字符串方法 – 转换为字符串后判断  
*  
* 时间复杂度分析:  
* - 数学方法:  $O(\log n)$ , 其中  $n$  为数字的位数  
* - 字符串方法:  $O(n)$ , 字符串转换和比较  
*  
* 空间复杂度分析:  
* - 数学方法:  $O(1)$ , 常数额外空间  
* - 字符串方法:  $O(n)$ , 需要存储字符串  
*  
* 工程化考量:  
* 1. 边界处理: 负数、0、边界值  
* 2. 溢出处理: 反转时可能出现的溢出问题  
* 3. 性能优化: 选择合适的算法策略  
* 4. 可测试性: 设计全面的测试用例  
*/
```

```
class PalindromeSolver {  
public:  
    /**  
     * 方法 1：数学方法  
     * 通过数学运算反转数字，避免字符串转换  
     */  
    bool isPalindromeMath(int x) {  
        // 边界情况处理  
        if (x < 0) return false; // 负数不是回文数  
        if (x < 10) return true; // 单个数字是回文数  
        if (x % 10 == 0) return false; // 以 0 结尾的数字不是回文数（除了 0 本身）  
  
        int original = x;  
        long reversed = 0; // 使用 long 防止溢出  
  
        while (x > 0) {  
            reversed = reversed * 10 + x % 10;  
            x /= 10;  
        }  
    }
```

```
        return original == reversed;
    }

/***
 * 方法 2：优化数学方法
 * 只反转一半数字，避免完全反转可能导致的溢出问题
 */
bool isPalindromeOptimized(int x) {
    // 边界情况处理
    if (x < 0) return false;
    if (x < 10) return true;
    if (x % 10 == 0) return false;

    int reversed = 0;

    // 当原始数字大于反转数字时继续
    while (x > reversed) {
        reversed = reversed * 10 + x % 10;
        x /= 10;
    }

    // 数字长度为奇数时，通过 reversed/10 去除中间位
    return x == reversed || x == reversed / 10;
}

/***
 * 方法 3：字符串方法
 * 转换为字符串后判断回文
 */
bool isPalindromeString(int x) {
    if (x < 0) return false;

    string s = to_string(x);
    int left = 0, right = s.length() - 1;

    while (left < right) {
        if (s[left] != s[right]) {
            return false;
        }
        left++;
        right--;
    }
}
```

```
    return true;
}

/***
 * 方法 4：递归方法
 * 使用递归判断回文数
 */
bool isPalindromeRecursive(int x) {
    if (x < 0) return false;
    if (x < 10) return true;

    // 获取数字的位数
    int digits = 0;
    int temp = x;
    while (temp > 0) {
        digits++;
        temp /= 10;
    }

    return isPalindromeRecursiveHelper(x, digits);
}

private:
    bool isPalindromeRecursiveHelper(int x, int digits) {
        if (digits <= 1) return true;

        // 获取最高位和最低位
        int power = 1;
        for (int i = 1; i < digits; i++) {
            power *= 10;
        }

        int firstDigit = x / power;
        int lastDigit = x % 10;

        if (firstDigit != lastDigit) {
            return false;
        }

        // 去掉最高位和最低位
        int remaining = (x % power) / 10;

        return isPalindromeRecursiveHelper(remaining, digits - 2);
    }
}
```

```

    }

};

/***
 * 补充训练题目 - C++实现
 */

/***
 * LeetCode 125. 验证回文串
 * 给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写
 */
bool isPalindromeString(string s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        // 跳过非字母数字字符
        while (left < right && !isalnum(s[left])) {
            left++;
        }
        while (left < right && !isalnum(s[right])) {
            right--;
        }

        // 比较字符（忽略大小写）
        if (tolower(s[left]) != tolower(s[right])) {
            return false;
        }

        left++;
        right--;
    }

    return true;
}

/***
 * 辅助函数：判断字符串指定范围是否是回文
 */
bool isPalindromeRange(string s, int left, int right) {
    while (left < right) {
        if (s[left] != s[right]) {
            return false;
        }
    }
}

```

```

        left++;
        right--;
    }
    return true;
}

/***
 * LeetCode 680. 验证回文字符串 II
 * 给定一个非空字符串 s，最多删除一个字符，判断是否能成为回文字符串
 */
bool validPalindrome(string s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        if (s[left] != s[right]) {
            // 尝试删除左边或右边的字符
            return isPalindromeRange(s, left + 1, right) ||
                   isPalindromeRange(s, left, right - 1);
        }
        left++;
        right--;
    }

    return true;
}

/***
 * 辅助函数：中心扩展法
 */
int expandAroundCenter(string s, int left, int right) {
    while (left >= 0 && right < s.length() && s[left] == s[right]) {
        left--;
        right++;
    }
    return right - left - 1;
}

/***
 * LeetCode 5. 最长回文子串
 * 使用中心扩展法找到最长回文子串
 */
string longestPalindrome(string s) {
    if (s.empty()) return "";

```

```

int start = 0, end = 0;

for (int i = 0; i < s.length(); i++) {
    // 奇数长度回文
    int len1 = expandAroundCenter(s, i, i);
    // 偶数长度回文
    int len2 = expandAroundCenter(s, i, i + 1);

    int len = (len1 > len2) ? len1 : len2;

    if (len > end - start) {
        start = i - (len - 1) / 2;
        end = i + len / 2;
    }
}

return s.substr(start, end - start + 1);
}

/***
 * 辅助函数：中心扩展法统计回文子串
 */
int expandAroundCenterCount(string s, int left, int right) {
    int count = 0;
    while (left >= 0 && right < s.length() && s[left] == s[right]) {
        count++;
        left--;
        right++;
    }
    return count;
}

/***
 * LeetCode 647. 回文子串
 * 统计字符串中的回文子串数目
 */
int countSubstrings(string s) {
    int count = 0;
    int n = s.length();

    for (int i = 0; i < n; i++) {
        // 奇数长度回文子串

```

```

        count += expandAroundCenterCount(s, i, i);
        // 偶数长度回文子串
        count += expandAroundCenterCount(s, i, i + 1);
    }

    return count;
}

/***
 * LeetCode 516. 最长回文子序列
 * 找出字符串中最长的回文子序列的长度
 */
int longestPalindromeSubseq(string s) {
    int n = s.length();
    // dp[i][j]表示s[i..j]的最长回文子序列长度
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // 单个字符的回文子序列长度为1
    for (int i = 0; i < n; i++) {
        dp[i][i] = 1;
    }

    // 从短到长填充 dp 数组
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j]) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = (dp[i + 1][j] > dp[i][j - 1]) ? dp[i + 1][j] : dp[i][j - 1];
            }
        }
    }

    return dp[0][n - 1];
}

/***
 * LeetCode 9. 回文数（简化版）
 * 不使用字符串转换的判断方法
 */
bool isPalindrome(int x) {
    if (x < 0) return false;

```

```

if (x < 10) return true;
if (x % 10 == 0) return false;

int reversed = 0;
while (x > reversed) {
    reversed = reversed * 10 + x % 10;
    x /= 10;
}

return x == reversed || x == reversed / 10;
}

// 测试函数
void testPalindrome() {
    PalindromeSolver solver;

    // 测试用例
    vector<int> testCases = {
        121, -121, 10, -101, 0, 1, 12321, 12345, 1001, 9999
    };

    cout << "回文数判断测试:" << endl;
    for (int num : testCases) {
        bool result1 = solver.isPalindromeMath(num);
        bool result2 = solver.isPalindromeOptimized(num);
        bool result3 = solver.isPalindromeString(num);
        bool result4 = solver.isPalindromeRecursive(num);

        cout << "数字: " << num << " -> ";
        cout << "数学方法: " << (result1 ? "是" : "否") << ", ";
        cout << "优化方法: " << (result2 ? "是" : "否") << ", ";
        cout << "字符串方法: " << (result3 ? "是" : "否") << ", ";
        cout << "递归方法: " << (result4 ? "是" : "否") << endl;
    }
    cout << endl;
}

// 测试补充题目
cout << "==== 补充训练题目测试 ===" << endl;

// 测试回文串验证
string testStr = "A man, a plan, a canal: Panama";
cout << "回文串验证: \" " << testStr << " \" -> " << (isPalindromeString(testStr) ? "是" : "否")
") << endl;

```

```

// 测试验证回文字符串 II
string testStr2 = "aba";
string testStr3 = "abca";
cout << "验证回文字符串 II: \"" << testStr2 << "\"" -> " " << (validPalindrome(testStr2) ? "是" :
"否") << endl;
cout << "验证回文字符串 II: \"" << testStr3 << "\"" -> " " << (validPalindrome(testStr3) ? "是" :
"否") << endl;

// 测试最长回文子串
string testStr4 = "babad";
cout << "最长回文子串: \"" << testStr4 << "\"" -> "\" " << longestPalindrome(testStr4) << "\""
<< endl;

// 测试回文子串统计
string testStr5 = "abc";
cout << "回文子串统计: \"" << testStr5 << "\"" -> " " << countSubstrings(testStr5) << endl;

// 测试最长回文子序列
string testStr6 = "bbbab";
cout << "最长回文子序列: \"" << testStr6 << "\"" -> " " << longestPalindromeSubseq(testStr6) <<
endl;
}

```

```

int main() {
    testPalindrome();
    return 0;
}

```

```

/**
 * 算法技巧总结 - C++版本
 *
 * 核心概念:
 * 1. 回文数判断技术:
 *     - 数学方法: 通过数字反转判断回文
 *     - 优化方法: 只反转一半数字, 避免溢出
 *     - 字符串方法: 转换为字符串后判断
 *     - 递归方法: 递归判断首尾字符
 *
 * 2. 算法选择策略:
 *     - 性能要求高: 使用优化数学方法
 *     - 代码简洁: 使用字符串方法
 *     - 教学演示: 使用递归方法

```

```
*  
* 3. 边界情况处理:  
*   - 负数处理: 负数不是回文数  
*   - 零处理: 0 是回文数  
*   - 溢出处理: 使用 long 类型或优化方法  
  
*  
* 调试技巧:  
* 1. 边界值测试: 测试各种边界情况  
* 2. 性能分析: 比较不同方法的执行时间  
* 3. 内存分析: 检查内存使用情况  
  
*  
* 工程化实践:  
* 1. 模块化设计: 分离不同算法实现  
* 2. 异常安全: 确保资源正确释放  
* 3. 代码可读性: 使用有意义的变量名和注释  
*/
```

=====

文件: Code03_IsPalindrome.java

=====

```
package class043;  
  
import java.util.*;  
  
/**  
 * 回文数判断  
 *  
 * 问题描述:  
 * 判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。  
 *  
 * 算法思路:  
 * 1. 负数不是回文数  
 * 2. 通过 offset 定位最高位  
 * 3. 比较最高位和最低位是否相等  
 * 4. 去掉最高位和最低位, 继续比较  
 * 5. 直到所有位都比较完毕  
 *  
 * 时间复杂度分析:  
 *  $O(\log n)$  - 其中 n 是输入数字的值, 需要遍历数字的一半位数  
 *  
 * 空间复杂度分析:  
 *  $O(1)$  - 只使用了常数级别的额外空间
```

*

* 工程化考量:

- * 1. 异常处理: 处理负数等边界情况
- * 2. 性能优化: 避免字符串转换, 直接通过数学运算处理
- * 3. 鲁棒性: 处理整数溢出问题

*

* 相关题目:

- * 1. LeetCode 9. 回文数 - <https://leetcode.cn/problems/palindrome-number/>
- * 2. LeetCode 125. 验证回文串 - <https://leetcode.cn/problems/valid-palindrome/>
- * 3. LeetCode 680. 验证回文字符串 II - <https://leetcode.cn/problems/valid-palindrome-ii/>
- * 4. LeetCode 409. 最长回文串 - <https://leetcode.cn/problems/longest-palindrome/>
- * 5. LeetCode 5. 最长回文子串 - <https://leetcode.cn/problems/longest-palindromic-substring/>
- * 6. LeetCode 234. 回文链表 - <https://leetcode.cn/problems/palindrome-linked-list/>
- * 7. LeetCode 266. 回文排列 - <https://leetcode.cn/problems/palindrome-permutation/>
- * 8. LeetCode 131. 分割回文串 - <https://leetcode.cn/problems/palindrome-partitioning/>
- * 9. LeetCode 336. 回文对 - <https://leetcode.cn/problems/palindrome-pairs/>
- * 10. LeetCode 516. 最长回文子序列 - <https://leetcode.cn/problems/longest-palindromic-subsequence/>
- * 11. LeetCode 479. 最大回文数乘积 - <https://leetcode.cn/problems/largest-palindrome-product/>
- * 12. 牛客网 - 回文数索引 - <https://www.nowcoder.com/practice/bcd40976533d45298591611b64c57bb0>
- * 13. 牛客网 - 回文数字判断 - <https://www.nowcoder.com/practice/35b8166c135448c5a5ba2cff8d430c32>
- * 14. 牛客网 - 最长回文子串 - <https://www.nowcoder.com/practice/b4525d1d84934cf280439aeecc36f4af>
- * 15. Codeforces 1438B - Valerii Against Everyone -
<https://codeforces.com/problemset/problem/1438/B>
- * 16. Codeforces 110A - Nearly Lucky Number - <https://codeforces.com/problemset/problem/110/A>
- * 17. Codeforces 1332B - 复合回文数 - <https://codeforces.com/problemset/problem/1332/B>
- * 18. AtCoder ABC162C - Sum of gcd of Tuples (Easy) -
https://atcoder.jp/contests/abc162/tasks/abc162_c
- * 19. AtCoder ABC155A - Poor - https://atcoder.jp/contests/abc155/tasks/abc155_a
- * 20. AtCoder ABC159E - Dividing Chocolate - https://atcoder.jp/contests/abc159/tasks/abc159_e
- * 21. 洛谷 P1012 - 拼数 - <https://www.luogu.com.cn/problem/P1012>
- * 22. 洛谷 P1157 - 组合的输出 - <https://www.luogu.com.cn/problem/P1157>
- * 23. 洛谷 P1217 - [USACO1.5]回文质数 Prime Palindromes - <https://www.luogu.com.cn/problem/P1217>
- * 24. 洛谷 P1048 - 采药 - <https://www.luogu.com.cn/problem/P1048>
- * 25. HackerRank - The Palindrome Index - <https://www.hackerrank.com/challenges/palindrome-index/problem>
- * 26. HackerRank - Making Anagrams - <https://www.hackerrank.com/challenges/making-anagrams/problem>
- * 27. HackerRank - Sherlock and Cost - <https://www.hackerrank.com/challenges/sherlock-and-cost/problem>
- * 28. HackerRank - The Longest Palindromic Subsequence -
<https://www.hackerrank.com/challenges/longest-palindromic-subsequence/problem>
- * 29. UVa 10945 - Mother Bear -

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1886

* 30. UVa 10189 - Minesweeper -

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=10189

* 31. POJ 3974 - Palindrome - <https://poj.org/problem?id=3974>

* 32. POJ 1163 - The Triangle - <https://poj.org/problem?id=1163>

* 33. HDU 1321 - Word Amalgamation - <https://acm.hdu.edu.cn/showproblem.php?pid=1321>

* 34. HDU 1203 - I NEED A OFFER! - <https://acm.hdu.edu.cn/showproblem.php?pid=1203>

* 35. LintCode 1178. 有效的回文 - <https://www.lintcode.com/problem/1178/>

* 36. LintCode 125. 背包问题 II - <https://www.lintcode.com/problem/125/>

* 37. LintCode 200. 最长回文子串 - <https://www.lintcode.com/problem/200/>

* 38. LintCode 130. 堆化 - <https://www.lintcode.com/problem/130/>

* 39. LintCode 135. 数字组合 - <https://www.lintcode.com/problem/135/>

* 40. AcWing 901. 滑雪 - <https://www.acwing.com/problem/content/903/>

* 41. AcWing 1482. 进制 - <https://www.acwing.com/problem/content/1484/>

* 42. LeetCode 334. 递增的三元子序列 - <https://leetcode.cn/problems/increasing-triplet-subsequence/>

* 43. LeetCode 344. 反转字符串 - <https://leetcode.cn/problems/reverse-string/>

* 44. LeetCode 459. 重复的子字符串 - <https://leetcode.cn/problems/repeated-substring-pattern/>

* 45. LeetCode 680. 验证回文字符串 II - <https://leetcode.cn/problems/valid-palindrome-ii/>

* 46. LeetCode 1281. 整数的各位积和之差 - <https://leetcode.cn/problems/subtract-the-product-and-sum-of-digits-of-an-integer/>

* 47. 牛客网 - 最大数 - <https://www.nowcoder.com/practice/fc897457408f4bbe9d3f87588f497729>

* 48. 牛客网 - 括号生成 - <https://www.nowcoder.com/practice/c9addb265cdf4cdd92c092c655d164ca>

* 49. Codeforces 1481A - Space Navigation - <https://codeforces.com/problemset/problem/1481/A>

* 50. Codeforces 1352B - Same Parity Summands - <https://codeforces.com/problemset/problem/1352/B>

*/

public class Code03_IsPalindrome {

/**

* 判断一个整数是否是回文数

*

* @param num 待判断的整数

* @return 如果是回文数返回 true, 否则返回 false

*

* 算法思路:

* 1. 负数不是回文数, 直接返回 false

* 2. 通过 offset 定位最高位数字

* 3. 比较最高位和最低位数字是否相等

* 4. 去掉最高位和最低位, 继续比较

* 5. 直到所有位都比较完毕

*

* 示例：

* isPalindrome(121) -> true (121 从左到右和从右到左都是 121)

* isPalindrome(-121) -> false (负数不是回文数)

* isPalindrome(10) -> false (从右到左是 01, 不是回文数)

*

* 工程化考量：

* 1. 算法选择：不使用字符串转换，直接通过数学运算判断，避免额外的内存分配

* 2. 边界处理：特别处理 0 和负数

* 3. 性能优化：通过除以 offset 定位最高位，避免使用 Math.pow 可能导致的浮点数精度问题

*/

```
public static boolean isPalindrome(int num) {  
    // 负数不是回文数  
    if (num < 0) {  
        return false;  
    }  
    int offset = 1;  
    // 注意这么写是为了防止溢出  
    // 通过不断乘以 10 来定位最高位的位置  
    while (num / offset >= 10) {  
        offset *= 10;  
    }  
    // 首尾判断  
    // 比较最高位数字(num / offset) 和最低位数字(num % 10)  
    // 然后去掉最高位和最低位继续比较  
    while (num != 0) {  
        // 如果最高位和最低位不相等，则不是回文数  
        if (num / offset != num % 10) {  
            return false;  
        }  
        // 去掉最高位和最低位  
        // (num % offset) 去掉最高位  
        // / 10 去掉最低位  
        num = (num % offset) / 10;  
        // offset 减少两位（因为去掉了两个数字）  
        offset /= 100;  
    }  
    return true;  
}
```

/**

* 补充训练题目

*/

```

/**
 * 补充题目 1: LeetCode 131. 分割回文串
 * 题目描述: 给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分
割方案。
 * 链接: https://leetcode.cn/problems/palindrome-partitioning/
 *
 * 解题思路:
 * - 使用回溯算法，枚举所有可能的分割点
 * - 对于每个分割点，判断当前子串是否为回文串
 * - 如果是回文串，将其加入当前路径，并继续递归处理剩余部分
 * - 时间复杂度: O(n*2^n)，空间复杂度: O(n)
 */
public List<List<String>> partition(String s) {
    List<List<String>> result = new ArrayList<>();
    backtrack(s, 0, new ArrayList<>(), result);
    return result;
}

private void backtrack(String s, int start, List<String> path, List<List<String>> result) {
    // 基本情况: 已经处理整个字符串
    if (start == s.length()) {
        result.add(new ArrayList<>(path));
        return;
    }

    // 枚举所有可能的结束位置
    for (int i = start; i < s.length(); i++) {
        // 检查子串是否为回文串
        if (isPalindromeStrHelper(s, start, i)) {
            // 如果是回文串，加入路径
            path.add(s.substring(start, i + 1));
            // 递归处理剩余部分
            backtrack(s, i + 1, path, result);
            // 回溯，移除最后添加的子串
            path.remove(path.size() - 1);
        }
    }
}

private boolean isPalindromeStrHelper(String s, int left, int right) {
    // 双指针法判断子串是否为回文串
    while (left < right) {
        if (s.charAt(left++) != s.charAt(right--)) {

```

```

        return false;
    }
}

return true;
}

/***
 * 补充题目 2: LeetCode 234. 回文链表
 * 题目描述: 请判断一个链表是否为回文链表。
 * 链接: https://leetcode.cn/problems/palindrome-linked-list/
 *
 * 解题思路:
 * - 找到链表的中点
 * - 反转链表的后半部分
 * - 比较前半部分和反转后的后半部分
 * - 恢复链表结构（可选）
 * - 时间复杂度: O(n), 空间复杂度: O(1)
 */
public class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

public boolean isPalindrome(ListNode head) {
    // 处理边界情况
    if (head == null || head.next == null) {
        return true;
    }

    // 找到链表的中点
    ListNode slow = head, fast = head;
    while (fast.next != null && fast.next.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // 反转后半部分链表
    ListNode secondHalfStart = reverseList(slow.next);

    // 比较前半部分和反转后的后半部分
    while (secondHalfStart != null) {
        if (head.val != secondHalfStart.val) {
            return false;
        }
        head = head.next;
        secondHalfStart = secondHalfStart.next;
    }
    return true;
}

// 反转链表
private ListNode reverseList(ListNode head) {
    if (head == null || head.next == null) {
        return head;
    }

    ListNode prev = null;
    ListNode curr = head;
    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}

```

```

ListNode p1 = head, p2 = secondHalfStart;
boolean isPalindrome = true;

while (p2 != null) {
    if (p1.val != p2.val) {
        isPalindrome = false;
        break;
    }
    p1 = p1.next;
    p2 = p2.next;
}

// 恢复链表结构（可选但好的做法）
slow.next = reverseList(secondHalfStart);

return isPalindrome;
}

private ListNode reverseList(ListNode head) {
    ListNode prev = null, curr = head;
    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}

/***
 * 补充题目 3: LeetCode 5. 最长回文子串（辅助方法版本）
 * 题目描述: 给你一个字符串 s，找到 s 中最长的回文子串。
 * 链接: https://leetcode.cn/problems/longest-palindromic-substring/
 *
 * 解题思路:
 * - 使用中心扩展法，枚举每个可能的回文中心
 * - 考虑奇数长度和偶数长度两种情况
 * - 对于每个中心，向两边扩展，直到不能形成回文为止
 * - 时间复杂度: O(n^2)，空间复杂度: O(1)
 */
public String findLongestPalindrome(String s) {
    if (s == null || s.length() < 1) {
        return "";
}

```

```
}
```

```
int start = 0, end = 0;

// 枚举每个可能的回文中心
for (int i = 0; i < s.length(); i++) {
    // 奇数长度的回文串，中心是单个字符
    int len1 = expandAroundCenter(s, i, i);
    // 偶数长度的回文串，中心是两个字符之间
    int len2 = expandAroundCenter(s, i, i + 1);

    // 取两种情况中的最大长度
    int maxLen = Math.max(len1, len2);

    // 如果找到更长的回文串，更新起始和结束位置
    if (maxLen > end - start + 1) {
        start = i - (maxLen - 1) / 2;
        end = i + maxLen / 2;
    }
}

return s.substring(start, end + 1);
}
```

```
private int expandAroundCenter(String s, int left, int right) {
    // 向两边扩展，直到不能形成回文为止
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
        left--;
        right++;
    }
    // 返回回文串的长度
    return right - left - 1;
}
```

```
/**
```

```
* 补充题目 4: LeetCode 516. 最长回文子序列
* 题目描述: 给你一个字符串 s，找到其中最长的回文子序列，并返回该序列的长度。
* 链接: https://leetcode.cn/problems/longest-palindromic-subsequence/
*
* 解题思路:
* - 使用动态规划，dp[i][j]表示子串 s[i...j] 的最长回文子序列长度
* - 状态转移方程: 如果 s[i] == s[j]，则 dp[i][j] = dp[i+1][j-1] + 2；否则 dp[i][j] =
max(dp[i+1][j], dp[i][j-1])
```

```

* - 时间复杂度: O(n^2), 空间复杂度: O(n^2)
*/
public int longestPalindromeSubseq(String s) {
    int n = s.length();
    // dp[i][j] 表示子串 s[i...j] 的最长回文子序列长度
    int[][] dp = new int[n][n];

    // 初始化: 单个字符的最长回文子序列长度为 1
    for (int i = 0; i < n; i++) {
        dp[i][i] = 1;
    }

    // 从长度为 2 的子串开始, 逐渐扩展到整个字符串
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            // 如果首尾字符相等, 可以同时包含在回文子序列中
            if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                // 否则取两种情况的最大值
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }

    // 整个字符串的最长回文子序列长度
    return dp[0][n - 1];
}

/**
 * 补充题目 5: LeetCode 336. 回文对
 * 题目描述: 给定一组唯一的单词, 找出所有不同的索引对 (i, j), 使得单词 words[i] + words[j] 形成回文串。
 * 链接: https://leetcode.cn/problems/palindrome-pairs/
 *
 * 解题思路:
 * - 使用哈希表存储单词及其索引
 * - 对于每个单词, 枚举所有可能的分割点, 检查前缀或后缀是否可以与其他单词形成回文
 * - 利用回文性质进行优化
 * - 时间复杂度: O(n*k^2), 其中 n 是单词数量, k 是单词的最大长度
 */
public List<List<Integer>> palindromePairs(String[] words) {

```

```

List<List<Integer>> result = new ArrayList<>();
if (words == null || words.length < 2) {
    return result;
}

// 构建哈希表，存储单词及其索引
Map<String, Integer> wordMap = new HashMap<>();
for (int i = 0; i < words.length; i++) {
    wordMap.put(words[i], i);
}

// 处理每个单词
for (int i = 0; i < words.length; i++) {
    String word = words[i];

    // 情况 1：空字符串与回文单词配对
    if (word.isEmpty()) {
        for (int j = 0; j < words.length; j++) {
            if (i != j && isPalindromeStrHelper(words[j], 0, words[j].length() - 1)) {
                result.add(Arrays.asList(i, j));
                result.add(Arrays.asList(j, i));
            }
        }
        continue;
    }

    // 情况 2：尝试所有可能的分割
    for (int j = 0; j <= word.length(); j++) {
        String prefix = word.substring(0, j);
        String suffix = word.substring(j);

        // 检查前缀的反转是否存在
        String reversedPrefix = new StringBuilder(prefix).reverse().toString();
        if (isPalindromeStrHelper(suffix, 0, suffix.length() - 1) &&
wordMap.containsKey(reversedPrefix) && wordMap.get(reversedPrefix) != i) {
            result.add(Arrays.asList(i, wordMap.get(reversedPrefix)));
        }
    }

    // 避免重复添加（当 j=0 时，reversedSuffix 和 word 相同）
    if (j > 0) {
        // 检查后缀的反转是否存在
        String reversedSuffix = new StringBuilder(suffix).reverse().toString();
        if (isPalindromeStrHelper(prefix, 0, prefix.length() - 1) &&

```

```

wordMap.containsKey(reversedSuffix) && wordMap.get(reversedSuffix) != i) {
    result.add(Arrays.asList(wordMap.get(reversedSuffix), i));
}
}

}

return result;
}

/***
 * 补充题目 6: LeetCode 680. 验证回文字符串 II
 * 题目描述: 给定一个非空字符串 s, 最多删除一个字符后, 判断能否成为回文字符串。
 * 链接: https://leetcode.cn/problems/valid-palindrome-ii/
 *
 * 解题思路:
 * - 使用双指针法判断是否为回文串
 * - 当遇到不匹配的字符时, 尝试删除左指针指向的字符或右指针指向的字符
 * - 递归检查剩余部分是否为回文串
 * - 时间复杂度: O(n), 空间复杂度: O(n) (递归调用栈)
 */
public boolean validPalindrome(String s) {
    return isPalindromeWithDelete(s, 0, s.length() - 1, false);
}

private boolean isPalindromeWithDelete(String s, int left, int right, boolean hasDeleted) {
    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            // 如果已经删除过字符, 则不能再删除
            if (hasDeleted) {
                return false;
            }
            // 尝试删除左字符或右字符
            return isPalindromeWithDelete(s, left + 1, right, true) ||
                   isPalindromeWithDelete(s, left, right - 1, true);
        }
        left++;
        right--;
    }
    return true;
}

/***

```

* 补充题目 7: LeetCode 409. 最长回文串

* 题目描述: 给定一个包含大写字母和小写字母的字符串 s , 返回通过这些字母构造成的最长的回文串的长度。

* 链接: <https://leetcode.cn/problems/longest-palindrome/>

*

* 解题思路:

* - 统计每个字符出现的次数

* - 偶数次的字符可以全部用于构建回文串

* - 奇数次的字符最多可以有一个字符保留所有出现次数 (放在回文串中心), 其他需要减 1 后取偶数部分

* - 时间复杂度: O(n) , 空间复杂度: O(1) (因为字符集大小固定)

*/

```
public int longestPalindrome(String s) {  
    int[] charCount = new int[128]; // 假设使用 ASCII 字符
```

// 统计每个字符出现的次数

```
for (char c : s.toCharArray()) {  
    charCount[c]++;  
}
```

```
int result = 0;
```

```
boolean hasOdd = false;
```

// 计算可以构成回文串的最大长度

```
for (int count : charCount) {  
    if (count % 2 == 0) {  
        // 偶数次的字符全部使用  
        result += count;  
    } else {  
        // 奇数次的字符, 使用最大的偶数部分  
        result += count - 1;  
        hasOdd = true;  
    }  
}
```

// 如果有奇数次的字符, 可以在回文串中心添加一个字符

```
if (hasOdd) {  
    result++;  
}
```

```
return result;
```

```
}
```

```

/**
 * 补充题目 8: LeetCode 266. 回文排列
 * 题目描述: 给定一个字符串, 判断该字符串中是否可以通过重新排列组合, 形成一个回文字符串。
 * 链接: https://leetcode.cn/problems/palindrome-permutation/
 *
 * 解题思路:
 * - 使用位掩码记录字符出现次数的奇偶性
 * - 遍历字符串, 对于每个字符, 翻转其对应的位
 * - 最终, 如果位掩码为 0 (所有字符出现次数均为偶数) 或只有一个位为 1 (有一个字符出现次数为奇数), 则可以构成回文串
 * - 时间复杂度: O(n), 空间复杂度: O(1)
 */
public boolean canPermutePalindrome(String s) {
    int bitMask = 0;

    for (char c : s.toCharArray()) {
        // 翻转对应字符的位
        bitMask ^= 1 << (c - 'a');
    }

    // 检查是否最多只有一个位为 1
    return bitMask == 0 || (bitMask & (bitMask - 1)) == 0;
}

/**
 * 回文算法技术与应用深度剖析
 *
 * 核心概念与理论基础:
 * 1. 回文数学理论:
 *   - 定义扩展: 经典回文、镜像回文、数字回文等概念体系
 *   - 回文数分布: 在自然数中的稀疏性与生成规律
 *   - 数学性质: 回文数与素数、平方数的特殊关系
 *   - 回文猜想: 关于回文数生成的数学猜想 (如利克瑞尔数猜想)
 *
 * 2. 计算模型:
 *   - 确定性有限自动机与回文识别
 *   - 上下文无关文法描述回文结构
 *   - 形式语言理论中的回文类
 *   - 回文等价类与商空间分析
 *
 * 3. 复杂度理论视角:
 *   - 不同回文问题的计算复杂度分类
 *   - P 与 NP 问题在回文领域的体现

```

- * - 近似算法与启发式方法的边界
- * - 参数化复杂度分析
- *
- * 高级算法技术:
 - * 1. Manacher 算法深度解析:
 - * - 核心思想: 利用已知回文信息避免重复计算
 - * - 算法流程: 中心扩展与边界维护
 - * - 复杂度证明: $O(n)$ 时间复杂度的理论依据
 - * - 实现技巧: 预处理与特殊字符插入
 - *
 - * 2. 回文自动机 (Palindromic Tree):
 - * - 数据结构设计: 节点定义与边结构
 - * - 构建算法: 增量添加字符的过程
 - * - 应用场景: 统计不同回文子串、最长回文子串等
 - * - 性能分析: 时间与空间复杂度保证
 - *
 - * 3. 高级哈希技术:
 - * - 前缀哈希与后缀哈希结合
 - * - 多项式滚动哈希的应用
 - * - 双向哈希验证策略
 - * - 哈希冲突的处理与防范
 - *
 - * 4. 动态规划优化技术:
 - * - 状态压缩: 从 $O(n^2)$ 到 $O(n)$ 空间优化
 - * - 并行计算: 对角线并行处理
 - * - 区间 DP 的高级应用
 - * - 决策单调性优化
 - *
 - * 多维度优化策略:
 - * 1. 算法层面优化:
 - * - 预计算与缓存: 空间换时间的经典策略
 - * - 剪枝技术: 基于启发式规则的搜索空间缩减
 - * - 问题转换: 等价问题的求解思路转换
 - * - 近似算法: 处理 NP 难问题的次优解
 - *
 - * 2. 实现层面优化:
 - * - 位运算加速: 字符频率统计与奇偶判断
 - * - 内存布局优化: 数据局部性原理应用
 - * - 指令级并行: SIMD 指令集的利用
 - * - 分支预测友好: 避免条件分支的优化技巧
 - *
 - * 3. 系统层面优化:
 - * - 多线程并行: 工作窃取算法的应用

- * - GPU 加速: 大规模并行计算
- * - 分布式处理: MapReduce 模式应用
- * - 内存层次结构优化: 缓存友好的数据结构
- *
- * 工程化实践指南:
 - * 1. 算法选择矩阵:
 - * - 问题特性与算法匹配表
 - * - 时间-空间权衡决策框架
 - * - 实现复杂度考量因素
 - * - 可维护性与性能平衡
 - *
 - * 2. 测试与验证体系:
 - * - 全面测试用例设计方法
 - * - 边界条件覆盖策略
 - * - 性能基准测试框架
 - * - 随机测试与压力测试
 - *
 - * 3. 代码质量保证:
 - * - 模块化设计模式
 - * - 接口抽象原则
 - * - 文档化实践
 - * - 单元测试覆盖策略
 - *
- * 跨学科应用:
 - * 1. 生物信息学:
 - * - DNA 序列中的回文结构识别
 - * - 基因调控区域分析
 - * - 蛋白质结构预测中的回文模式
 - *
 - * 2. 密码学应用:
 - * - 基于回文的哈希函数
 - * - 对称加密算法中的应用
 - * - 消息完整性验证
 - * - 数字签名中的回文特性利用
 - *
 - * 3. 自然语言处理:
 - * - 回文在诗歌与修辞中的应用
 - * - 语言模式识别
 - * - 文本生成中的回文构造
 - * - 语言演化研究
 - *
 - * 4. 量子计算视角:
 - * - 量子算法中的回文处理

- * - 量子并行性在回文搜索中的优势
- * - 量子复杂度与经典算法的对比
- *
- * 前沿研究方向:
 - * 1. 理论研究:
 - * - 回文数在大数域中的分布性质
 - * - 回文猜想的计算验证
 - * - 参数化回文问题的复杂度
 - * - 随机字符串中的回文统计
 - *
 - * 2. 算法创新:
 - * - 面向流数据的在线回文算法
 - * - 大规模分布式回文搜索
 - * - 量子回文算法
 - * - 近似算法的性能边界研究
 - *
 - * 3. 应用拓展:
 - * - 区块链中的回文应用
 - * - AI 模型中的回文模式学习
 - * - 量子通信中的回文编码
 - * - 分布式系统中的回文验证机制
 - *
- * 总结:
 - * 回文算法作为计算机科学中的经典问题，其应用范围已远超传统的字符串处理领域。
 - * 从数学理论到工程实践，从经典算法到前沿研究，回文问题体现了算法设计的深刻思想和广泛应用。
 - * 掌握回文算法的设计与优化技巧，不仅有助于解决具体的算法问题，更能培养抽象思维和问题转化能力。
 - * 在未来的计算机科学发展，回文算法将继续在各个领域发挥重要作用，特别是在大数据处理、人工智能和量子计算等新兴领域，为解决复杂问题提供创新思路。
- */

```
public static void main(String[] args) {  
    // 测试回文数判断  
    int[] testCases = {121, -121, 10, -101, 0, 1, 12321, 12345, 1001, 9999};  
  
    System.out.println("回文数判断测试:");  
    for (int num : testCases) {  
        boolean result = isPalindrome(num);  
        System.out.println("数字: " + num + " -> " + (result ? "是" : "否"));  
    }  
}
```

文件: Code03_IsPalindrome.py

=====

"""

回文数判断问题 - Python 版本

问题描述:

判断一个整数是否是回文数。回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

算法思路:

方法 1: 数学方法 - 通过位运算判断回文

方法 2: 字符串方法 - 转换为字符串后判断

方法 3: 优化数学方法 - 只反转一半数字

方法 4: 递归方法 - 使用递归判断回文

时间复杂度分析:

- 数学方法: $O(\log n)$, 其中 n 为数字的位数
- 字符串方法: $O(\log n)$, 字符串转换和比较
- 优化数学方法: $O(\log n)$, 只反转一半数字
- 递归方法: $O(\log n)$, 递归深度为数位数的一半

空间复杂度分析:

- 数学方法: $O(1)$, 常数额外空间
- 字符串方法: $O(\log n)$, 需要存储字符串
- 优化数学方法: $O(1)$, 常数额外空间
- 递归方法: $O(\log n)$, 递归调用栈

工程化考量:

1. 边界处理: 负数、0、边界值
2. 溢出处理: Python 内置大整数支持, 无需担心溢出
3. 性能优化: 选择合适的算法策略
4. 可测试性: 设计全面的测试用例

"""

```
class PalindromeSolver:
```

```
    def __init__(self):
```

```
        pass
```

```
    def is_palindrome_math(self, x: int) -> bool:
```

```
        """
```

方法 1: 数学方法

通过数学运算反转数字, 避免字符串转换

```
        """
```

```
# 边界情况处理
if x < 0:
    return False # 负数不是回文数
if x < 10:
    return True # 单个数字是回文数
if x % 10 == 0:
    return False # 以 0 结尾的数字不是回文数（除了 0 本身）

original = x
reversed_num = 0

while x > 0:
    reversed_num = reversed_num * 10 + x % 10
    x //= 10

return original == reversed_num
```

```
def is_palindrome_optimized(self, x: int) -> bool:
```

```
"""

```

```
方法 2：优化数学方法
只反转一半数字，避免完全反转
"""

```

```
# 边界情况处理
```

```
if x < 0:
    return False
if x < 10:
    return True
if x % 10 == 0:
    return False
```

```
reversed_num = 0
```

```
# 当原始数字大于反转数字时继续
```

```
while x > reversed_num:
    reversed_num = reversed_num * 10 + x % 10
    x //= 10
```

```
# 数字长度为奇数时，通过 reversed_num//10 去除中间位
return x == reversed_num or x == reversed_num // 10
```

```
def is_palindrome_string(self, x: int) -> bool:
```

```
"""

```

```
方法 3：字符串方法

```

```
转换为字符串后判断回文
"""

if x < 0:
    return False

s = str(x)
return s == s[::-1] # 使用切片反转字符串

def is_palindrome_recursive(self, x: int) -> bool:
    """
方法 4: 递归方法
使用递归判断回文数
"""

    if x < 0:
        return False
    if x < 10:
        return True

    # 转换为字符串进行递归判断
    s = str(x)
    return self._is_palindrome_recursive_helper(s, 0, len(s) - 1)

def _is_palindrome_recursive_helper(self, s: str, left: int, right: int) -> bool:
    """
递归辅助函数

# 基础情况
if left >= right:
    return True

# 首尾字符不相等
if s[left] != s[right]:
    return False

# 递归判断剩余部分
return self._is_palindrome_recursive_helper(s, left + 1, right - 1)

def is_palindrome_generator(self, x: int) -> bool:
    """
方法 5: 生成器方法
使用生成器逐个比较数字的各位
"""

    if x < 0:
```

```
    return False
if x < 10:
    return True

# 获取数字的各位数字
digits = []
temp = x
while temp > 0:
    digits.append(temp % 10)
    temp //= 10

# 比较对称位置的数字
left, right = 0, len(digits) - 1
while left < right:
    if digits[left] != digits[right]:
        return False
    left += 1
    right -= 1

return True
```

补充训练题目 - Python 实现

```
def is_palindrome_string(s: str) -> bool:
    """
    LeetCode 125. 验证回文串
    给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写
    """
    left, right = 0, len(s) - 1

    while left < right:
        # 跳过非字母数字字符
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1

        # 比较字符（忽略大小写）
        if s[left].lower() != s[right].lower():
            return False

        left += 1
        right -= 1
```

```

return True

def valid_palindrome(s: str) -> bool:
    """
    LeetCode 680. 验证回文字符串 II
    给定一个非空字符串 s，最多删除一个字符，判断是否能成为回文字符串
    """
    def is_palindrome_range(left: int, right: int) -> bool:
        while left < right:
            if s[left] != s[right]:
                return False
            left += 1
            right -= 1
        return True

    left, right = 0, len(s) - 1

    while left < right:
        if s[left] != s[right]:
            # 尝试删除左边或右边的字符
            return is_palindrome_range(left + 1, right) or is_palindrome_range(left, right - 1)
        left += 1
        right -= 1

    return True

def longest_palindrome(s: str) -> str:
    """
    LeetCode 5. 最长回文子串
    使用中心扩展法找到最长回文子串
    """
    if not s:
        return ""

    def expand_around_center(left: int, right: int) -> int:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return right - left - 1

    start, end = 0, 0

```

```

for i in range(len(s)):
    # 奇数长度回文
    len1 = expand_around_center(i, i)
    # 偶数长度回文
    len2 = expand_around_center(i, i + 1)

    max_len = max(len1, len2)

    if max_len > end - start:
        start = i - (max_len - 1) // 2
        end = i + max_len // 2

return s[start:end + 1]

def count_substrings(s: str) -> int:
    """
    LeetCode 647. 回文子串
    统计字符串中的回文子串数目
    """
    def expand_around_center(left: int, right: int) -> int:
        count = 0
        while left >= 0 and right < len(s) and s[left] == s[right]:
            count += 1
            left -= 1
            right += 1
        return count

        total = 0
        for i in range(len(s)):
            # 奇数长度回文子串
            total += expand_around_center(i, i)
            # 偶数长度回文子串
            total += expand_around_center(i, i + 1)

        return total

def longest_palindrome_subseq(s: str) -> int:
    """
    LeetCode 516. 最长回文子序列
    找出字符串中最长的回文子序列的长度
    """
    n = len(s)
    # dp[i][j]表示 s[i:j+1] 的最长回文子序列长度

```

```

dp = [[0] * n for _ in range(n)]

# 单个字符的回文子序列长度为 1
for i in range(n):
    dp[i][i] = 1

# 从短到长填充 dp 数组
for length in range(2, n + 1):
    for i in range(n - length + 1):
        j = i + length - 1
        if s[i] == s[j]:
            dp[i][j] = dp[i + 1][j - 1] + 2
        else:
            dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

return dp[0][n - 1]

def is_palindrome_linked_list(head) -> bool:
    """
    LeetCode 234. 回文链表
    判断一个链表是否为回文链表
    """
    # 找到链表中点
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # 反转后半部分链表
    prev = None
    while slow:
        next_node = slow.next
        slow.next = prev
        prev = slow
        slow = next_node

    # 比较前后两部分
    left, right = head, prev
    while right:
        if left.val != right.val:
            return False
        left = left.next
        right = right.next

```

```
return True

# 链表节点定义（用于测试回文链表）
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def test_palindrome():
    """测试函数"""
    solver = PalindromeSolver()

    # 测试用例
    test_cases = [121, -121, 10, -101, 0, 1, 12321, 12345, 1001, 9999, 1234321]

    print("回文数判断测试:")
    for num in test_cases:
        result1 = solver.is_palindrome_math(num)
        result2 = solver.is_palindrome_optimized(num)
        result3 = solver.is_palindrome_string(num)
        result4 = solver.is_palindrome_recursive(num)
        result5 = solver.is_palindrome_generator(num)

        print(f"数字: {num:8d} -> ", end="")
        print(f"数学: {'是' if result1 else '否'}, ", end="")
        print(f"优化: {'是' if result2 else '否'}, ", end="")
        print(f"字符串: {'是' if result3 else '否'}, ", end="")
        print(f"递归: {'是' if result4 else '否'}, ", end="")
        print(f"生成器: {'是' if result5 else '否'}")

    print()

    # 测试补充题目
    print("== 补充训练题目测试 ==")

    # 测试回文串验证
    test_str = "A man, a plan, a canal: Panama"
    print(f"回文串验证: '{test_str}' -> {'是' if is_palindrome_string(test_str) else '否'}")

    # 测试验证回文字符串 II
    test_str2 = "aba"
    test_str3 = "abca"
    print(f"验证回文字符串 II: '{test_str2}' -> {'是' if valid_palindrome(test_str2) else '否'}")
```

```

print(f"验证回文字符串 II: '{test_str3}' -> {'是' if valid_palindrome(test_str3) else '否'}")

# 测试最长回文子串
test_str4 = "babad"
print(f"最长回文子串: '{test_str4}' -> {longest_palindrome(test_str4)}")

# 测试回文子串统计
test_str5 = "abc"
print(f"回文子串统计: '{test_str5}' -> {count_substrings(test_str5)}")

# 测试最长回文子序列
test_str6 = "bbbab"
print(f"最长回文子序列: '{test_str6}' -> {longest_palindrome_subseq(test_str6)}")

# 测试回文链表（简单演示）
# 创建链表: 1->2->2->1
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(2)
head.next.next.next = ListNode(1)
# 注意：这里需要实际实现链表判断逻辑，此处仅为演示

if __name__ == "__main__":
    test_palindrome()

"""

```

算法技巧总结 – Python 版本

核心概念：

1. 回文数判断技术：

- 数学方法：通过数字反转判断回文
- 优化方法：只反转一半数字，避免完全反转
- 字符串方法：利用 Python 字符串操作的便利性
- 递归方法：递归判断首尾字符
- 生成器方法：逐个比较数字的各位

2. Python 特有优势：

- 内置大整数支持：无需担心数值溢出
- 字符串切片：[::-1] 快速反转字符串
- 列表推导式：简化数字分解操作

3. 算法选择策略：

- 性能要求高：使用优化数学方法

- 代码简洁：使用字符串方法
- 教学演示：使用递归或生成器方法

调试技巧：

1. 使用 pdb 进行调试
2. 打印中间状态变量
3. 使用 assert 进行条件验证

性能优化：

1. 避免不必要的字符串转换
2. 使用局部变量减少属性查找
3. 利用 Python 内置函数的高效实现

工程化实践：

1. 类型注解：提高代码可读性
2. 异常处理：确保程序健壮性
3. 单元测试：保证代码质量
4. 文档字符串：提供清晰的接口说明

边界情况处理：

1. 负数：负数不是回文数
2. 零：0 是回文数
3. 个位数：所有个位数都是回文数
4. 以 0 结尾的数：除了 0 本身，其他以 0 结尾的数不是回文数

"""

文件：Code04_PalindromePartitioning.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>

using namespace std;

// 函数声明
void backtrackIP(string& s, int start, vector<string>& current, vector<string>& result);
void backtrackWordBreak(string& s, int start, vector<string>& wordDict,
                      string& current, vector<string>& result);
```

```
/**  
 * 分割回文串问题 - C++版本  
 *  
 * 问题描述:  
 * 给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。  
 *  
 * 算法思路:  
 * 方法 1: 回溯算法 - 枚举所有可能的分割方案  
 * 方法 2: 动态规划预处理 - 优化回文串判断  
 *  
 * 时间复杂度分析:  
 * - 回溯算法: O(n * 2^n)，其中 n 为字符串长度  
 * - 动态规划预处理: O(n^2)，预处理回文信息  
 *  
 * 空间复杂度分析:  
 * - 回溯算法: O(n)，递归栈深度  
 * - 动态规划预处理: O(n^2)，存储回文信息  
 *  
 * 工程化考量:  
 * 1. 剪枝优化: 提前终止无效分支  
 * 2. 记忆化搜索: 缓存中间结果  
 * 3. 边界处理: 空字符串、单字符等情况  
 * 4. 可测试性: 设计全面的测试用例  
 */
```

```
class PalindromePartitioningSolver {  
public:  
    /**  
     * 方法 1: 回溯算法  
     * 枚举所有可能的分割方案  
     */  
    vector<vector<string>> partition(string s) {  
        vector<vector<string>> result;  
        vector<string> current;  
        backtrack(s, 0, current, result);  
        return result;  
    }  
  
    /**  
     * 方法 2: 回溯算法 + 动态规划预处理  
     * 使用动态规划预处理回文信息，优化判断效率  
     */  
    vector<vector<string>> partitionWithDP(string s) {
```

```

int n = s.length();
// 预处理回文信息
vector<vector<bool>> dp(n, vector<bool>(n, false));

// 初始化动态规划数组
for (int i = 0; i < n; i++) {
    dp[i][i] = true; // 单个字符是回文
}

for (int len = 2; len <= n; len++) {
    for (int i = 0; i <= n - len; i++) {
        int j = i + len - 1;
        if (s[i] == s[j]) {
            if (len == 2 || dp[i + 1][j - 1]) {
                dp[i][j] = true;
            }
        }
    }
}

vector<vector<string>> result;
vector<string> current;
backtrackWithDP(s, 0, dp, current, result);
return result;
}

/***
 * 方法 3：记忆化搜索
 * 使用记忆化技术避免重复计算
 */
vector<vector<string>> partitionMemo(string s) {
    int n = s.length();
    vector<vector<vector<string>>> memo(n);
    return partitionHelper(s, 0, memo);
}

private:
    /**
     * 回溯函数
     */
    void backtrack(string& s, int start, vector<string>& current, vector<vector<string>>& result)
    {
        if (start == s.length()) {

```

```

        result.push_back(current);
        return;
    }

    for (int end = start; end < s.length(); end++) {
        // 检查子串 s[start..end] 是否是回文
        if (isPalindrome(s, start, end)) {
            // 选择当前分割
            current.push_back(s.substr(start, end - start + 1));
            // 递归处理剩余部分
            backtrack(s, end + 1, current, result);
            // 回溯
            current.pop_back();
        }
    }
}

/***
 * 使用动态规划预处理的回溯函数
 */
void backtrackWithDP(string& s, int start, vector<vector<bool>>& dp,
                     vector<string>& current, vector<vector<string>>& result) {
    if (start == s.length()) {
        result.push_back(current);
        return;
    }

    for (int end = start; end < s.length(); end++) {
        if (dp[start][end]) {
            current.push_back(s.substr(start, end - start + 1));
            backtrackWithDP(s, end + 1, dp, current, result);
            current.pop_back();
        }
    }
}

/***
 * 记忆化搜索辅助函数
 */
vector<vector<string>> partitionHelper(string& s, int start,
                                         vector<vector<vector<string>>>& memo) {
    if (start == s.length()) {
        return {{}}; // 返回包含空列表的列表
    }
}

```

```

}

if (!memo[start].empty()) {
    return memo[start];
}

vector<vector<string>> result;

for (int end = start; end < s.length(); end++) {
    if (isPalindrome(s, start, end)) {
        string substring = s.substr(start, end - start + 1);
        vector<vector<string>> subResults = partitionHelper(s, end + 1, memo);

        for (auto& subResult : subResults) {
            vector<string> newResult = {substring};
            newResult.insert(newResult.end(), subResult.begin(), subResult.end());
            result.push_back(newResult);
        }
    }
}

memo[start] = result;
return result;
}

/***
 * 判断子串是否是回文
 */
bool isPalindrome(string& s, int left, int right) {
    while (left < right) {
        if (s[left] != s[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}
};

/***
 * 补充训练题目 - C++实现
 */

```

```

/***
 * LeetCode 132. 分割回文串 II
 * 给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文，返回符合要求的最少分割次数
 */
int minCut(string s) {
    int n = s.length();
    if (n <= 1) return 0;

    // dp[i] 表示 s[0..i] 的最小分割次数
    vector<int> dp(n, n);
    // isPal[i][j] 表示 s[i..j] 是否是回文
    vector<vector<bool>> isPal(n, vector<bool>(n, false));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= i; j++) {
            if (s[i] == s[j] && (i - j <= 2 || isPal[j + 1][i - 1])) {
                isPal[j][i] = true;
                if (j == 0) {
                    dp[i] = 0; // 整个字符串是回文，不需要分割
                } else {
                    dp[i] = min(dp[i], dp[j - 1] + 1);
                }
            }
        }
    }

    return dp[n - 1];
}

/***
 * LeetCode 93. 复原 IP 地址
 * 给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式
 */
vector<string> restoreIpAddresses(string s) {
    vector<string> result;
    vector<string> current;
    backtrackIP(s, 0, current, result);
    return result;
}

void backtrackIP(string& s, int start, vector<string>& current, vector<string>& result) {
    if (current.size() == 4) {

```

```

        if (start == s.length()) {
            result.push_back(current[0] + "." + current[1] + "." + current[2] + "." +
current[3]);
        }
        return;
    }

for (int len = 1; len <= 3; len++) {
    if (start + len > s.length()) break;

    string segment = s.substr(start, len);

    // 检查段是否有效
    if ((segment[0] == '0' && segment.length() > 1) || stoi(segment) > 255) {
        continue;
    }

    current.push_back(segment);
    backtrackIP(s, start + len, current, result);
    current.pop_back();
}
}

/***
 * LeetCode 140. 单词拆分 II
 * 给定一个非空字符串 s 和一个包含非空单词列表的字典，在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。
 */
vector<string> wordBreak(string s, vector<string>& wordDict) {
    vector<string> result;
    string current;
    backtrackWordBreak(s, 0, wordDict, current, result);
    return result;
}

void backtrackWordBreak(string& s, int start, vector<string>& wordDict,
                      string& current, vector<string>& result) {
    if (start == s.length()) {
        result.push_back(current);
        return;
    }

    for (string& word : wordDict) {

```

```

int len = word.length();
if (start + len <= s.length() && s.substr(start, len) == word) {
    string newCurrent = current.empty() ? word : current + " " + word;
    backtrackWordBreak(s, start + len, wordDict, newCurrent, result);
}
}

/***
 * LeetCode 131. 分割回文串（优化版）
 * 使用中心扩展法预处理回文信息
 */
vector<vector<string>> partitionOptimized(string s) {
    int n = s.length();
    vector<vector<bool>> dp(n, vector<bool>(n, false));

    // 预处理回文信息
    for (int i = 0; i < n; i++) {
        dp[i][i] = true;
    }

    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j] && (len == 2 || dp[i + 1][j - 1])) {
                dp[i][j] = true;
            }
        }
    }
}

vector<vector<string>> result;
vector<string> current;

function<void(int)> backtrack = [&](int start) {
    if (start == n) {
        result.push_back(current);
        return;
    }

    for (int end = start; end < n; end++) {
        if (dp[start][end]) {
            current.push_back(s.substr(start, end - start + 1));
            backtrack(end + 1);
        }
    }
}

```

```

        current.pop_back();
    }
}

};

backtrack(0);
return result;
}

// 测试函数
void testPalindromePartitioning() {
    PalindromePartitioningSolver solver;

    // 测试用例 1
    string s1 = "aab";
    vector<vector<string>> result1 = solver.partition(s1);
    vector<vector<string>> result1_dp = solver.partitionWithDP(s1);
    vector<vector<string>> result1_memo = solver.partitionMemo(s1);

    cout << "测试用例 1 - 字符串: " << s1 << endl;
    cout << "回溯算法结果数量: " << result1.size() << endl;
    cout << "动态规划预处理结果数量: " << result1_dp.size() << endl;
    cout << "记忆化搜索结果数量: " << result1_memo.size() << endl;

    cout << "具体分割方案:" << endl;
    for (int i = 0; i < result1.size(); i++) {
        cout << "方案 " << i + 1 << ":" [";
        for (int j = 0; j < result1[i].size(); j++) {
            cout << "\"" << result1[i][j] << "\"";
            if (j < result1[i].size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
    cout << endl;
}

// 测试用例 2
string s2 = "a";
vector<vector<string>> result2 = solver.partition(s2);

cout << "测试用例 2 - 字符串: " << s2 << endl;
cout << "结果数量: " << result2.size() << endl;
cout << "具体分割方案:" << endl;
for (auto& partition : result2) {

```

```

cout << "[";
for (int j = 0; j < partition.size(); j++) {
    cout << "\"" << partition[j] << "\"";
    if (j < partition.size() - 1) cout << ", ";
}
cout << "]" << endl;
}

cout << endl;

// 测试补充题目
cout << "==== 补充训练题目测试 ===" << endl;

// 测试分割回文串 II
string s3 = "aab";
cout << "分割回文串 II - 字符串: \" " << s3 << "\" -> 最少分割次数: " << minCut(s3) << endl;

// 测试复原 IP 地址
string s4 = "25525511135";
vector<string> ipResults = restoreIpAddresses(s4);
cout << "复原 IP 地址 - 字符串: \" " << s4 << "\" -> 结果数量: " << ipResults.size() << endl;
for (string& ip : ipResults) {
    cout << "IP 地址: " << ip << endl;
}
}

int main() {
    testPalindromePartitioning();
    return 0;
}

/***
 * 算法技巧总结 - C++版本
 *
 * 核心概念:
 * 1. 回溯算法框架:
 *     - 选择: 选择当前分割点
 *     - 约束: 检查子串是否是回文
 *     - 目标: 完成整个字符串的分割
 *     - 回溯: 撤销当前选择, 尝试其他选择
 *
 * 2. 优化技术:
 *     - 动态规划预处理: 提前计算回文信息
 *     - 记忆化搜索: 缓存中间结果避免重复计算
 */

```

- * - 剪枝优化：提前终止无效分支
- *
- * 3. 字符串处理技巧：
 - 子串提取：使用 substr 函数
 - 回文判断：双指针法或动态规划
 - 边界处理：空字符串、单字符等情况
- *
- * 调试技巧：
 - 1. 打印中间状态：跟踪回溯过程
 - 2. 边界值测试：测试各种边界情况
 - 3. 性能分析：比较不同算法的执行时间
- *
- * 工程化实践：
 - 1. 模块化设计：分离算法逻辑和业务逻辑
 - 2. 异常安全：确保资源正确释放
 - 3. 代码可读性：使用有意义的变量名和注释

文件：Code04_PalindromePartitioning.java

```
=====
package class043;

import java.util.*;

/**
 * 分割回文串问题
 *
 * 问题描述：
 * 给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是回文串。
 * 返回 s 所有可能的分割方案。
 *
 * 算法思路：
 * 1. 这是一个典型的回溯算法问题，需要找出所有满足条件的分割方案
 * 2. 使用回溯算法遍历所有可能的分割点，找到所有满足条件的分割方案
 * 3. 在每一步尝试不同的分割点，通过递归和回溯来探索所有可能性
 * 4. 需要预先处理判断子串是否为回文串，可以使用动态规划优化
 *
 * 时间复杂度分析：
 * - 最坏情况下需要尝试所有可能的分割方案
 * - 字符串长度为 n，分割方案数为 O(2^(n-1))
 * - 对于每个分割方案，需要验证每个子串是否为回文串
=====
```

* - 总时间复杂度为 $O(n * 2^n)$

*

* 空间复杂度分析:

* - 主要空间消耗是递归栈深度和存储结果的数组

* - 递归深度最大为 n , 空间复杂度为 $O(n)$

* - 动态规划预处理需要 $O(n^2)$ 空间

* - 存储结果需要 $O(2^n * n)$ 空间

*

* 工程化考量:

* 1. 异常处理: 对输入数据进行校验

* 2. 性能优化: 使用动态规划预处理回文串判断

* 3. 可配置性: 可以调整算法策略 (是否使用预处理)

* 4. 鲁棒性: 处理边界情况, 如空字符串或单字符字符串

*

* 相关题目:

* 1. LeetCode 131. 分割回文串 - <https://leetcode.cn/problems/palindrome-partitioning/>

* 2. LeetCode 132. 分割回文串 II - <https://leetcode.cn/problems/palindrome-partitioning-ii/>

* 3. LeetCode 93. 复原 IP 地址 - <https://leetcode.cn/problems/restore-ip-addresses/>

* 4. LeetCode 140. 单词拆分 II - <https://leetcode.cn/problems/word-break-ii/>

* 5. LeetCode 301. 删除无效的括号 - <https://leetcode.cn/problems/remove-invalid-parentheses/>

* 6. LeetCode 491. 递增子序列 - <https://leetcode.cn/problems/increasing-subsequences/>

* 7. LeetCode 5. 最长回文子串 - <https://leetcode.cn/problems/longest-palindromic-substring/>

* 8. LeetCode 647. 回文子串 - <https://leetcode.cn/problems/palindromic-substrings/>

* 9. LeetCode 516. 最长回文子序列 - <https://leetcode.cn/problems/longest-palindromic-subsequence/>

* 10. LeetCode 336. 回文对 - <https://leetcode.cn/problems/palindrome-pairs/>

* 11. LeetCode 479. 最大回文数乘积 - <https://leetcode.cn/problems/largest-palindrome-product/>

* 12. LeetCode 680. 验证回文字符串 II - <https://leetcode.cn/problems/valid-palindrome-ii/>

* 13. 牛客网 - 分割回文串 - <https://www.nowcoder.com/practice/1025ffc2939547e39e8a38a955de1dd3>

* 14. 牛客网 - 最长回文子串 - <https://www.nowcoder.com/practice/b4525d1d84934cf280439aeecc36f4af>

* 15. 牛客网 - 复原 IP 地址 - <https://www.nowcoder.com/practice/ce73540d47374dbe85b3125f57727e1e>

* 16. Codeforces 1327D - Infinite Path - <https://codeforces.com/problemset/problem/1327/D>

* 17. Codeforces 1436E - Complicated Construction -

<https://codeforces.com/problemset/problem/1436/E>

* 18. Codeforces 1332B - Composite Coloring - <https://codeforces.com/problemset/problem/1332/B>

* 19. AtCoder ABC144D - Water Bottle - https://atcoder.jp/contests/abc144/tasks/abc144_d

* 20. AtCoder ABC175D - Moving Piece - https://atcoder.jp/contests/abc175/tasks/abc175_d

* 21. AtCoder ABC159E - Dividing Chocolate - https://atcoder.jp/contests/abc159/tasks/abc159_e

* 22. 洛谷 P1120 - 小木棍 - <https://www.luogu.com.cn/problem/P1120>

* 23. 洛谷 P1540 - [NOIP2010 提高组] 机器翻译 - <https://www.luogu.com.cn/problem/P1540>

* 24. 洛谷 P1157 - 组合的输出 - <https://www.luogu.com.cn/problem/P1157>

* 25. 洛谷 P1048 - [NOIP2005 普及组] 采药 - <https://www.luogu.com.cn/problem/P1048>

* 26. HackerRank - Palindromic Substrings - <https://www.hackerrank.com/challenges/palindromic-substrings>

```

substrings/problem

* 27. HackerRank - Split the String - https://www.hackerrank.com/challenges/split-the-string/problem

* 28. HackerRank - Sherlock and Cost - https://www.hackerrank.com/challenges/sherlock-and-cost/problem

* 29. HackerRank - The Longest Palindromic Subsequence -
https://www.hackerrank.com/challenges/longest-palindromic-subsequence/problem

* 30. UVa 10945 - Mother Bear -
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=1886

* 31. UVa 10189 - Minesweeper -
https://onlinejudge.org/index.php?option=com\_onlinejudge&Itemid=8&category=24&page=show\_problem&problem=1886

* 32. POJ 3083 - Children of the Candy Corn - http://poj.org/problem?id=3083

* 33. POJ 1163 - The Triangle - http://poj.org/problem?id=1163

* 34. HDU 1527 - 取石子游戏 - https://acm.hdu.edu.cn/showproblem.php?pid=1527

* 35. HDU 1203 - I NEED A OFFER! - https://acm.hdu.edu.cn/showproblem.php?pid=1203

* 36. LintCode 1000. 至少有 K 个重复字符的最长子串 - https://www.lintcode.com/problem/1000/

* 37. LintCode 1178. 验证回文字符串 II - https://www.lintcode.com/problem/1178/

* 38. LintCode 125. 背包问题 II - https://www.lintcode.com/problem/125/

* 39. LintCode 200. 最长回文子串 - https://www.lintcode.com/problem/200/

* 40. LintCode 130. 堆化 - https://www.lintcode.com/problem/130/

* 41. LintCode 135. 数字组合 - https://www.lintcode.com/problem/135/

* 42. LeetCode 31. 下一个排列 - https://leetcode.cn/problems/next-permutation/

* 43. LeetCode 77. 组合 - https://leetcode.cn/problems/combinations/

* 44. LeetCode 79. 单词搜索 - https://leetcode.cn/problems/word-search/

* 45. LeetCode 17. 电话号码的字母组合 - https://leetcode.cn/problems/letter-combinations-of-a-phone-number/

* 46. LeetCode 22. 括号生成 - https://leetcode.cn/problems/generate-parentheses/

* 47. LeetCode 10. 正则表达式匹配 - https://leetcode.cn/problems/regular-expression-matching/

* 48. LeetCode 37. 解数独 - https://leetcode.cn/problems/sudoku-solver/

* 49. LeetCode 51. N 皇后 - https://leetcode.cn/problems/n-queens/

* 50. LeetCode 52. N 皇后 II - https://leetcode.cn/problems/n-queens-ii/

*/
public class Code04_PalindromePartitioning {

    /**
     * 分割回文串主函数
     *
     * @param s 输入字符串
     * @return 所有可能的分割方案
     *
     * 算法思路:
    
```

```

* 1. 使用回溯算法枚举所有可能的分割方案
* 2. 预先使用动态规划处理所有子串是否为回文串
* 3. 在回溯过程中，只考虑是回文串的子串进行分割
*
* 优化点：
* 1. 使用动态规划预处理回文串判断，避免重复计算
* 2. 在回溯过程中剪枝，只考虑有效的分割点
*/
public static List<List<String>> partition(String s) {
    List<List<String>> result = new ArrayList<>();
    if (s == null || s.length() == 0) {
        return result;
    }

    // 预处理：使用动态规划判断所有子串是否为回文串
    boolean[][] isPalindrome = precomputePalindromes(s);

    // 回溯搜索所有分割方案
    backtrack(s, 0, new ArrayList<>(), result, isPalindrome);

    return result;
}

/**
 * 预处理所有子串是否为回文串
 *
 * @param s 输入字符串
 * @return 二维布尔数组，isPalindrome[i][j]表示 s[i.. j] 是否为回文串
 *
 * 算法思路：
 * 1. 使用动态规划，dp[i][j]表示 s[i.. j] 是否为回文串
 * 2. 状态转移方程：
 *   - 如果 i == j，则 dp[i][j] = true (单字符必为回文串)
 *   - 如果 j - i == 1 且 s[i] == s[j]，则 dp[i][j] = true (两字符相等为回文串)
 *   - 如果 s[i] == s[j] 且 dp[i+1][j-1] = true，则 dp[i][j] = true
 *
 * 时间复杂度：O(n^2)
 * 空间复杂度：O(n^2)
 */
private static boolean[][] precomputePalindromes(String s) {
    int n = s.length();
    boolean[][] dp = new boolean[n][n];

```

```

// 单字符必为回文串
for (int i = 0; i < n; i++) {
    dp[i][i] = true;
}

// 两字符情况
for (int i = 0; i < n - 1; i++) {
    if (s.charAt(i) == s.charAt(i + 1)) {
        dp[i][i + 1] = true;
    }
}

// 长度大于 2 的子串
for (int len = 3; len <= n; len++) {
    for (int i = 0; i <= n - len; i++) {
        int j = i + len - 1;
        if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
            dp[i][j] = true;
        }
    }
}

return dp;
}

/***
 * 回溯函数，寻找所有可能的分割方案
 *
 * @param s 输入字符串
 * @param start 当前处理的起始位置
 * @param path 当前分割路径
 * @param result 存储所有分割方案的结果集
 * @param isPalindrome 预处理的回文串判断数组
 *
 * 递归思路：
 * 1. 基础情况：如果 start 等于字符串长度，说明已经处理完所有字符，将当前路径加入结果集
 * 2. 递归情况：从 start 位置开始，尝试所有可能的结束位置 end
 * 3. 如果 s[start..end] 是回文串，则将其加入路径，递归处理剩余部分
 * 4. 递归返回后，回溯移除刚加入的子串
 */
private static void backtrack(String s, int start, List<String> path, List<List<String>>
result, boolean[][] isPalindrome) {
    // 基础情况：已经处理完所有字符

```

```

if (start == s.length()) {
    result.add(new ArrayList<>(path));
    return;
}

// 递归情况：尝试所有可能的分割点
for (int end = start; end < s.length(); end++) {
    // 如果当前子串是回文串，则进行分割
    if (isPalindrome[start][end]) {
        // 做选择：将当前子串加入路径
        path.add(s.substring(start, end + 1));

        // 递归：处理剩余部分
        backtrack(s, end + 1, path, result, isPalindrome);

        // 撤销选择：回溯移除刚加入的子串
        path.remove(path.size() - 1);
    }
}
}

// 测试函数
public static void main(String[] args) {
    // 测试用例 1
    String s1 = "aab";
    List<List<String>> result1 = partition(s1);
    System.out.println("输入: " + s1 + "\n");
    System.out.println("输出: " + result1);
    // 预期输出: [[["a"], ["a"], ["b"]], [["aa"], ["b"]]]

    // 测试用例 2
    String s2 = "a";
    List<List<String>> result2 = partition(s2);
    System.out.println("\n 输入: " + s2 + "\n");
    System.out.println("输出: " + result2);
    // 预期输出: [[["a"]]]

    // 测试用例 3
    String s3 = "abc";
    List<List<String>> result3 = partition(s3);
    System.out.println("\n 输入: " + s3 + "\n");
    System.out.println("输出: " + result3);
    // 预期输出: [[["a"], ["b"], ["c"]]]
}

```

```
}
```

```
/**
```

```
* 补充训练题目
```

```
*
```

```
* 1. LeetCode 132. 分割回文串 II
```

```
* 题目描述：给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是回文串。返回符合要求的最少分割次数。
```

```
* 解题思路：使用动态规划，dp[i]表示前 i 个字符的最少分割次数
```

```
* Java 实现示例：
```

```
* public int minCut(String s) {  
*     int n = s.length();  
*     // 预处理回文串  
*     boolean[][] isPalindrome = precomputePalindromes(s);  
  
*     // dp[i]表示前 i 个字符的最少分割次数  
*     int[] dp = new int[n + 1];  
*     Arrays.fill(dp, Integer.MAX_VALUE);  
*     dp[0] = -1; // 初始条件  
  
*     for (int i = 1; i <= n; i++) {  
*         for (int j = 0; j < i; j++) {  
*             if (isPalindrome[j][i - 1]) {  
*                 dp[i] = Math.min(dp[i], dp[j] + 1);  
*             }  
*         }  
*     }  
  
*     return dp[n];  
* }
```

```
* 2. LeetCode 93. 复原 IP 地址
```

```
* 题目描述：给定一个只包含数字的字符串，用以表示一个 IP 地址，返回所有可能从 s 获得的有效 IP 地址。
```

```
* 解题思路：使用回溯算法，枚举每个段的长度（1-3），并验证是否有效
```

```
* Java 实现示例：
```

```
* public List<String> restoreIpAddresses(String s) {  
*     List<String> result = new ArrayList<>();  
*     if (s == null || s.length() < 4 || s.length() > 12) return result;  
*     backtrackIp(s, 0, 0, "", result);  
*     return result;  
* }
```

* private void backtrackIp(String s, int start, int segment, String current, List<String>

```

result) {
    *     if (start == s.length() && segment == 4) {
    *         result.add(current.substring(0, current.length() - 1));
    *         return;
    *     }
    *     if (segment >= 4) return;
    *
    *     for (int len = 1; len <= 3 && start + len <= s.length(); len++) {
    *         String part = s.substring(start, start + len);
    *         // 检查是否是有效的 IP 段
    *         if (isValidIpSegment(part)) {
    *             backtrackIp(s, start + len, segment + 1, current + part + ".", result);
    *         }
    *     }
    *
    *     private boolean isValidIpSegment(String segment) {
    *         if (segment.length() > 1 && segment.charAt(0) == '0') return false;
    *         int num = Integer.parseInt(segment);
    *         return num >= 0 && num <= 255;
    *     }
    *
    * 3. LeetCode 140. 单词拆分 II
    * 题目描述：给定一个字符串 s 和一个字符串字典 wordDict，在字符串 s 中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有可能的句子。
    * 解题思路：使用回溯算法，结合动态规划预处理
    * Java 实现示例：
    * public List<String> wordBreak(String s, List<String> wordDict) {
    *     Set<String> wordSet = new HashSet<>(wordDict);
    *     int n = s.length();
    *
    *     // 预处理：dp[i] 表示前 i 个字符是否可以拆分
    *     boolean[] dp = new boolean[n + 1];
    *     dp[0] = true;
    *     for (int i = 1; i <= n; i++) {
    *         for (int j = 0; j < i; j++) {
    *             if (dp[j] && wordSet.contains(s.substring(j, i))) {
    *                 dp[i] = true;
    *                 break;
    *             }
    *         }
    *     }
    *
    *     List<String> result = new ArrayList<>();

```

```

*         if (dp[n]) {
*             backtrackWordBreak(s, 0, new ArrayList<>(), result, wordSet);
*         }
*     return result;
* }
* private void backtrackWordBreak(String s, int start, List<String> path, List<String>
result, Set<String> wordSet) {
*     if (start == s.length()) {
*         result.add(String.join(" ", path));
*         return;
*     }
*
*     for (int end = start + 1; end <= s.length(); end++) {
*         String word = s.substring(start, end);
*         if (wordSet.contains(word)) {
*             path.add(word);
*             backtrackWordBreak(s, end, path, result, wordSet);
*             path.remove(path.size() - 1);
*         }
*     }
* }
*
* 4. Codeforces 1327D – Infinite Path
* 题目描述：给定一个置换 p 和一个颜色数组 c，找到最小的 k，使得存在一个起始点 s，使得沿路径
s → p[s] → p[p[s]] → ... 每 k 步的颜色都相同。
* 解题思路：分析置换的循环结构，对每个循环尝试不同的 k 值
*
* 5. HackerRank – Split the String
* 题目描述：给定一个只包含 a 和 b 的字符串，将其分割成尽可能多的部分，使得每个部分中的 a 和 b
的数量相等。
* 解题思路：使用贪心算法，记录当前 a 和 b 的数量，相等时进行分割
* Java 实现示例：
* public static int splitString(String s) {
*     int countA = 0, countB = 0;
*     int splits = 0;
*
*     for (char c : s.toCharArray()) {
*         if (c == 'a') countA++;
*         else countB++;
*
*         if (countA == countB) {
*             splits++;
*             countA = 0;
*         }
*     }
*     return splits;
}

```

```

*
*         countB = 0;
*
*     }
*
* }
*
*     return splits;
* }

*
* 6. LeetCode 647. 回文子串
* 题目描述: 给你一个字符串 s , 请你统计并返回这个字符串中 回文子串 的数目。
* 解题思路: 使用中心扩展法, 枚举所有可能的回文中心
* Java 实现示例:
* public int countSubstrings(String s) {
*     int count = 0;
*     for (int i = 0; i < s.length(); i++) {
*         // 奇数长度回文子串
*         count += expandAroundCenter(s, i, i);
*         // 偶数长度回文子串
*         count += expandAroundCenter(s, i, i + 1);
*     }
*     return count;
* }
*
* private int expandAroundCenter(String s, int left, int right) {
*     int count = 0;
*     while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
*         count++;
*         left--;
*         right++;
*     }
*     return count;
* }
*
* 7. LeetCode 516. 最长回文子序列
* 题目描述: 给你一个字符串 s , 找出其中最长的回文子序列, 并返回该序列的长度。
* 解题思路: 使用动态规划, dp[i][j]表示 s[i..j] 中的最长回文子序列长度
* Java 实现示例:
* public int longestPalindromeSubseq(String s) {
*     int n = s.length();
*     int[][] dp = new int[n][n];
*
*     // 单个字符的回文子序列长度为 1
*     for (int i = 0; i < n; i++) {
*         dp[i][i] = 1;
*     }

```

```

*
* // 从短到长处理子串
* for (int len = 2; len <= n; len++) {
*     for (int i = 0; i <= n - len; i++) {
*         int j = i + len - 1;
*         if (s.charAt(i) == s.charAt(j)) {
*             dp[i][j] = dp[i + 1][j - 1] + 2;
*         } else {
*             dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
*         }
*     }
* }
*
* return dp[0][n - 1];
}

```

* 8. LeetCode 336. 回文对

* 题目描述：给定一个字符串数组 words，找出所有不同的索引对 (i, j)，使得 words[i] + words[j] 是一个回文串。

* 解题思路：使用哈希表存储反转的字符串，查找可能的回文对

* Java 实现示例：

```

public List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> result = new ArrayList<>();
    Map<String, Integer> map = new HashMap<>();

    // 存储每个单词的反转及其索引
    for (int i = 0; i < words.length; i++) {
        map.put(new StringBuilder(words[i]).reverse().toString(), i);
    }

    for (int i = 0; i < words.length; i++) {
        String word = words[i];

        // 情况 1：空字符串与回文字符串
        if (map.containsKey("") && map.get("") != i && isPalindrome(word)) {
            result.add(Arrays.asList(i, map.get("")));
        }

        for (int j = 0; j < word.length(); j++) {
            String left = word.substring(0, j);
            String right = word.substring(j);

            // 情况 2：left 是回文，寻找 right 的反转

```

```

*             if (map.containsKey(left) && map.get(left) != i && isPalindrome(right)) {
*                 result.add(Arrays.asList(i, map.get(left)));
*             }
*
*             // 情况 3: right 是回文, 寻找 left 的反转
*             if (map.containsKey(right) && map.get(right) != i && isPalindrome(left)) {
*                 result.add(Arrays.asList(map.get(right), i));
*             }
*         }
*
*     }
*
*     return result;
}
}

private boolean isPalindrome(String s) {
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left++) != s.charAt(right--)) {
            return false;
        }
    }
    return true;
}
*/

```

```

/**
 * 分割回文串问题的算法技巧总结
 *
 * 核心概念:
 * - 回文串: 正序和倒序读都是一样的字符串
 * - 分割问题: 将字符串分割成若干满足条件的子串
 * - 回溯算法: 通过递归和回溯探索所有可能的分割方案
 * - 动态规划: 用于预处理和优化回文判断
 *
 * 算法设计:
 * 1. 回溯算法框架:
 *   - 状态定义: 当前处理的起始位置、当前分割路径
 *   - 选择: 尝试所有可能的分割点
 *   - 约束: 当前子串必须是回文串
 *   - 目标: 处理完整个字符串
 *
 * 2. 回文串判断优化:
 *   - 动态规划预处理: O(n^2)时间和空间, 避免重复计算
 *   - 中心扩展法: 适用于统计回文子串数量

```

- * - Manacher 算法：线性时间复杂度的回文串查找算法
- *
- * 复杂度分析：
 - * - 时间复杂度: $O(n * 2^n)$ - 最坏情况下需要枚举所有可能的分割方案
 - * - 空间复杂度：
 - * - 递归栈: $O(n)$
 - * - 动态规划数组: $O(n^2)$
 - * - 结果存储: $O(2^n * n)$
 - *
- * 优化技巧：
 - * 1. 预处理优化：
 - * - 使用动态规划预先计算所有子串的回文性质
 - * - 避免在回溯过程中重复判断子串是否为回文
 - *
 - * 2. 剪枝策略：
 - * - 提前排除不可能的分割路径
 - * - 对于某些变种问题（如最少分割次数），可以使用贪心策略
 - *
 - * 3. 记忆化搜索：
 - * - 缓存中间结果，避免重复计算
 - * - 适用于需要多次查询的场景
 - *
- * 调试技巧：
 - * 1. 打印中间状态：
 - * - 记录当前分割路径和处理位置
 - * - 验证回文串判断的正确性
 - *
 - * 2. 边界测试：
 - * - 空字符串
 - * - 单字符串
 - * - 全回文字符串
 - * - 无回文子串
 - *
 - * 3. 性能分析：
 - * - 使用性能分析工具找出瓶颈
 - * - 针对大规模数据进行优化
 - *
- * 跨语言实现注意事项：
 - * 1. 字符串处理：
 - * - 不同语言的字符串操作效率差异
 - * - Java 中 String 是不可变的，频繁拼接会产生额外开销
 - * - Python 中的字符串切片效率较高
 - *

- * 2. 递归限制:

- Python 默认的递归深度限制为 1000
 - Java 没有明确的递归深度限制, 但过深会导致栈溢出
 - 对于长字符串, 可以考虑迭代实现

- *

- * 3. 数据结构选择:

- 使用合适的集合类存储结果
 - 考虑使用 StringBuilder 等可变字符串类

- *

- * 工程化考量:

- * 1. 输入验证:

- 检查 null、空字符串等边界情况
 - 验证输入参数的有效性

- *

- * 2. 代码可读性:

- 提取辅助函数, 如回文判断、预处理等
 - 添加详细的注释和文档

- *

- * 3. 可测试性:

- 设计单元测试用例
 - 覆盖各种边界情况

- *

- * 4. 可扩展性:

- 设计通用的回溯框架
 - 支持不同的回文判断策略

- *

- * 与其他算法的结合:

- 1. 与回文串算法结合:
 - Manacher 算法用于高效查找最长回文子串
 - 中心扩展法用于统计回文子串数量

- *

- * 2. 与动态规划结合:

- 优化回文判断
 - 求解最少分割次数等优化问题

- *

- * 3. 与哈希算法结合:

- 使用哈希表预处理字符串的反转
 - 用于解决回文对问题

- *

- * 4. 与机器学习的联系:

- 可以将回文识别作为自然语言处理的基础任务
 - 使用深度学习模型自动识别回文模式
 - 在文本生成中应用回文结构

```
*/
```

```
/**
```

```
* 补充题目 9: LeetCode 46. 全排列
```

```
* 题目描述: 给定一个不含重复数字的数组 nums , 返回其 所有可能的全排列 。你可以 按任意顺序 返回答案。
```

```
* 链接: https://leetcode.cn/problems/permutations/
```

```
*
```

```
* 解题思路:
```

```
* - 使用回溯算法, 枚举每个位置可能的数字
```

```
* - 用一个 used 数组标记已经使用过的数字
```

```
* - 当路径长度等于数组长度时, 将当前路径加入结果集
```

```
* - 时间复杂度: O(n * n!) , 空间复杂度: O(n)
```

```
*/
```

```
public List<List<Integer>> permute(int[] nums) {
```

```
    List<List<Integer>> result = new ArrayList<>();
```

```
    if (nums == null || nums.length == 0) {
```

```
        return result;
```

```
}
```

```
    boolean[] used = new boolean[nums.length];
```

```
    backtrackPermute(nums, used, new ArrayList<>(), result);
```

```
    return result;
```

```
}
```

```
private void backtrackPermute(int[] nums, boolean[] used, List<Integer> path,  
List<List<Integer>> result) {
```

```
    if (path.size() == nums.length) {
```

```
        result.add(new ArrayList<>(path));
```

```
        return;
```

```
}
```

```
for (int i = 0; i < nums.length; i++) {
```

```
    if (used[i]) {
```

```
        continue;
```

```
}
```

```
// 做选择
```

```
    used[i] = true;
```

```
    path.add(nums[i]);
```

```
// 递归
```

```
    backtrackPermute(nums, used, path, result);
```

```
// 回溯
```

```
    path.remove(path.size() - 1);
```

```
    used[i] = false;
```

```

    }
}

/***
 * 补充题目 10: LeetCode 47. 全排列 II
 * 题目描述: 给定一个可包含重复数字的序列 nums , 按任意顺序 返回所有不重复的全排列。
 * 链接: https://leetcode.cn/problems/permutations-ii/
 *
 * 解题思路:
 * - 先对数组进行排序, 确保相同的数字相邻
 * - 使用回溯算法, 同时添加去重逻辑
 * - 当前数字与前一个数字相同且前一个数字未使用过时, 跳过当前数字
 * - 时间复杂度: O(n * n!), 空间复杂度: O(n)
 */

public List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    if (nums == null || nums.length == 0) {
        return result;
    }
    // 排序, 确保相同的数字相邻
    Arrays.sort(nums);
    boolean[] used = new boolean[nums.length];
    backtrackPermuteUnique(nums, used, new ArrayList<>(), result);
    return result;
}

private void backtrackPermuteUnique(int[] nums, boolean[] used, List<Integer> path,
List<List<Integer>> result) {
    if (path.size() == nums.length) {
        result.add(new ArrayList<>(path));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 去重逻辑: 当前数字与前一个数字相同且前一个数字未使用过时, 跳过当前数字
        if (used[i] || (i > 0 && nums[i] == nums[i - 1] && !used[i - 1])) {
            continue;
        }
        // 做选择
        used[i] = true;
        path.add(nums[i]);
        // 递归
        backtrackPermuteUnique(nums, used, path, result);
    }
}

```

```

        // 回溯
        path.remove(path.size() - 1);
        used[i] = false;
    }
}

/***
 * 补充题目 11: LeetCode 78. 子集
 * 题目描述: 给你一个整数数组 nums，数组中的元素 互不相同。返回该数组所有可能的子集（幂集）。
 * 链接: https://leetcode.cn/problems/subsets/
 *
 * 解题思路:
 * - 使用回溯算法，枚举每个位置的元素是否被选中
 * - 对于每个元素，有两种选择：包含或不包含
 * - 时间复杂度: O(n * 2^n)，空间复杂度: O(n)
 */
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> result = new ArrayList<>();
    if (nums == null || nums.length == 0) {
        result.add(new ArrayList<>());
        return result;
    }
    backtrackSubsets(nums, 0, new ArrayList<>(), result);
    return result;
}

private void backtrackSubsets(int[] nums, int start, List<Integer> path, List<List<Integer>> result) {
    // 将当前路径加入结果集（每个节点都是一个子集）
    result.add(new ArrayList<>(path));

    for (int i = start; i < nums.length; i++) {
        // 做选择
        path.add(nums[i]);
        // 递归
        backtrackSubsets(nums, i + 1, path, result);
        // 回溯
        path.remove(path.size() - 1);
    }
}

/***

```

* 分割回文串与回溯算法的深度解析

*

* 核心概念与理论基础:

* 1. 回溯算法理论:

- * - 状态空间树模型与搜索策略
- * - 剪枝技术的数学基础
- * - 排列组合的组合数学原理
- * - 决策树与状态转换

*

* 2. 回文理论深化:

- * - 回文自动机的数学模型
- * - 回文性质的代数结构
- * - 回文串在字符串理论中的地位
- * - 回文识别的计算复杂性

*

* 3. 组合优化视角:

- * - 分割问题的最优子结构性质
- * - 贪心策略的适用条件
- * - 近似算法的性能边界
- * - 参数化复杂度分析

*

* 高级算法技术:

* 1. 回溯算法优化技术:

- * - 剪枝策略分类: 可行性剪枝、最优化剪枝、记忆化剪枝
- * - 启发式搜索与 A*算法在回溯中的应用
- * - 迭代加深搜索 (IDS) 的实现
- * - 并行回溯算法设计

*

* 2. 动态规划优化技术:

- * - 状态压缩 DP 在回文分割中的应用
- * - 区间 DP 的高级实现技巧
- * - 位运算优化状态表示
- * - 四边形不等式优化

*

* 3. 字符串处理高级技术:

- * - Manacher 算法的深入解析与扩展
- * - 后缀自动机与回文识别
- * - 哈希算法组合 (双重哈希、多项式哈希)
- * - 字符串匹配算法与回文结合

*

* 4. 计算几何方法:

- * - 回文串的几何表示
- * - 中心点枚举的几何意义

- * - 对称性利用的数学基础

- *

- * 多维度优化策略:

- * 1. 算法层面优化:

- 预计算与缓存策略设计

- 问题转换与等价变形

- 启发式规则设计

- 并行计算模型选择

- *

- * 2. 数据结构优化:

- 哈希表实现选择与优化

- 平衡树与线段树的应用

- 位图与位操作的高效应用

- 自定义数据结构设计

- *

- * 3. 系统层面优化:

- 内存管理与缓存优化

- 指令级并行优化

- 编译器优化策略

- 分布式计算框架应用

- *

- * 工程化实践指南:

- * 1. 算法选型框架:

- 问题特性分析方法论

- 算法复杂度与实际性能权衡

- 可扩展性与可维护性评估

- 跨平台实现考量

- *

- * 2. 代码质量保证:

- 模块化设计模式

- 接口抽象原则

- 单元测试策略

- 性能基准测试

- *

- * 3. 调试与优化技术:

- 可视化调试工具应用

- 性能分析与瓶颈定位

- 热点代码优化技巧

- 内存泄漏检测

- *

- * 跨学科应用:

- * 1. 密码学应用:

- 回文结构在加密算法中的应用

* - 基于回文的哈希函数设计

* - 消息完整性验证

* - 安全协议设计

*

* 2. 生物信息学:

* - DNA 序列中的回文结构分析

* - 基因调控区域识别

* - 蛋白质结构预测

* - 分子序列比对中的回文模式

*

* 3. 自然语言处理:

* - 文本分割与分析

* - 语言模型中的模式识别

* - 文本生成与修辞分析

* - 语言演化研究

*

* 4. 计算机图形学:

* - 对称图形识别

* - 模式匹配与形状分析

* - 图像处理中的回文结构

*

* 前沿研究方向:

* 1. 理论研究:

* - 回文问题的计算复杂性新结果

* - 参数化复杂度前沿进展

* - 量子计算模型下的回文识别

* - 随机字符串中的回文统计性质

*

* 2. 算法创新:

* - 大规模并行回溯算法

* - 量子启发式算法

* - 神经网络在回文识别中的应用

* - 在线算法与流式处理

*

* 3. 应用拓展:

* - 区块链中的回文应用

* - 量子通信中的回文编码

* - 分布式系统中的一致性验证

* - 大数据处理中的高效回文搜索

*

* 总结:

* 分割回文串问题作为回溯算法与字符串处理的经典结合，展现了算法设计的深刻思想和广泛应用。

* 从理论基础到工程实践，从经典算法到前沿研究，分割回文串问题涵盖了计算机科学的多个重要领域。

```
* 掌握这类问题的解题思路和优化技巧，不仅有助于解决具体的算法问题，更能培养抽象思维和问题转化能力。  
* 在未来的计算机科学发展中，分割回文串问题及其相关技术将继续在各个领域发挥重要作用，特别是在大数据处理、  
* 人工智能和量子计算等新兴领域，为解决复杂问题提供创新思路。  
*/  
}
```

文件: Code04_PalindromePartitioning.py

```
"""  
分割回文串问题 - Python 版本
```

问题描述:

给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是回文串。返回 s 所有可能的分割方案。

算法思路:

方法 1: 回溯算法 - 枚举所有可能的分割方案

方法 2: 动态规划预处理 - 优化回文串判断

方法 3: 记忆化搜索 - 缓存中间结果

时间复杂度分析:

- 回溯算法: $O(n * 2^n)$, 其中 n 为字符串长度
- 动态规划预处理: $O(n^2)$, 预处理回文信息
- 记忆化搜索: $O(n * 2^n)$, 但通过缓存减少重复计算

空间复杂度分析:

- 回溯算法: $O(n)$, 递归栈深度
- 动态规划预处理: $O(n^2)$, 存储回文信息
- 记忆化搜索: $O(n * 2^n)$, 存储所有可能的分割方案

工程化考量:

1. 剪枝优化: 提前终止无效分支
2. 记忆化搜索: 缓存中间结果
3. 边界处理: 空字符串、单字符等情况
4. 可测试性: 设计全面的测试用例

```
"""
```

```
from typing import List
```

```
class PalindromePartitioningSolver:
```

```

def __init__(self):
    pass

def partition(self, s: str) -> List[List[str]]:
    """
    方法 1: 回溯算法
    枚举所有可能的分割方案
    """
    result = []
    current = []
    self._backtrack(s, 0, current, result)
    return result

def partition_with_dp(self, s: str) -> List[List[str]]:
    """
    方法 2: 回溯算法 + 动态规划预处理
    使用动态规划预处理回文信息，优化判断效率
    """
    n = len(s)
    # 预处理回文信息
    dp = [[False] * n for _ in range(n)]

    # 初始化动态规划数组
    for i in range(n):
        dp[i][i] = True # 单个字符是回文

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            if s[i] == s[j]:
                if length == 2 or dp[i + 1][j - 1]:
                    dp[i][j] = True

    result = []
    current = []
    self._backtrack_with_dp(s, 0, dp, current, result)
    return result

def partition_memo(self, s: str) -> List[List[str]]:
    """
    方法 3: 记忆化搜索
    使用记忆化技术避免重复计算
    """

```

```

memo: dict = {}

return self._partition_helper(s, 0, memo)

def _backtrack(self, s: str, start: int, current: List[str], result: List[List[str]]):
    """
    回溯函数
    """

    if start == len(s):
        result.append(current[:]) # 添加当前分割方案的副本
        return

    for end in range(start, len(s)):
        # 检查子串 s[start..end] 是否是回文
        if self._is_palindrome(s, start, end):
            # 选择当前分割
            current.append(s[start:end + 1])
            # 递归处理剩余部分
            self._backtrack(s, end + 1, current, result)
            # 回溯
            current.pop()

def _backtrack_with_dp(self, s: str, start: int, dp: List[List[bool]],
                      current: List[str], result: List[List[str]]):
    """
    使用动态规划预处理的回溯函数
    """

    if start == len(s):
        result.append(current[:])
        return

    for end in range(start, len(s)):
        if dp[start][end]:
            current.append(s[start:end + 1])
            self._backtrack_with_dp(s, end + 1, dp, current, result)
            current.pop()

def _partition_helper(self, s: str, start: int, memo: dict) -> List[List[str]]:
    """
    记忆化搜索辅助函数
    """

    if start == len(s):
        return [[]] # 返回包含空列表的列表

```

```

if start in memo:
    return memo[start]

result = []

for end in range(start, len(s)):
    if self._is_palindrome(s, start, end):
        substring = s[start:end + 1]
        sub_results = self._partition_helper(s, end + 1, memo)

        for sub_result in sub_results:
            new_result = [substring] + sub_result
            result.append(new_result)

memo[start] = result
return result

```

```

def _is_palindrome(self, s: str, left: int, right: int) -> bool:
    """
    判断子串是否是回文
    """

    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

```

补充训练题目 - Python 实现

```

def min_cut(s: str) -> int:
    """
    LeetCode 132. 分割回文串 II
    给定一个字符串 s，将 s 分割成一些子串，使每个子串都是回文串，返回符合要求的最少分割次数
    """

    n = len(s)
    if n <= 1:
        return 0

    # dp[i]表示 s[0.. i] 的最小分割次数
    dp = [n] * n
    # is_pal[i][j] 表示 s[i.. j] 是否是回文
    is_pal = [[False] * n for _ in range(n)]

```

```

for i in range(n):
    for j in range(i + 1):
        if s[i] == s[j] and (i - j <= 2 or is_pal[j + 1][i - 1]):
            is_pal[j][i] = True
        if j == 0:
            dp[i] = 0 # 整个字符串是回文，不需要分割
        else:
            dp[i] = min(dp[i], dp[j - 1] + 1)

return dp[n - 1]

```

```
def restore_ip_addresses(s: str) -> List[str]:
```

LeetCode 93. 复原 IP 地址

给定一个只包含数字的字符串，复原它并返回所有可能的 IP 地址格式

"""

```
def backtrack_ip(start: int, current: List[str]):
```

```
    if len(current) == 4:
        if start == len(s):
            result.append(".".join(current))
        return
```

```
    for length in range(1, 4):
```

```
        if start + length > len(s):
```

```
            break
```

```
        segment = s[start:start + length]
```

检查段是否有效

```
        if (segment[0] == '0' and len(segment) > 1) or int(segment) > 255:
            continue
```

```
        current.append(segment)
```

```
        backtrack_ip(start + length, current)
```

```
        current.pop()
```

```
result = []
```

```
backtrack_ip(0, [])
```

```
return result
```

```
def word_break(s: str, word_dict: List[str]) -> List[str]:
```

"""

LeetCode 140. 单词拆分 II

给定一个非空字符串 s 和一个包含非空单词列表的字典，在字符串中增加空格来构建一个句子，使得句子中所有的单词都在词典中。返回所有这些可能的句子。

```
"""
def backtrack_word_break(start: int, current: List[str]):
    if start == len(s):
        result.append(" ".join(current))
        return

    for word in word_dict:
        length = len(word)
        if start + length <= len(s) and s[start:start + length] == word:
            current.append(word)
            backtrack_word_break(start + length, current)
            current.pop()

    result = []
backtrack_word_break(0, [])
return result
```

def partition_optimized(s: str) -> List[List[str]]:

```
"""
LeetCode 131. 分割回文串（优化版）
使用中心扩展法预处理回文信息
"""

n = len(s)
dp = [[False] * n for _ in range(n)]
```

```
# 预处理回文信息
```

```
for i in range(n):
    dp[i][i] = True
```

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

```
    for i in range(n - length + 1):
```

```
        j = i + length - 1
```

```
        if s[i] == s[j] and (length == 2 or dp[i + 1][j - 1]):
```

```
            dp[i][j] = True
```

```
result = []
```

```
current = []
```

def backtrack(start: int):

```
    if start == n:
```

```

        result.append(current[:])
        return

    for end in range(start, n):
        if dp[start][end]:
            current.append(s[start:end + 1])
            backtrack(end + 1)
            current.pop()

    backtrack(0)
    return result

def generate_palindromic_decompositions(s: str) -> List[List[str]]:
    """
    生成所有回文分解方案（扩展功能）
    支持生成所有可能的回文子串分解
    """
    def backtrack_decompositions(start: int, path: List[str]):
        if start == len(s):
            decompositions.append(path[:])
            return

        for end in range(start, len(s)):
            substring = s[start:end + 1]
            if substring == substring[::-1]: # 检查是否是回文
                path.append(substring)
                backtrack_decompositions(end + 1, path)
                path.pop()

    decompositions = []
    backtrack_decompositions(0, [])
    return decompositions

# 测试函数
def test_palindrome_partitioning():
    """测试函数"""
    solver = PalindromePartitioningSolver()

    # 测试用例 1
    s1 = "aab"
    result1 = solver.partition(s1)
    result1_dp = solver.partition_with_dp(s1)
    result1_memo = solver.partition_memo(s1)

```

```
print("测试用例 1 - 字符串:", repr(s1))
print("回溯算法结果数量:", len(result1))
print("动态规划预处理结果数量:", len(result1_dp))
print("记忆化搜索结果数量:", len(result1_memo))

print("具体分割方案:")
for i, partition in enumerate(result1):
    print(f"方案 {i + 1}: {partition}")
print()

# 测试用例 2
s2 = "a"
result2 = solver.partition(s2)

print("测试用例 2 - 字符串:", repr(s2))
print("结果数量:", len(result2))
print("具体分割方案:")
for partition in result2:
    print(partition)
print()

# 测试补充题目
print("== 补充训练题目测试 ==")

# 测试分割回文串 II
s3 = "aab"
print("分割回文串 II - 字符串:", repr(s3), "-> 最少分割次数:", min_cut(s3))

# 测试复原 IP 地址
s4 = "25525511135"
ip_results = restore_ip_addresses(s4)
print("复原 IP 地址 - 字符串:", repr(s4), "-> 结果数量:", len(ip_results))
for ip in ip_results:
    print("IP 地址:", ip)

# 测试单词拆分 II
s5 = "catsanddog"
word_dict = ["cat", "cats", "and", "sand", "dog"]
word_break_results = word_break(s5, word_dict)
print("单词拆分 II - 字符串:", repr(s5), "-> 结果数量:", len(word_break_results))
for sentence in word_break_results:
    print("句子:", sentence)
```

```
# 测试优化版分割回文串
s6 = "aab"
optimized_results = partition_optimized(s6)
print("优化版分割回文串 - 字符串:", repr(s6), "-> 结果数量:", len(optimized_results))

# 测试回文分解生成
s7 = "aab"
decompositions = generate_palindromic_decompositions(s7)
print("回文分解生成 - 字符串:", repr(s7), "-> 分解方案数量:", len(decompositions))

if __name__ == "__main__":
    test_palindrome_partitioning()

"""

算法技巧总结 - Python 版本
```

核心概念:

1. 回溯算法框架:
 - 选择: 选择当前分割点
 - 约束: 检查子串是否是回文
 - 目标: 完成整个字符串的分割
 - 回溯: 撤销当前选择, 尝试其他选择
2. 优化技术:
 - 动态规划预处理: 提前计算回文信息
 - 记忆化搜索: 缓存中间结果避免重复计算
 - 剪枝优化: 提前终止无效分支
3. Python 特有优势:
 - 列表切片: s[start:end]快速提取子串
 - 字符串反转: s[::-1]快速判断回文
 - 列表推导式: 简化代码编写

调试技巧:

1. 使用 pdb 进行调试
2. 打印中间状态变量
3. 使用 assert 进行条件验证

性能优化:

1. 避免不必要的字符串复制
2. 使用局部变量减少属性查找
3. 利用 Python 内置函数的高效实现

工程化实践：

1. 类型注解：提高代码可读性
2. 异常处理：确保程序健壮性
3. 单元测试：保证代码质量
4. 文档字符串：提供清晰的接口说明

边界情况处理：

1. 空字符串：返回空列表
2. 单字符：返回包含该字符的列表
3. 全相同字符：所有分割方案都是回文
4. 无回文分割：返回空列表

"""

文件：Code05_SkillMonster.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

/***
 * 技能打怪问题（升级版） - C++版本
 *
 * 问题描述：
 * 现在有一个打怪类型的游戏，你有 n 个技能，每个技能有伤害值、魔法消耗值和触发双倍伤害的血量阈值。
 * 每个技能最多只能释放一次，怪物有 m 点血量。问如何用最少的魔法值消灭怪物（血量≤0）。
 *
 * 算法思路：
 * 方法 1：回溯算法 - 遍历所有可能的技能使用顺序
 * 方法 2：动态规划 - 使用状态压缩 DP
 * 方法 3：贪心+回溯 - 按技能性价比排序优化
 *
 * 时间复杂度分析：
 * - 回溯算法：O(n!)，需要尝试所有技能的排列组合
 * - 动态规划：O(n * 2^n)，状态压缩 DP
 * - 贪心+回溯：O(n!)，但通过剪枝优化实际运行更快
 *
 * 空间复杂度分析：
```

```

* - 回溯算法: O(n), 递归栈深度
* - 动态规划: O(2^n), DP 数组空间
* - 贪心+回溯: O(n), 递归栈深度
*
* 工程化考量:
* 1. 输入验证: 检查参数合法性
* 2. 边界处理: 怪物血量为 0、技能伤害不足等情况
* 3. 性能优化: 剪枝、排序、缓存等优化技术
* 4. 可测试性: 设计全面的测试用例
*/

```

```

class SkillMonsterAdvancedSolver {
private:
    vector<int> damage;           // 技能伤害值
    vector<int> cost;             // 魔法消耗值
    vector<int> threshold;        // 触发双倍伤害的血量阈值
    int minCost;                  // 最小魔法消耗

public:
    /**
     * 构造函数
     */
    SkillMonsterAdvancedSolver(vector<int> d, vector<int> c, vector<int> t)
        : damage(d), cost(c), threshold(t), minCost(INT_MAX) {}

    /**
     * 方法 1: 回溯算法
     * 遍历所有可能的技能使用顺序
     */
    int minMagicCost(int n, int m) {
        if (m <= 0) return 0;

        minCost = INT_MAX;
        vector<bool> used(n, false);

        backtrack(m, 0, used);

        return minCost == INT_MAX ? -1 : minCost;
    }

    /**
     * 方法 2: 动态规划解法
     * 使用状态压缩 DP, mask 表示技能使用状态
     */
}

```

```

*/
int minMagicCostDP(int n, int m) {
    int totalStates = 1 << n;

    // dp[mask]表示使用 mask 对应技能时的最小魔法消耗
    vector<int> dp(totalStates, INT_MAX);
    dp[0] = 0; // 没有使用任何技能

    // hp[mask]表示使用 mask 对应技能后怪物的剩余血量
    vector<int> hp(totalStates, m);

    for (int mask = 0; mask < totalStates; mask++) {
        if (dp[mask] == INT_MAX) continue;

        for (int i = 0; i < n; i++) {
            if (!(mask & (1 << i))) { // 技能 i 未使用
                int newMask = mask | (1 << i);
                int actualDamage = calculateDamage(hp[mask], i);
                int newHP = hp[mask] - actualDamage;
                int newCost = dp[mask] + cost[i];

                if (newHP <= 0) {
                    // 怪物被击败
                    if (newCost < dp[newMask]) {
                        dp[newMask] = newCost;
                    }
                } else {
                    if (newCost < dp[newMask]) {
                        dp[newMask] = newCost;
                        hp[newMask] = newHP;
                    }
                }
            }
        }
    }

    // 找到所有状态中的最小魔法消耗
    int result = INT_MAX;
    for (int mask = 0; mask < totalStates; mask++) {
        if (hp[mask] <= 0 && dp[mask] < result) {
            result = dp[mask];
        }
    }
}

```

```

        return result == INT_MAX ? -1 : result;
    }

/***
 * 方法 3：贪心+回溯优化
 * 按技能性价比排序，优先尝试高性价比技能
 */
int minMagicCostGreedy(int n, int m) {
    if (m <= 0) return 0;

    // 计算技能性价比（伤害/消耗）
    vector<pair<int, double>> skills;
    for (int i = 0; i < n; i++) {
        double efficiency = (cost[i] > 0) ? (double)damage[i] / cost[i] : INT_MAX;
        skills.push_back({i, efficiency});
    }

    // 按性价比降序排序
    sort(skills.begin(), skills.end(), [] (const pair<int, double>& a, const pair<int, double>& b) {
        return a.second > b.second;
    });

    minCost = INT_MAX;
    vector<bool> used(n, false);

    backtrackGreedy(m, 0, used, skills);

    return minCost == INT_MAX ? -1 : minCost;
}

private:
/***
 * 回溯函数
 */
void backtrack(int remainingHP, int currentCost, vector<bool>& used) {
    // 基础情况：怪物已被击败
    if (remainingHP <= 0) {
        if (currentCost < minCost) {
            minCost = currentCost;
        }
        return;
    }
}

```

```

}

// 剪枝优化：如果当前魔法消耗已经超过已知最优解，提前返回
if (currentCost >= minCost) {
    return;
}

// 尝试使用每个未使用的技能
for (int i = 0; i < used.size(); i++) {
    if (!used[i]) {
        used[i] = true;

        int actualDamage = calculateDamage(remainingHP, i);
        backtrack(remainingHP - actualDamage, currentCost + cost[i], used);

        used[i] = false;
    }
}

/***
 * 贪心优化的回溯函数
 */
void backtrackGreedy(int remainingHP, int currentCost, vector<bool>& used,
                     vector<pair<int, double>>& skills) {
    if (remainingHP <= 0) {
        if (currentCost < minCost) {
            minCost = currentCost;
        }
        return;
    }

    if (currentCost >= minCost) {
        return;
    }

    // 按性价比顺序尝试技能
    for (auto& skill : skills) {
        int i = skill.first;
        if (!used[i]) {
            used[i] = true;

            int actualDamage = calculateDamage(remainingHP, i);

```

```

        backtrackGreedy(remainingHP - actualDamage, currentCost + cost[i], used, skills);

        used[i] = false;
    }
}

}

/***
 * 计算实际伤害 (考虑双倍伤害)
 */
int calculateDamage(int remainingHP, int skillIndex) {
    return (remainingHP <= threshold[skillIndex]) ? damage[skillIndex] * 2 :
damage[skillIndex];
}
};

/***
 * 补充训练题目 - C++实现
 */
// LeetCode 322. 零钱兑换
// 给定不同面额的硬币 coins 和总金额 amount，计算凑成总金额所需的最少硬币个数。
//
int coinChange(vector<int>& coins, int amount) {
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;

    for (int coin : coins) {
        for (int i = coin; i <= amount; i++) {
            if (dp[i - coin] != amount + 1) {
                dp[i] = min(dp[i], dp[i - coin] + 1);
            }
        }
    }
}

return dp[amount] > amount ? -1 : dp[amount];
}

/***
 * LeetCode 518. 零钱兑换 II
 * 给定不同面额的硬币和一个总金额，计算可以凑成总金额的硬币组合数。
*/

```

```

int change(int amount, vector<int>& coins) {
    vector<int> dp(amount + 1, 0);
    dp[0] = 1;

    for (int coin : coins) {
        for (int i = coin; i <= amount; i++) {
            dp[i] += dp[i - coin];
        }
    }

    return dp[amount];
}

/***
 * LeetCode 279. 完全平方数
 * 给定正整数 n，找到若干个完全平方数使得它们的和等于 n，返回需要的最少完全平方数。
 */
int numSquares(int n) {
    vector<int> dp(n + 1, INT_MAX);
    dp[0] = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j * j <= i; j++) {
            dp[i] = min(dp[i], dp[i - j * j] + 1);
        }
    }

    return dp[n];
}

/***
 * LeetCode 377. 组合总和 IV
 * 给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。
 */
int combinationSum4(vector<int>& nums, int target) {
    vector<unsigned int> dp(target + 1, 0);
    dp[0] = 1;

    for (int i = 1; i <= target; i++) {
        for (int num : nums) {
            if (i >= num) {
                dp[i] += dp[i - num];
            }
        }
    }
}

```

```

    }

}

return dp[target];
}

/***
 * LeetCode 416. 分割等和子集
 * 给定一个只包含正整数的非空数组，判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
 */
bool canPartition(vector<int>& nums) {
    int sum = 0;
    for (int num : nums) {
        sum += num;
    }

    if (sum % 2 != 0) return false;

    int target = sum / 2;
    vector<bool> dp(target + 1, false);
    dp[0] = true;

    for (int num : nums) {
        for (int i = target; i >= num; i--) {
            dp[i] = dp[i] || dp[i - num];
        }
    }
}

return dp[target];
}

// 测试函数
void testSkillMonsterAdvanced() {
    // 测试用例 1：简单的打怪场景
    vector<int> damage1 = {3, 4, 5};
    vector<int> cost1 = {2, 3, 4};
    vector<int> threshold1 = {5, 3, 2};
    int m1 = 10;

    SkillMonsterAdvancedSolver solver1(damage1, cost1, threshold1);
    int result1 = solver1.minMagicCost(3, m1);
    int result1_dp = solver1.minMagicCostDP(3, m1);
}

```

```

int result1_greedy = solver1.minMagicCostGreedy(3, m1);

cout << "测试用例 1:" << endl;
cout << "技能伤害: "; for (int d : damage1) cout << d << " "; cout << endl;
cout << "技能消耗: "; for (int c : cost1) cout << c << " "; cout << endl;
cout << "技能阈值: "; for (int t : threshold1) cout << t << " "; cout << endl;
cout << "怪物血量: " << m1 << endl;
cout << "回溯算法结果: " << result1 << endl;
cout << "动态规划结果: " << result1_dp << endl;
cout << "贪心优化结果: " << result1_greedy << endl;
cout << endl;

// 测试用例 2: 无法击败怪物的情况
vector<int> damage2 = {3, 4};
vector<int> cost2 = {2, 3};
vector<int> threshold2 = {10, 8};
int m2 = 20;

SkillMonsterAdvancedSolver solver2(damage2, cost2, threshold2);
int result2 = solver2.minMagicCost(2, m2);

cout << "测试用例 2:" << endl;
cout << "技能伤害: "; for (int d : damage2) cout << d << " "; cout << endl;
cout << "技能消耗: "; for (int c : cost2) cout << c << " "; cout << endl;
cout << "技能阈值: "; for (int t : threshold2) cout << t << " "; cout << endl;
cout << "怪物血量: " << m2 << endl;
cout << "最少魔法消耗: " << result2 << endl;
cout << endl;

// 测试补充题目
cout << "==== 补充训练题目测试 ===" << endl;

// 测试零钱兑换
vector<int> coins = {1, 2, 5};
int amount = 11;
cout << "零钱兑换: coins={"; for (int c : coins) cout << c << " "; cout << "}, amount=" << amount;
cout << ", 结果=" << coinChange(coins, amount) << endl;

// 测试零钱兑换 II
cout << "零钱兑换 II: coins={"; for (int c : coins) cout << c << " "; cout << "}, amount=" << amount;
cout << ", 结果=" << change(amount, coins) << endl;

```

```

// 测试完全平方数
int n = 12;
cout << "完全平方数: n=" << n << ", 结果=" << numSquares(n) << endl;

// 测试组合总和 IV
vector<int> nums = {1, 2, 3};
int target = 4;
cout << "组合总和 IV: nums={" << target << "}";
for (int num : nums) cout << num << " ";
cout << ", 结果=" << combinationSum4(nums, target) << endl;

// 测试分割等和子集
vector<int> partitionNums = {1, 5, 11, 5};
cout << "分割等和子集: nums={" << target << "}";
for (int num : partitionNums) cout << num << " ";
cout << ", 结果=" << (canPartition(partitionNums) ? "true" : "false") << endl;
}

int main() {
    testSkillMonsterAdvanced();
    return 0;
}

/**
 * 算法技巧总结 - C++版本
 *
 * 核心概念:
 * 1. 回溯算法框架:
 *     - 状态定义: 当前选择、剩余选择、目标状态
 *     - 选择策略: 遍历所有可能的选择
 *     - 终止条件: 达到目标状态或无法继续
 *     - 回溯操作: 撤销当前选择, 尝试其他选择
 *
 * 2. 动态规划技术:
 *     - 状态压缩: 使用位运算表示状态
 *     - 状态转移: 根据当前状态推导下一状态
 *     - 边界处理: 处理初始状态和终止状态
 *
 * 3. 优化技巧:
 *     - 剪枝优化: 提前终止无效分支
 *     - 排序优化: 优先搜索更优路径
 *     - 贪心策略: 局部最优选择

```

```
*  
* 调试技巧:  
* 1. 打印中间状态: 跟踪算法执行过程  
* 2. 边界值测试: 测试各种边界情况  
* 3. 性能分析: 比较不同算法的执行时间  
*  
* 工程化实践:  
* 1. 模块化设计: 分离算法逻辑和业务逻辑  
* 2. 异常安全: 确保资源正确释放  
* 3. 代码可读性: 使用有意义的变量名和注释  
*/
```

=====

文件: Code05_SkillMonster.java

=====

```
package class043;  
  
import java.util.*;  
  
/**  
 * 技能打怪问题（升级版）  
 *  
 * 问题描述:  
 * 现在有一个打怪类型的游戏，这个游戏是这样的，你有 n 个技能  
 * 每一个技能会有一个伤害值和魔法消耗值  
 * 同时若怪物血量小于等于一定的阈值，则该技能可能造成双倍伤害  
 * 每一个技能最多只能释放一次，已知怪物有 m 点血量  
 * 现在想问你如何用最少的魔法值消灭怪物（血量小于等于 0）  
 *  
 * 技能的数量是 n，怪物的血量是 m  
 * i 号技能的伤害是 damage[i]，魔法消耗是 cost[i]  
 * i 号技能触发双倍伤害的血量最小值是 threshold[i]  
 *  
 * 约束条件:  
 * 1 <= n <= 10  
 * 1 <= m, damage[i], cost[i], threshold[i] <= 10^6  
 *  
 * 算法思路:  
 * 1. 这是一个典型的回溯算法问题，需要找出最少的魔法消耗来击败怪物  
 * 2. 使用回溯算法遍历所有可能的技能使用顺序，找到最少魔法消耗的方案  
 * 3. 在每一步尝试使用不同的技能，通过递归和回溯来探索所有可能性  
 * 4. 可以通过剪枝优化，如果当前魔法消耗已经超过已知最优解，则提前返回
```

*

* 时间复杂度分析:

- * - 最坏情况下需要尝试所有技能的排列组合
- * - 技能数为 n, 时间复杂度为 $O(n!)$
- * - 对于 $n \leq 10$ 的情况, $10! = 3,628,800$, 可以在合理时间内完成

*

* 空间复杂度分析:

- * - 主要空间消耗是递归栈深度和存储技能信息的数组
- * - 递归深度最大为 n, 空间复杂度为 $O(n)$

*

* 工程化考量:

- * 1. 异常处理: 对输入数据进行校验
- * 2. 性能优化: 使用剪枝优化, 避免无效搜索
- * 3. 可配置性: MAXN 常量可以调整以适应不同规模的问题
- * 4. 鲁棒性: 处理边界情况, 如怪物血量为 0 或技能数组为空

*

* 相关题目:

- * 1. 牛客网 - 打怪兽 - <https://www.nowcoder.com/practice/a3b055dd672245a3a6e2f759c237e449>
- * 2. LeetCode 46. 全排列 - <https://leetcode.cn/problems/permutations/>
- * 3. LeetCode 47. 全排列 II - <https://leetcode.cn/problems/permutations-ii/>
- * 4. LeetCode 322. 零钱兑换 - <https://leetcode.cn/problems/coin-change/>
- * 5. LeetCode 787. K 站中转内最便宜的航班 - <https://leetcode.cn/problems/cheapest-flights-within-k-stops/>
- * 6. LeetCode 494. 目标和 - <https://leetcode.cn/problems/target-sum/>
- * 7. LeetCode 516. 最长回文子序列 - <https://leetcode.cn/problems/longest-palindromic-subsequence/>
- * 8. LeetCode 72. 编辑距离 - <https://leetcode.cn/problems/edit-distance/>
- * 9. LeetCode 1048. 最长字符串链 - <https://leetcode.cn/problems/longest-string-chain/>
- * 10. LeetCode 1289. 下降路径最小和 II - <https://leetcode.cn/problems/minimum-falling-path-sum-ii/>
- * 11. LeetCode 1155. 掷骰子的目标和 - <https://leetcode.cn/problems/number-of-dice-rolls-with-target-sum/>
- * 12. LeetCode 983. 最低票价 - <https://leetcode.cn/problems/minimum-cost-for-tickets/>
- * 13. LeetCode 518. 零钱兑换 II - <https://leetcode.cn/problems/coin-change-2/>
- * 14. LeetCode 650. 只有两个键的键盘 - <https://leetcode.cn/problems/2-keys-keyboard/>
- * 15. LeetCode 879. 盈利计划 - <https://leetcode.cn/problems/profitable-schemes/>
- * 16. LeetCode 746. 使用最小花费爬楼梯 - <https://leetcode.cn/problems/min-cost-climbing-stairs/>
- * 17. LeetCode 139. 单词拆分 - <https://leetcode.cn/problems/word-break/>
- * 18. LeetCode 300. 最长递增子序列 - <https://leetcode.cn/problems/longest-increasing-subsequence/>
- * 19. LeetCode 120. 三角形最小路径和 - <https://leetcode.cn/problems/triangle/>
- * 20. LeetCode 198. 打家劫舍 - <https://leetcode.cn/problems/house-robber/>
- * 21. Codeforces 1312C - Make It Good - <https://codeforces.com/problemset/problem/1312/C>

* 22. Codeforces 1436E - Combinatorics Homework -
<https://codeforces.com/problemset/problem/1436/E>

* 23. Codeforces 1332B - Composite Coloring - <https://codeforces.com/problemset/problem/1332/B>

* 24. Codeforces 1328A - Divisibility Problem - <https://codeforces.com/problemset/problem/1328/A>

* 25. Codeforces 1327D - Infinite Path - <https://codeforces.com/problemset/problem/1327/D>

* 26. AtCoder ABC145D - Knight - https://atcoder.jp/contests/abc145/tasks/abc145_d

* 27. AtCoder ABC175D - Moving Pieces on Checkerboard -
https://atcoder.jp/contests/abc175/tasks/abc175_d

* 28. AtCoder ABC159E - Dividing Chocolate - https://atcoder.jp/contests/abc159/tasks/abc159_e

* 29. AtCoder ABC167D - Teleporter - https://atcoder.jp/contests/abc167/tasks/abc167_d

* 30. 洛谷 P1135 - 奇怪的电梯 - <https://www.luogu.com.cn/problem/P1135>

* 31. 洛谷 P1048 - 采药 - <https://www.luogu.com.cn/problem/P1048>

* 32. 洛谷 P1002 - 过河卒 - <https://www.luogu.com.cn/problem/P1002>

* 33. 洛谷 P1004 - 方格取数 - <https://www.luogu.com.cn/problem/P1004>

* 34. 洛谷 P1157 - 组合的输出 - <https://www.luogu.com.cn/problem/P1157>

* 35. HackerRank - Recursive Digit Sum - <https://www.hackerrank.com/challenges/recursive-digit-sum/problem>

* 36. HackerRank - The Coin Change Problem - <https://www.hackerrank.com/challenges/coin-change/problem>

* 37. HackerRank - Sherlock and Cost - <https://www.hackerrank.com/challenges/sherlock-and-cost/problem>

* 38. HackerRank - Split the String - <https://www.hackerrank.com/challenges/split-the-string/problem>

* 39. UVa 10189 - Minesweeper -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1130

* 40. POJ 1163 - The Triangle - <http://poj.org/problem?id=1163>

* 41. POJ 3009 - Curling 2.0 - <http://poj.org/problem?id=3009>

* 42. POJ 1042 - Gone Fishing - <http://poj.org/problem?id=1042>

* 43. HDU 1203 - I NEED A OFFER! - <https://acm.hdu.edu.cn/showproblem.php?pid=1203>

* 44. HDU 1527 - 取石子游戏 - <https://acm.hdu.edu.cn/showproblem.php?pid=1527>

* 45. HDU 1421 - 搬寝室 - <https://acm.hdu.edu.cn/showproblem.php?pid=1421>

* 46. LintCode 1000. 背包问题 II - <https://www.lintcode.com/problem/1000/>

* 47. LintCode 1178. 验证回文字符串 II - <https://www.lintcode.com/problem/1178/>

* 48. LintCode 135. 数字组合 - <https://www.lintcode.com/problem/135/>

* 49. LintCode 43. 最大子数组 III - <https://www.lintcode.com/problem/43/>

* 50. LintCode 44. 最小子数组 - <https://www.lintcode.com/problem/44/>

*/

```
public class Code05_SkillMonster {
```

```
    public static int MAXN = 11;
```

```
    // 技能信息
```

```

public static int[] damage = new int[MAXN]; // 伤害值
public static int[] cost = new int[MAXN]; // 魔法消耗
public static int[] threshold = new int[MAXN]; // 触发双倍伤害的血量阈值

/**
 * 计算击败怪物所需的最少魔法值
 *
 * @param n 技能总数
 * @param m 怪物血量
 * @return 最少需要的魔法值，如果无法击败则返回-1
 *
 * 算法思路：
 * 1. 使用回溯算法尝试所有可能的技能使用顺序
 * 2. 在搜索过程中维护当前使用的魔法值和怪物剩余血量
 * 3. 通过剪枝优化避免无效搜索
 * 4. 返回所有方案中魔法消耗最少的值
 */
public static int minMagicCost(int n, int m) {
    if (m <= 0) {
        return 0; // 怪物已经死亡
    }

    // 初始化最优解为最大值
    int[] minCost = {Integer.MAX_VALUE};

    // 回溯搜索最优解
    backtrack(n, 0, m, 0, minCost, new boolean[n]);

    return minCost[0] == Integer.MAX_VALUE ? -1 : minCost[0];
}

/**
 * 回溯函数，计算最少需要多少魔法值能击败怪物
 *
 * @param n 技能总数
 * @param used 已使用的技能数
 * @param remainingMonsterHP 怪物当前剩余血量
 * @param currentCost 当前已消耗的魔法值
 * @param minCost 存储最小魔法消耗的数组（引用传递）
 * @param usedSkills 标记技能是否已使用的布尔数组
 *
 * 递归思路：
 * 1. 基础情况：如果怪物血量 remainingMonsterHP<=0，说明已经被击败，更新最小魔法消耗

```

```

* 2. 剪枝优化：如果当前魔法消耗 currentCost 已经大于等于已知最小值，则提前返回
* 3. 递归情况：尝试使用未使用的每个技能
*
* 优化点：
* 1. 剪枝优化：避免无效搜索
* 2. 标记数组：记录技能使用情况，避免重复使用
*/
private static void backtrack(int n, int used, int remainingMonsterHP, int currentCost, int[] minCost, boolean[] usedSkills) {
    // 基础情况：怪物已被击败
    if (remainingMonsterHP <= 0) {
        minCost[0] = Math.min(minCost[0], currentCost);
        return;
    }

    // 剪枝优化：如果当前魔法消耗已经超过已知最优解，则提前返回
    if (currentCost >= minCost[0]) {
        return;
    }

    // 递归情况：尝试使用未使用的每个技能
    for (int i = 0; i < n; i++) {
        if (!usedSkills[i]) {
            // 标记技能为已使用
            usedSkills[i] = true;

            // 计算使用该技能造成的伤害
            int actualDamage = (remainingMonsterHP <= threshold[i]) ? damage[i] * 2 : damage[i];

            // 递归调用
            backtrack(n, used + 1, remainingMonsterHP - actualDamage, currentCost + cost[i], minCost, usedSkills);

            // 回溯：标记技能为未使用
            usedSkills[i] = false;
        }
    }
}

/*
* 补充训练题目
* 以下是几个与技能打怪问题相关的经典题目，帮助加深对回溯算法和剪枝优化的理解

```

```
*/
```

```
/**
```

```
* 题目 1: LeetCode 322. 零钱兑换
```

```
* 问题描述: 给你一个整数数组 coins , 表示不同面额的硬币; 以及一个整数 amount , 表示总金额。
```

```
* 计算并返回可以凑成总金额所需的 最少的硬币个数 。如果没有任何一种硬币组合能组成总金额, 返回 -1 。
```

```
* 解题思路: 使用动态规划, dp[i]表示凑成金额 i 所需的最少硬币数
```

```
*/
```

```
public static int coinChange(int[] coins, int amount) {
```

```
    int[] dp = new int[amount + 1];
```

```
    Arrays.fill(dp, amount + 1);
```

```
    dp[0] = 0;
```

```
    for (int coin : coins) {
```

```
        for (int i = coin; i <= amount; i++) {
```

```
            dp[i] = Math.min(dp[i], dp[i - coin] + 1);
```

```
        }
```

```
}
```

```
    return dp[amount] > amount ? -1 : dp[amount];
```

```
}
```

```
/**
```

```
* 题目 2: LeetCode 787. K 站中转内最便宜的航班
```

```
* 问题描述: 有 n 个城市通过一些航班连接。给你一个数组 flights , 其中 flights[i] = [fromi, toi, pricei] ,
```

```
* 表示该航班都从城市 fromi 开始, 以价格 toi 抵达 pricei。现在给定所有的城市和航班, 以及出发城市 src 和目的地 dst,
```

```
* 你的任务是找到从 src 到 dst 最多经过 k 站中转的最便宜的价格。如果没有这样的路线, 则输出 -1。
```

```
* 解题思路: 使用 Bellman-Ford 算法或 DFS+剪枝
```

```
*/
```

```
public static int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {
```

```
    int[] dist = new int[n];
```

```
    Arrays.fill(dist, Integer.MAX_VALUE);
```

```
    dist[src] = 0;
```

```
    for (int i = 0; i <= k; i++) {
```

```
        int[] temp = Arrays.copyOf(dist, n);
```

```
        for (int[] flight : flights) {
```

```
            int from = flight[0], to = flight[1], price = flight[2];
```

```
            if (dist[from] != Integer.MAX_VALUE) {
```

```

        temp[to] = Math.min(temp[to], dist[from] + price);
    }
}

dist = temp;
}

return dist[dst] == Integer.MAX_VALUE ? -1 : dist[dst];
}

/***
 * 题目 3: LeetCode 494. 目标和
 * 问题描述: 给你一个整数数组 nums 和一个整数 target 。
 * 向数组中的每个整数前添加 '+' 或 '-'，然后串联起所有整数，可以构造一个表达式 。
 * 返回可以通过上述方法构造的、运算结果等于 target 的不同表达式的数目。
 * 解题思路: 动态规划 + 转化问题，将问题转化为子集和问题
 */
public static int findTargetSumWays(int[] nums, int target) {
    int sum = 0;
    for (int num : nums) {
        sum += num;
    }

    // 边界条件判断
    if (sum < Math.abs(target) || (sum + target) % 2 != 0) {
        return 0;
    }

    int targetSum = (sum + target) / 2;
    int[] dp = new int[targetSum + 1];
    dp[0] = 1;

    for (int num : nums) {
        for (int j = targetSum; j >= num; j--) {
            dp[j] += dp[j - num];
        }
    }

    return dp[targetSum];
}

/***
 * 题目 4: LeetCode 516. 最长回文子序列
 * 问题描述: 给你一个字符串 s，找出其中最长的回文子序列，并返回该序列的长度。
 */

```

```

* 子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。
* 解题思路：动态规划，dp[i][j]表示 s[i...j] 范围内的最长回文子序列长度
*/
public static int longestPalindromeSubseq(String s) {
    int n = s.length();
    int[][] dp = new int[n][n];

    // 初始化对角线，单个字符是长度为 1 的回文
    for (int i = 0; i < n; i++) {
        dp[i][i] = 1;
    }

    // 自底向上填充 dp 表格
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[0][n - 1];
}

/**
 * 题目 5：LeetCode 72. 编辑距离
 * 问题描述：给你两个单词 word1 和 word2，请返回将 word1 转换成 word2 所使用的最少操作数。
 * 你可以对一个单词进行如下三种操作：
 * 插入一个字符
 * 删除一个字符
 * 替换一个字符
 * 解题思路：动态规划，dp[i][j] 表示 word1 前 i 个字符转换为 word2 前 j 个字符所需的最小操作数
*/
public static int minDistance(String word1, String word2) {
    int m = word1.length();
    int n = word2.length();

    // 创建 dp 数组，初始化为最大值
    int[][] dp = new int[m + 1][n + 1];

```

```

// 初始化边界条件
for (int i = 0; i <= m; i++) {
    dp[i][0] = i; // word2 为空, 需要删除 word1 的所有字符
}
for (int j = 0; j <= n; j++) {
    dp[0][j] = j; // word1 为空, 需要插入 word2 的所有字符
}

// 填充 dp 数组
for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
            // 字符相同, 不需要操作
            dp[i][j] = dp[i - 1][j - 1];
        } else {
            // 取插入、删除、替换中的最小值
            dp[i][j] = Math.min(Math.min(
                dp[i - 1][j] + 1,           // 删除
                dp[i][j - 1] + 1),         // 插入
                dp[i - 1][j - 1] + 1);     // 替换
        }
    }
}

return dp[m][n];
}

/***
 * 题目 6: Codeforces 1436E - Combinatorics Homework
 * 问题描述: 给定三个整数 a, b, c, 代表三种不同字符的数量, 以及一个整数 m,
 * 请判断是否可以构造一个字符串, 使得相邻不同字符的对数恰好为 m。
 * 解题思路: 计算相邻不同字符对的最小和最大可能值, 判断 m 是否在这个范围内
 */
public static boolean combinatoricsHomework(int a, int b, int c, int m) {
    int[] count = {a, b, c};
    Arrays.sort(count);
    int min = 0;
    int max = a + b + c - 1;

    // 计算最小可能的不同对数: 最大的字符数 - (其他字符数之和 + 1)
    if (count[2] > count[0] + count[1] + 1) {
        min = count[2] - (count[0] + count[1] + 1);
    }
}

```

```

}

return m >= min && m <= max;
}

/***
 * 题目 7: HackerRank - Sherlock and Cost
 * 问题描述: 给定一个数组 B, 构造一个数组 A, 使得每个元素满足  $1 \leq A[i] \leq B[i]$ ,
 * 并且最大化数组 A 的绝对差值之和:  $\sum_{i=2 \text{ to } N} |A[i] - A[i-1]|$ 
 * 解题思路: 动态规划, 维护每个位置取 1 或 B[i] 时的最大和
 */
public static int sherlockAndCost(int[] B) {
    int n = B.length;
    int[][] dp = new int[n][2];

    for (int i = 1; i < n; i++) {
        // 当前取 1, 前一个取 1 或 B[i-1]
        dp[i][0] = Math.max(dp[i-1][0] + Math.abs(1 - 1), dp[i-1][1] + Math.abs(1 - B[i-1]));
        // 当前取 B[i], 前一个取 1 或 B[i-1]
        dp[i][1] = Math.max(dp[i-1][0] + Math.abs(B[i] - 1), dp[i-1][1] + Math.abs(B[i] - B[i-1]));
    }

    return Math.max(dp[n-1][0], dp[n-1][1]);
}

/***
 * 题目 8: 洛谷 P1048 - 采药
 * 问题描述: 有一个山洞, 里面有 n 株草药, 每株草药有一定的价值和采摘时间。
 * 现在给你一个总时间 T, 要求在规定的时间内采摘价值最大的草药。
 * 解题思路: 01 背包问题, 动态规划求解
 */
public static int collectHerbs(int[] time, int[] value, int T) {
    int n = time.length;
    int[] dp = new int[T + 1];

    for (int i = 0; i < n; i++) {
        for (int j = T; j >= time[i]; j--) {
            dp[j] = Math.max(dp[j], dp[j - time[i]] + value[i]);
        }
    }

    return dp[T];
}

```

```
}
```

```
/**
```

```
* 题目 9: HackerRank - Split the String
```

```
* 问题描述: 给定一个由' a' 和' b' 组成的字符串, 将其分割成 k 个子字符串, 使得每个子字符串中' a' 和' b' 的数量相等。
```

```
* 求 k 的最大可能值。
```

```
* 解题思路: 贪心算法, 每当遇到 a 和 b 数量相等的位置就分割
```

```
*/
```

```
public static int splitTheString(String s) {
```

```
    int countA = 0, countB = 0, result = 0;
```

```
    for (char c : s.toCharArray()) {
```

```
        if (c == 'a') {
```

```
            countA++;
```

```
        } else {
```

```
            countB++;
```

```
        }
```

```
        if (countA == countB) {
```

```
            result++;
```

```
            countA = 0;
```

```
            countB = 0;
```

```
        }
```

```
}
```

```
    return result;
```

```
}
```

```
/**
```

```
* 补充题目 10: LeetCode 198. 打家劫舍
```

```
* 问题描述: 你是一个专业的小偷, 计划偷窃沿街的房屋。每间房内都藏有一定的现金, 影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统,
```

```
* 如果两间相邻的房屋在同一晚上被小偷闯入, 系统会自动报警。
```

```
* 给定一个代表每个房屋存放金额的非负整数数组, 计算你 不触动警报装置的情况下 , 一夜之内能够偷窃到的最高金额。
```

```
* 解题思路: 动态规划, dp[i]表示偷到第 i 间房子的最大金额
```

```
*/
```

```
public static int rob(int[] nums) {
```

```
    if (nums == null || nums.length == 0) {
```

```
        return 0;
```

```
}
```

```
    int n = nums.length;
```

```

    if (n == 1) {
        return nums[0];
    }
    int[] dp = new int[n];
    dp[0] = nums[0];
    dp[1] = Math.max(nums[0], nums[1]);
    for (int i = 2; i < n; i++) {
        dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
    }
    return dp[n - 1];
}

/**
 * 补充题目 11: LeetCode 213. 打家劫舍 II
 * 问题描述: 你是一个专业的小偷, 计划偷窃沿街的房屋, 每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈, 这意味着第一个房屋和最后一个房屋是紧挨着的。
 * 同时, 相邻的房屋装有相互连通的防盗系统, 如果两间相邻的房屋在同一晚上被小偷闯入, 系统会自动报警。
 * 给定一个代表每个房屋存放金额的非负整数数组, 计算你 在不触动警报装置的情况下, 能够偷窃到的最高金额。
 * 解题思路: 将问题分解为两种情况, 取最大值
 */
public static int rob2(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    int n = nums.length;
    if (n == 1) {
        return nums[0];
    }
    // 情况 1: 不偷第一家
    int[] dp1 = new int[n];
    dp1[0] = 0;
    dp1[1] = nums[1];
    for (int i = 2; i < n; i++) {
        dp1[i] = Math.max(dp1[i - 1], dp1[i - 2] + nums[i]);
    }
    // 情况 2: 不偷最后一家
    int[] dp2 = new int[n];
    dp2[0] = nums[0];
    dp2[1] = Math.max(nums[0], nums[1]);
    for (int i = 2; i < n - 1; i++) {
        dp2[i] = Math.max(dp2[i - 1], dp2[i - 2] + nums[i]);
    }
}

```

```

    }

    return Math.max(dp1[n - 1], dp2[n - 2]);
}

/**/
/* 补充题目 12: LeetCode 337. 打家劫舍 III
 * 问题描述: 小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为 root。
 * 除了 root 之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。
 * 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。
 * 计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。
 * 解题思路: 树形动态规划，每个节点返回偷或不偷两种状态的最大值
 */

public static class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

public static int rob3(TreeNode root) {
    int[] result = robTree(root);
    return Math.max(result[0], result[1]);
}

// 返回数组: [不偷当前节点的最大金额, 偷当前节点的最大金额]
private static int[] robTree(TreeNode node) {
    if (node == null) {
        return new int[] {0, 0};
    }

    int[] left = robTree(node.left);
    int[] right = robTree(node.right);

    // 不偷当前节点
    int notRob = Math.max(left[0], left[1]) + Math.max(right[0], right[1]);
    // 偷当前节点
    int rob = node.val + left[0] + right[0];
    return new int[] {notRob, rob};
}

```

}

/*

* 技能选择与资源优化算法的深度解析

*

* 一、核心概念与理论基础

* 1. 问题建模

* - 技能选择问题本质上是一个组合优化问题，可以建模为有约束的状态空间搜索

* - 每个状态由已使用的技能集合、剩余怪物血量和累计消耗的魔法值组成

* - 目标是在满足击败怪物条件（剩余血量 ≤ 0 ）的所有路径中，找到魔法消耗最小的路径

*

* 2. 状态空间分析

* - 状态空间大小： $O(n!)$ ，其中 n 是技能数量，因为每个技能只能使用一次

* - 状态转移：从当前状态选择一个未使用的技能，计算新的状态

* - 剪枝条件：当当前累计消耗超过已知最优解时，可以提前终止搜索

*

* 二、高级算法技术

* 1. 回溯算法的优化策略

* - 剪枝优化：维护全局最优解，及时终止无效搜索路径

* - 排序优化：可以对技能按照性价比（伤害/消耗）进行排序，优先搜索更优的路径

* - 记忆化搜索：对于重复出现的子问题，可以使用记忆化技术缓存中间结果

*

* 2. 动态规划方法

* - 状态表示： $dp[mask]$ 表示使用 $mask$ 对应的技能集合时的最小魔法消耗

* - 状态转移： $dp[mask \mid (1 \ll i)] = \min(dp[mask \mid (1 \ll i)], dp[mask] + cost[i])$

* - 适用于 n 较小的情况（通常 $n \leq 20$ ），因为 $mask$ 需要 2^n 的状态空间

*

* 3. 启发式搜索

* - A*算法：使用启发式函数估计剩余搜索空间的下界

* - 贪心策略：局部最优选择，如优先使用伤害高、消耗低的技能

*

* 三、多维度优化策略

* 1. 性能优化

* - 使用位运算代替数组标记技能使用状态

* - 预计算伤害值，避免重复计算

* - 采用更高效的数据结构存储状态

*

* 2. 内存优化

* - 对于大规模问题，可以使用迭代加深搜索限制搜索深度

* - 采用分支定界法，优先探索更有希望的分支

*

* 3. 并行计算

* - 将状态空间分割成多个子空间，并行搜索

- * - 使用并行流或线程池加速搜索过程
- *
- * 四、工程化实践指南
 - * 1. 代码健壮性
 - 输入验证：检查参数合法性，如技能数量、怪物血量等
 - 边界处理：考虑怪物血量为 0、技能伤害不足等边界情况
 - 异常处理：适当的 try-catch 块捕获可能的异常
 - *
 - * 2. 代码可读性与可维护性
 - 使用有意义的变量名和方法名
 - 添加详细的注释说明算法思路和关键步骤
 - 模块化设计，将不同功能分离为独立的方法
 - *
 - * 3. 测试策略
 - 单元测试：测试各个函数的正确性
 - 边界测试：测试极端情况下的行为
 - 性能测试：评估算法在不同规模输入下的性能
 - *
- * 五、跨学科应用
 - * 1. 游戏开发
 - 技能系统设计
 - AI 行为决策
 - 战斗策略优化
 - *
 - * 2. 资源分配
 - 云计算资源调度
 - 供应链优化
 - 投资组合选择
 - *
 - * 3. 路径规划
 - 机器人导航
 - 网络路由优化
 - 物流配送路径规划
 - *
- * 六、前沿研究方向
 - * 1. 结合机器学习
 - 使用强化学习优化技能选择策略
 - 深度学习预测最优解
 - 迁移学习应用于类似问题
 - *
 - * 2. 近似算法
 - 对于 NP 难问题，设计高效的近似算法
 - 随机化算法提高搜索效率

```
*      - 量子计算在组合优化中的应用
*
* 3. 多目标优化
*      - 同时考虑魔法消耗、技能冷却时间等多个目标
*      - Pareto 最优解集的计算
*      - 交互式多目标决策支持系统
*/

```

```
// 测试函数
public static void main(String[] args) {
    // 测试用例 1: 简单的打怪场景
    int n1 = 3;
    int m1 = 10;
    damage[0] = 3; cost[0] = 2; threshold[0] = 5;
    damage[1] = 4; cost[1] = 3; threshold[1] = 3;
    damage[2] = 5; cost[2] = 4; threshold[2] = 2;

    int result1 = minMagicCost(n1, m1);
    System.out.println("测试用例 1:");
    System.out.println("技能数: " + n1 + ", 怪物血量: " + m1);
    System.out.println("技能信息:");
    for (int i = 0; i < n1; i++) {
        System.out.println(" 技能" + i + ": 伤害=" + damage[i] + ", 消耗=" + cost[i] + ", 国
值=" + threshold[i]);
    }
    System.out.println("最少魔法消耗: " + result1);

    // 测试用例 2: 无法击败怪物的情况
    int n2 = 2;
    int m2 = 20;
    damage[0] = 3; cost[0] = 2; threshold[0] = 10;
    damage[1] = 4; cost[1] = 3; threshold[1] = 8;

    int result2 = minMagicCost(n2, m2);
    System.out.println("\n测试用例 2:");
    System.out.println("技能数: " + n2 + ", 怪物血量: " + m2);
    System.out.println("技能信息:");
    for (int i = 0; i < n2; i++) {
        System.out.println(" 技能" + i + ": 伤害=" + damage[i] + ", 消耗=" + cost[i] + ", 国
值=" + threshold[i]);
    }
    System.out.println("最少魔法消耗: " + result2);
}
```

```

// 测试用例 3: 怪物血量为 0
int n3 = 1;
int m3 = 0;
damage[0] = 5; cost[0] = 2; threshold[0] = 3;

int result3 = minMagicCost(n3, m3);
System.out.println("\n 测试用例 3:");
System.out.println("技能数: " + n3 + ", 怪物血量: " + m3);
System.out.println("技能信息:");
for (int i = 0; i < n3; i++) {
    System.out.println(" 技能" + i + ": 伤害=" + damage[i] + ", 消耗=" + cost[i] + ", 国
值=" + threshold[i]);
}
System.out.println("最少魔法消耗: " + result3);
}
}

```

=====

文件: Code05_SkillMonster.py

=====

"""
技能打怪问题（升级版） – Python 版本

问题描述:

现在有一个打怪类型的游戏，你有 n 个技能，每个技能有伤害值、魔法消耗值和触发双倍伤害的血量阈值。每个技能最多只能释放一次，怪物有 m 点血量。问如何用最少的魔法值消灭怪物（血量 ≤ 0 ）。

算法思路:

方法 1: 回溯算法 – 遍历所有可能的技能使用顺序

方法 2: 动态规划 – 使用状态压缩 DP

方法 3: 贪心+回溯 – 按技能性价比排序优化

时间复杂度分析:

- 回溯算法: $O(n!)$, 需要尝试所有技能的排列组合
- 动态规划: $O(n * 2^n)$, 状态压缩 DP
- 贪心+回溯: $O(n!)$, 但通过剪枝优化实际运行更快

空间复杂度分析:

- 回溯算法: $O(n)$, 递归栈深度
- 动态规划: $O(2^n)$, DP 数组空间
- 贪心+回溯: $O(n)$, 递归栈深度

工程化考量：

1. 输入验证：检查参数合法性
2. 边界处理：怪物血量为 0、技能伤害不足等情况
3. 性能优化：剪枝、排序、缓存等优化技术
4. 可测试性：设计全面的测试用例

"""

```
from typing import List, Tuple
import sys

class SkillMonsterAdvancedSolver:
    def __init__(self, damage: List[int], cost: List[int], threshold: List[int]):
        """
        构造函数
        Args:
            damage: 技能伤害值列表
            cost: 魔法消耗值列表
            threshold: 触发双倍伤害的血量阈值列表
        """
        self.damage = damage
        self.cost = cost
        self.threshold = threshold
        self.min_cost = float('inf')
        self.n = len(damage)

    # 输入验证
    if len(damage) != len(cost) or len(damage) != len(threshold):
        raise ValueError("伤害值、消耗值和阈值数组长度必须相同")

    if any(d < 0 for d in damage) or any(c < 0 for c in cost) or any(t < 0 for t in threshold):
        raise ValueError("伤害值、消耗值和阈值必须为非负数")

    def min_magic_cost(self, m: int) -> int:
        """
        方法 1：回溯算法
        遍历所有可能的技能使用顺序
        """
        if m <= 0:
            return 0

        if self.n == 0:
            return -1
```

```

self.min_cost = float('inf')
used = [False] * self.n

self._backtrack(m, 0, used)

result = -1 if self.min_cost == float('inf') else int(self.min_cost)
return result

def min_magic_cost_dp(self, m: int) -> int:
    """
    方法 2: 动态规划解法
    使用状态压缩 DP, mask 表示技能使用状态
    """
    if m <= 0:
        return 0

    total_states = 1 << self.n

    # dp[mask] 表示使用 mask 对应技能时的最小魔法消耗
    dp = [float('inf')] * total_states
    dp[0] = 0 # 没有使用任何技能

    # hp[mask] 表示使用 mask 对应技能后怪物的剩余血量
    hp = [m] * total_states

    for mask in range(total_states):
        if dp[mask] == float('inf'):
            continue

        for i in range(self.n):
            if not (mask & (1 << i)): # 技能 i 未使用
                new_mask = mask | (1 << i)
                actual_damage = self._calculate_damage(hp[mask], i)
                new_hp = hp[mask] - actual_damage
                new_cost = dp[mask] + self.cost[i]

                if new_hp <= 0:
                    # 怪物被击败
                    if new_cost < dp[new_mask]:
                        dp[new_mask] = new_cost
                else:
                    if new_cost < dp[new_mask]:

```

```

        dp[new_mask] = new_cost
        hp[new_mask] = new_hp

# 找到所有状态中的最小魔法消耗
result = float('inf')
for mask in range(total_states):
    if hp[mask] <= 0 and dp[mask] < result:
        result = dp[mask]

return -1 if result == float('inf') else int(result)

def min_magic_cost_greedy(self, m: int) -> int:
    """
    方法 3: 贪心+回溯优化
    按技能性价比排序，优先尝试高性价比技能
    """
    if m <= 0:
        return 0

    if self.n == 0:
        return -1

    # 计算技能性价比 (伤害/消耗)
    skills = []
    for i in range(self.n):
        efficiency = self.damage[i] / self.cost[i] if self.cost[i] > 0 else float('inf')
        skills.append((i, efficiency))

    # 按性价比降序排序
    skills.sort(key=lambda x: x[1], reverse=True)

    self.min_cost = float('inf')
    used = [False] * self.n

    self._backtrack_greedy(m, 0, used, skills)

    result = -1 if self.min_cost == float('inf') else int(self.min_cost)
    return result

def _backtrack(self, remaining_hp: int, current_cost: int, used: List[bool]):
    """
    回溯函数
    """

```

```

# 基础情况：怪物已被击败
if remaining_hp <= 0:
    if current_cost < self.min_cost:
        self.min_cost = current_cost
    return

# 剪枝优化：如果当前魔法消耗已经超过已知最优解，提前返回
if current_cost >= self.min_cost:
    return

# 尝试使用每个未使用的技能
for i in range(self.n):
    if not used[i]:
        used[i] = True

        actual_damage = self._calculate_damage(remaining_hp, i)
        self._backtrack(remaining_hp - actual_damage, current_cost + self.cost[i], used)

        used[i] = False

def _backtrack_greedy(self, remaining_hp: int, current_cost: int, used: List[bool],
                      skills: List[Tuple[int, float]]):
    """
    贪心优化的回溯函数
    """

    if remaining_hp <= 0:
        if current_cost < self.min_cost:
            self.min_cost = current_cost
        return

    if current_cost >= self.min_cost:
        return

    # 按性价比顺序尝试技能
    for skill_idx, _ in skills:
        i = skill_idx
        if not used[i]:
            used[i] = True

            actual_damage = self._calculate_damage(remaining_hp, i)
            self._backtrack_greedy(remaining_hp - actual_damage, current_cost + self.cost[i],
                                  used, skills)

```

```

        used[i] = False

def _calculate_damage(self, remaining_hp: int, skill_index: int) -> int:
    """
    计算实际伤害（考虑双倍伤害）
    """
    return self.damage[skill_index] * 2 if remaining_hp <= self.threshold[skill_index] else
self.damage[skill_index]

# 补充训练题目 - Python 实现

def coin_change(coins: List[int], amount: int) -> int:
    """
    LeetCode 322. 零钱兑换
    给定不同面额的硬币 coins 和总金额 amount，计算凑成总金额所需的最少硬币个数。
    """
    dp = [amount + 1] * (amount + 1)
    dp[0] = 0

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] = min(dp[i], dp[i - coin] + 1)

    return -1 if dp[amount] > amount else dp[amount]

def change(amount: int, coins: List[int]) -> int:
    """
    LeetCode 518. 零钱兑换 II
    给定不同面额的硬币和一个总金额，计算可以凑成总金额的硬币组合数。
    """
    dp = [0] * (amount + 1)
    dp[0] = 1

    for coin in coins:
        for i in range(coin, amount + 1):
            dp[i] += dp[i - coin]

    return dp[amount]

def num_squares(n: int) -> int:
    """
    LeetCode 279. 完全平方数
    给定正整数 n，找到若干个完全平方数使得它们的和等于 n，返回需要的最少完全平方数。
    """

```

```
"""
dp = [float('inf')] * (n + 1)
dp[0] = 0

for i in range(1, n + 1):
    j = 1
    while j * j <= i:
        dp[i] = min(dp[i], dp[i - j * j] + 1)
        j += 1

return dp[n]
```

```
def combination_sum4(nums: List[int], target: int) -> int:
```

LeetCode 377. 组合总和 IV

给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。

```
"""

dp = [0] * (target + 1)
dp[0] = 1

for i in range(1, target + 1):
    for num in nums:
        if i >= num:
            dp[i] += dp[i - num]

return dp[target]
```

```
def can_partition(nums: List[int]) -> bool:
```

LeetCode 416. 分割等和子集

给定一个只包含正整数的非空数组，判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

```
total_sum = sum(nums)

if total_sum % 2 != 0:
    return False

target = total_sum // 2
dp = [False] * (target + 1)
dp[0] = True

for num in nums:
```

```
        for i in range(target, num - 1, -1):
            dp[i] = dp[i] or dp[i - num]

    return dp[target]
```

```
def knapsack(weights: List[int], values: List[int], capacity: int) -> int:
    """
```

0-1 背包问题

给定物品重量和价值，在不超过背包容量的情况下，求能装入的最大价值。

```
"""
```

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

```
for i in range(n):
    for j in range(capacity, weights[i] - 1, -1):
        dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
```

```
return dp[capacity]
```

```
def unbounded_knapsack(weights: List[int], values: List[int], capacity: int) -> int:
    """
```

完全背包问题

物品可以无限次使用，求在不超过背包容量的情况下最大价值。

```
"""
```

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

```
for i in range(n):
    for j in range(weights[i], capacity + 1):
        dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
```

```
return dp[capacity]
```

```
# 测试函数
```

```
def test_skill_monster_advanced():
    """测试函数"""

# 测试用例 1: 简单的打怪场景
```

```
damage1 = [3, 4, 5]
```

```
cost1 = [2, 3, 4]
```

```
threshold1 = [5, 3, 2]
```

```
m1 = 10
```

```

solver1 = SkillMonsterAdvancedSolver(damage1, cost1, threshold1)
result1 = solver1.min_magic_cost(m1)
result1_dp = solver1.min_magic_cost_dp(m1)
result1_greedy = solver1.min_magic_cost_greedy(m1)

print("测试用例 1:")
print(f"技能伤害: {damage1}")
print(f"技能消耗: {cost1}")
print(f"技能阈值: {threshold1}")
print(f"怪物血量: {m1}")
print(f"回溯算法结果: {result1}")
print(f"动态规划结果: {result1_dp}")
print(f"贪心优化结果: {result1_greedy}")
print()

# 测试用例 2: 无法击败怪物的情况
damage2 = [3, 4]
cost2 = [2, 3]
threshold2 = [10, 8]
m2 = 20

solver2 = SkillMonsterAdvancedSolver(damage2, cost2, threshold2)
result2 = solver2.min_magic_cost(m2)

print("测试用例 2:")
print(f"技能伤害: {damage2}")
print(f"技能消耗: {cost2}")
print(f"技能阈值: {threshold2}")
print(f"怪物血量: {m2}")
print(f"最少魔法消耗: {result2}")
print()

# 测试补充题目
print("== 补充训练题目测试 ==")

# 测试零钱兑换
coins = [1, 2, 5]
amount = 11
print(f"零钱兑换: coins={coins}, amount={amount}, 结果={coin_change(coins, amount)}")

# 测试零钱兑换 II
print(f"零钱兑换 II: coins={coins}, amount={amount}, 结果={change(amount, coins)}")

```

```

# 测试完全平方数
n = 12
print(f"完全平方数: n={n}, 结果={num_squares(n)}")

# 测试组合总和 IV
nums = [1, 2, 3]
target = 4
print(f"组合总和 IV: nums={nums}, target={target}, 结果={combination_sum4(nums, target)}")

# 测试分割等和子集
partition_nums = [1, 5, 11, 5]
print(f"分割等和子集: nums={partition_nums}, 结果={can_partition(partition_nums)}")

# 测试 0-1 背包问题
weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
capacity = 8
print(f"0-1 背包问题: weights={weights}, values={values}, capacity={capacity}, 结果={knapsack(weights, values, capacity)}")

# 测试完全背包问题
print(f"完全背包问题: weights={weights}, values={values}, capacity={capacity}, 结果={unbounded_knapsack(weights, values, capacity)}")

if __name__ == "__main__":
    test_skill_monster_advanced()

"""

```

算法技巧总结 - Python 版本

核心概念:

1. 回溯算法框架:

- 状态定义: 当前选择、剩余选择、目标状态
- 选择策略: 遍历所有可能的选择
- 终止条件: 达到目标状态或无法继续
- 回溯操作: 撤销当前选择, 尝试其他选择

2. 动态规划技术:

- 状态压缩: 使用位运算表示状态
- 状态转移: 根据当前状态推导下一状态
- 边界处理: 处理初始状态和终止状态

3. 优化技巧:

- 剪枝优化：提前终止无效分支
- 排序优化：优先搜索更优路径
- 贪心策略：局部最优选择

Python 特有优势：

1. 内置大整数支持：无需担心数值溢出
2. 列表操作简便：灵活的列表切片和操作
3. 类型注解：提高代码可读性

调试技巧：

1. 使用 pdb 进行调试
2. 打印中间状态变量
3. 使用 assert 进行条件验证

性能优化：

1. 避免不必要的列表复制
2. 使用局部变量减少属性查找
3. 利用 Python 内置函数的高效实现

工程化实践：

1. 类型注解：提高代码可读性
2. 异常处理：确保程序健壮性
3. 单元测试：保证代码质量
4. 文档字符串：提供清晰的接口说明

边界情况处理：

1. 空输入：返回默认值或错误
2. 边界值：测试最小/最大输入
3. 异常情况：处理非法输入和边界条件

"""

=====

文件：Code06_SuperPalindromesII.cpp

=====

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <climits>
#include <limits>

using namespace std;
```

```

/***
 * 超级回文数 II 问题 - C++版本
 *
 * 问题描述:
 * 如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。
 * 给定两个正整数 L 和 R (以字符串形式表示)，返回包含在范围 [L, R] 中的超级回文数的最小值和最大值。
 * 如果不存在超级回文数，返回 [-1, -1]。
 *
 * 算法思路:
 * 方法 1: 枚举法 - 通过生成回文数来优化搜索
 * 方法 2: 打表法 - 预计算所有可能的超级回文数
 *
 * 时间复杂度分析:
 * - 枚举法:  $O(\sqrt{R} * \log R)$ , 其中  $\sqrt{R}$  为平方根范围,  $\log R$  为回文判断时间
 * - 打表法: 预处理  $O(K * \log K)$ , 查询  $O(\log K)$ , 其中 K 为超级回文数个数 (约 70)
 *
 * 空间复杂度分析:
 * - 枚举法:  $O(1)$ , 常数额外空间
 * - 打表法:  $O(K)$ , 存储预计算的超级回文数
 *
 * 工程化考量:
 * 1. 大数处理: 使用 long long 类型避免溢出
 * 2. 边界处理: 处理 L 和 R 的边界情况
 * 3. 性能优化: 选择合适的算法策略
 * 4. 可测试性: 设计全面的测试用例
 */

```

```

class SuperPalindromesIISolver {
public:
    /**
     * 方法 1: 枚举法
     * 通过生成回文数来优化搜索，避免直接遍历大范围
     */
    vector<long long> superpalindromesInRange(string left, string right) {
        long long L = stoll(left);
        long long R = stoll(right);
        long long minValue = numeric_limits<long long>::max();
        long long maxValue = numeric_limits<long long>::min();
        bool found = false;

        // 预计算的超级回文数数组
    }
}

```

```

vector<long long> superPalindromes = getSuperPalindromes();

// 查找范围内的最小值和最大值
for (long long num : superPalindromes) {
    if (num >= L && num <= R) {
        found = true;
        if (num < minValue) minValue = num;
        if (num > maxValue) maxValue = num;
    }
}

if (!found) {
    vector<long long> result;
    result.push_back(-1);
    result.push_back(-1);
    return result;
}

vector<long long> result;
result.push_back(minValue);
result.push_back(maxValue);
return result;
}

/***
 * 方法 2：枚举生成法
 * 通过生成回文数种子来构造超级回文数
 */
vector<long long> superpalindromesInRangeGenerate(string left, string right) {
    long long L = stoll(left);
    long long R = stoll(right);
    long long minValue = numeric_limits<long long>::max();
    long long maxValue = numeric_limits<long long>::min();
    bool found = false;

    long long seedLimit = 100000; // 10^5

    // 枚举所有可能的种子
    for (long long seed = 1; seed < seedLimit; seed++) {
        // 生成偶数长度的回文数
        long long evenPal = evenEnlarge(seed);
        long long squareEven = evenPal * evenPal;
    }
}

```

```

// 检查是否在范围内且是回文数
if (squareEven >= L && squareEven <= R && isPalindrome(squareEven)) {
    found = true;
    if (squareEven < minValue) minValue = squareEven;
    if (squareEven > maxValue) maxValue = squareEven;
}

// 生成奇数长度的回文数
long long oddPal = oddEnlarge(seed);
long long squareOdd = oddPal * oddPal;

// 检查是否在范围内且是回文数
if (squareOdd >= L && squareOdd <= R && isPalindrome(squareOdd)) {
    found = true;
    if (squareOdd < minValue) minValue = squareOdd;
    if (squareOdd > maxValue) maxValue = squareOdd;
}

// 如果生成的平方已经超过 R, 可以提前终止
if (squareEven > R && squareOdd > R) {
    break;
}
}

if (!found) {
    vector<long long> result;
    result.push_back(-1);
    result.push_back(-1);
    return result;
}

vector<long long> result;
result.push_back(minValue);
result.push_back(maxValue);
return result;
}

private:
/***
 * 获取预算算的超级回文数列表
 */
vector<long long> getSuperPalindromes() {
    return {

```

```

        1LL, 4LL, 9LL, 121LL, 484LL, 10201LL, 12321LL, 14641LL, 40804LL, 44944LL,
        1002001LL, 1234321LL, 4008004LL, 100020001LL, 102030201LL, 104060401LL,
        121242121LL, 123454321LL, 125686521LL, 400080004LL, 404090404LL,
        10000200001LL, 10221412201LL, 12102420121LL, 12345654321LL, 40000800004LL,
        1000002000001LL, 1002003002001LL, 1004006004001LL, 1020304030201LL,
        1022325232201LL, 1024348434201LL, 1210024200121LL, 1212225222121LL,
        1214428244121LL, 1232346432321LL, 1234567654321LL, 4000008000004LL,
        4004009004004LL, 100000020000001LL, 100220141022001LL, 102012040210201LL,
        102234363432201LL, 121000242000121LL, 121242363242121LL, 123212464212321LL,
        123456787654321LL, 400000080000004LL, 10000000200000001LL, 10002000300020001LL,
        10004000600040001LL, 10020210401202001LL, 10022212521222001LL, 10024214841242001LL,
        10201020402010201LL, 10203040504030201LL, 10205060806050201LL, 10221432623412201LL,
        10223454745432201LL, 12100002420000121LL, 12102202520220121LL, 12104402820440121LL,
        12122232623222121LL, 12124434743442121LL, 12321024642012321LL, 12323244744232321LL,
        12343456865434321LL, 12345678987654321LL, 40000000800000004LL, 40004000900040004LL
    } ;
}

/***
 * 根据种子扩充到偶数长度的回文数字
 */
long long evenEnlarge(long long seed) {
    long long ans = seed;
    long long temp = seed;
    while (temp > 0) {
        ans = ans * 10 + temp % 10;
        temp /= 10;
    }
    return ans;
}

/***
 * 根据种子扩充到奇数长度的回文数字
 */
long long oddEnlarge(long long seed) {
    long long ans = seed;
    long long temp = seed / 10;
    while (temp > 0) {
        ans = ans * 10 + temp % 10;
        temp /= 10;
    }
    return ans;
}

```

```
/**  
 * 判断一个数是否是回文数  
 */  
bool isPalindrome(long long x) {  
    if (x < 0) return false;  
    if (x < 10) return true;  
  
    long long original = x;  
    long long reversed = 0;  
  
    while (x > 0) {  
        reversed = reversed * 10 + x % 10;  
        x /= 10;  
    }  
  
    return original == reversed;  
}  
};
```

```
/**  
 * 补充训练题目 - C++实现  
 */  
  
/**  
 * 判断一个数是否是回文数（全局函数版本）  
 */  
bool isPalindrome(long long x) {  
    if (x < 0) return false;  
    if (x < 10) return true;  
  
    long long original = x;  
    long long reversed = 0;  
  
    while (x > 0) {  
        reversed = reversed * 10 + x % 10;  
        x /= 10;  
    }  
  
    return original == reversed;  
}
```

```
/**
```

```

* LeetCode 479. 最大回文数乘积
* 给定一个整数 n，返回可表示为两个 n 位整数乘积的最大回文整数
*/
long long largestPalindrome(int n) {
    if (n == 1) return 9;

    long long maxNum = 1;
    for (int i = 0; i < n; i++) {
        maxNum *= 10;
    }
    maxNum -= 1;

    long long minNum = 1;
    for (int i = 0; i < n - 1; i++) {
        minNum *= 10;
    }

    for (long long i = maxNum; i >= minNum; i--) {
        // 创建回文数
        string s = to_string(i);
        string rev = s;
        reverse(rev.begin(), rev.end());
        long long palindrome = stoll(s + rev);

        // 检查是否可以分解为两个 n 位数的乘积
        for (long long j = maxNum; j * j >= palindrome; j--) {
            if (palindrome % j == 0) {
                long long factor = palindrome / j;
                if (factor >= minNum && factor <= maxNum) {
                    return palindrome;
                }
            }
        }
    }

    return -1;
}

/**
* LeetCode 906. 超级回文数（统计版本）
* 统计范围内的超级回文数个数
*/
int countSuperPalindromes(string left, string right) {

```

```

long long L = stoll(left);
long long R = stoll(right);
int count = 0;

vector<long long> superPalindromes = {
    1LL, 4LL, 9LL, 121LL, 484LL, 10201LL, 12321LL, 14641LL, 40804LL, 44944LL,
    1002001LL, 1234321LL, 4008004LL, 100020001LL, 102030201LL, 104060401LL,
    121242121LL, 123454321LL, 125686521LL, 400080004LL, 404090404LL,
    10000200001LL, 10221412201LL, 12102420121LL, 12345654321LL, 40000800004LL
};

for (long long num : superPalindromes) {
    if (num >= L && num <= R) {
        count++;
    }
}

return count;
}

/***
 * LeetCode 9. 回文数（大数版本）
 * 支持大数判断的回文数函数
 */
bool isPalindromeBig(string s) {
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s[left] != s[right]) {
            return false;
        }
        left++;
        right--;
    }
    return true;
}

/***
 * 生成指定范围内的所有回文数
 */
vector<long long> generatePalindromesInRange(long long start, long long end) {
    vector<long long> result;

    for (long long i = start; i <= end; i++) {

```

```

        if (isPalindrome(i)) {
            result.push_back(i);
        }
    }

    return result;
}

// 测试函数
void testSuperPalindromesII() {
    SuperPalindromesIISolver solver;

    // 测试用例 1
    string left1 = "4";
    string right1 = "1000";
    vector<long long> result1 = solver.superpalindromesInRange(left1, right1);
    vector<long long> result1_gen = solver.superpalindromesInRangeGenerate(left1, right1);

    cout << "测试用例 1:" << endl;
    cout << "范围: [" << left1 << ", " << right1 << "]" << endl;
    cout << "打表法结果: [" << result1[0] << ", " << result1[1] << "]" << endl;
    cout << "生成法结果: [" << result1_gen[0] << ", " << result1_gen[1] << "]" << endl;
    cout << endl;

    // 测试用例 2: 无超级回文数的情况
    string left2 = "10000000000000000000";
    string right2 = "10000000000000000000";
    vector<long long> result2 = solver.superpalindromesInRange(left2, right2);

    cout << "测试用例 2:" << endl;
    cout << "范围: [" << left2 << ", " << right2 << "]" << endl;
    cout << "结果: [" << result2[0] << ", " << result2[1] << "]" << endl;
    cout << endl;

    // 测试补充题目
    cout << "==== 补充训练题目测试 ===" << endl;

    // 测试最大回文数乘积
    cout << "最大回文数乘积(n=2): " << largestPalindrome(2) << endl;

    // 测试超级回文数统计
    cout << "超级回文数统计: 范围[4, 1000] -> " << countSuperPalindromes("4", "1000") << endl;
}

```

```
// 测试大数回文判断
string bigNum = "12345678987654321";
cout << "大数回文判断: " << bigNum << " -> " << (isPalindromeBig(bigNum) ? "是" : "否") <<
endl;

// 测试回文数生成
vector<long long> palindromes = generatePalindromesInRange(100, 200);
cout << "回文数生成(100-200): 数量=" << palindromes.size() << endl;
}

int main() {
    testSuperPalindromesII();
    return 0;
}

/***
 * 算法技巧总结 - C++版本
 *
 * 核心概念:
 * 1. 回文数生成技术:
 *     - 种子生成法: 通过种子数字构造回文数
 *     - 对称性利用: 利用回文数的对称特征
 *     - 数学方法: 避免字符串转换的开销
 *
 * 2. 算法选择策略:
 *     - 小范围查询: 枚举法更节省内存
 *     - 多次查询: 打表法性能最优
 *     - 大数据范围: 需要数学优化和边界处理
 *
 * 3. 性能优化技巧:
 *     - 提前终止: 当结果超出范围时提前结束搜索
 *     - 数学优化: 使用位运算和数学方法
 *     - 缓存策略: 预计算常用结果
 *
 * 调试技巧:
 * 1. 边界值测试: 最小/最大输入、边界情况
 * 2. 中间结果打印: 跟踪算法执行过程
 * 3. 性能分析: 使用 profiler 工具分析瓶颈
 *
 * 工程化实践:
 * 1. 模块化设计: 分离算法逻辑和业务逻辑
 * 2. 异常安全: 确保资源正确释放
 * 3. 代码可读性: 使用有意义的变量名和注释
*/
```

*/

=====

文件: Code06_SuperPalindromesII.java

=====

```
package class043;
```

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

```
/**
```

```
* 超级回文数 II 问题
```

```
*
```

```
* 问题描述:
```

```
* 如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。
```

```
* 现在，给定两个正整数 L 和 R（以字符串形式表示），
```

```
* 返回包含在范围 [L, R] 中的超级回文数的最小值和最大值。
```

```
* 如果不存在超级回文数，返回 [-1, -1]。
```

```
*
```

```
* 算法思路:
```

```
* 方法 1：枚举法
```

```
* 1. 由于 L 和 R 的范围可达  $10^{18}$ ，直接遍历会超时
```

```
* 2. 考虑到超级回文数是另一个回文数的平方，我们可以枚举较小的回文数
```

```
* 3. 平方根的范围约为  $10^9$ ，但仍太大，继续优化
```

```
* 4. 回文数可以通过“种子”生成，如 123 可以生成 123321（偶数长度）和 12321（奇数长度）
```

```
* 5. 种子的范围约为  $10^5$ ，可以接受
```

```
*
```

```
* 方法 2：打表法
```

```
* 1. 预先计算出所有可能的超级回文数
```

```
* 2. 在查询时直接在已排序数组中查找范围内的最小值和最大值
```

```
* 3. 时间复杂度最低，但需要额外的存储空间
```

```
*
```

```
* 时间复杂度分析:
```

```
* 方法 1：枚举法 -  $O(\sqrt{R} * \log R)$ 
```

```
* - 枚举回文数需要  $O(\sqrt{R})$  时间
```

```
* - 检查每个数是否为回文需要  $O(\log R)$  时间（数的位数）
```

```
*
```

```
* 方法 2：打表法 -  $O(\log R)$ 
```

```
* - 预处理阶段需要  $O(K * \log K)$  时间，其中 K 是超级回文数的个数
```

```
* - 查询阶段只需要在已排序数组中进行二分查找，时间复杂度为  $O(\log K)$ 
```

```
*
```

```
* 空间复杂度分析:
```

```
* 方法 1：枚举法 -  $O(1)$ 
```

- * - 只需要常数额外空间
- *
- * 方法 2: 打表法 - $O(K)$
- * - 需要存储所有超级回文数, K 约为 70
- *
- * 工程化考量:
 - * 1. 异常处理: 处理大数运算和字符串转换
 - * 2. 可配置性: 可以调整算法策略 (枚举 vs 打表)
 - * 3. 性能优化: 使用打表法避免重复计算
 - * 4. 鲁棒性: 处理边界情况和溢出问题
- *
- * 相关题目:
 - * 1. LeetCode 906. 超级回文数 - <https://leetcode.cn/problems/super-palindromes/>
 - * 2. LeetCode 9. 回文数 - <https://leetcode.cn/problems/palindrome-number/>
 - * 3. LeetCode 479. 最大回文数乘积 - <https://leetcode.cn/problems/largest-palindrome-product/>
 - * 4. LeetCode 680. 验证回文字符串 II - <https://leetcode.cn/problems/valid-palindrome-ii/>
 - * 5. LeetCode 125. 验证回文串 - <https://leetcode.cn/problems/valid-palindrome/>
 - * 6. LeetCode 131. 分割回文串 - <https://leetcode.cn/problems/palindrome-partitioning/>
 - * 7. LeetCode 132. 分割回文串 II - <https://leetcode.cn/problems/palindrome-partitioning-ii/>
 - * 8. LeetCode 5. 最长回文子串 - <https://leetcode.cn/problems/longest-palindromic-substring/>
 - * 9. LeetCode 516. 最长回文子序列 - <https://leetcode.cn/problems/longest-palindromic-subsequence/>
 - * 10. LeetCode 336. 回文对 - <https://leetcode.cn/problems/palindrome-pairs/>
 - * 11. 牛客网 - 回文数 - <https://www.nowcoder.com/practice/38802713414c4852b6982410c4187dd2>
 - * 12. 牛客网 - 最长回文子串 - <https://www.nowcoder.com/practice/b4525d1d84934cf280439aeecc36f4af>
 - * 13. 牛客网 - 分割回文串 - <https://www.nowcoder.com/practice/10454d86222841189316919d3e988584>
 - * 14. 牛客网 - 复原 IP 地址 - <https://www.nowcoder.com/practice/10454d86222841189316919d3e988584>
 - * 15. Codeforces 1335D - Anti-Sudoku - <https://codeforces.com/problemset/problem/1335/D>
 - * 16. Codeforces 1332B - Composite Coloring - <https://codeforces.com/problemset/problem/1332/B>
 - * 17. Codeforces 1327D - Infinite Path - <https://codeforces.com/problemset/problem/1327/D>
 - * 18. Codeforces 1436E - Complicated Computations -
<https://codeforces.com/problemset/problem/1436/E>
 - * 19. AtCoder ABC136D - Gathering Children - https://atcoder.jp/contests/abc136/tasks/abc136_d
 - * 20. AtCoder ABC159E - Dividing Chocolate - https://atcoder.jp/contests/abc159/tasks/abc159_e
 - * 21. 洛谷 P1012 - 拼数 - <https://www.luogu.com.cn/problem/P1012>
 - * 22. 洛谷 P1157 - 组合的输出 - <https://www.luogu.com.cn/problem/P1157>
 - * 23. 洛谷 P1048 - 采药 - <https://www.luogu.com.cn/problem/P1048>
 - * 24. 洛谷 P1125 - 笨小猴 - <https://www.luogu.com.cn/problem/P1125>
 - * 25. HackerRank - Sherlock and the Valid String -
<https://www.hackerrank.com/challenges/sherlock-and-valid-string/problem>
 - * 26. HackerRank - The Longest Palindromic Subsequence -
<https://www.hackerrank.com/challenges/longest-palindromic-subsequence/problem>
 - * 27. HackerRank - Sherlock and Cost - <https://www.hackerrank.com/challenges/sherlock-and-cost/problem>

```

cost/problem

* 28. HackerRank - Split the String - https://www.hackerrank.com/challenges/split-the-
string/problem

* 29. UVa 10945 - Mother Bear -

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&p
roblem=1886

* 30. UVa 10189 - Minesweeper -

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&p
roblem=10189

* 31. POJ 3083 - Children of the Candy Corn - http://poj.org/problem?id=3083

* 32. POJ 1163 - The Triangle - http://poj.org/problem?id=1163

* 33. HDU 1203 - I NEED A OFFER! - http://acm.hdu.edu.cn/showproblem.php?pid=1203

* 34. HDU 2000 - ASCII 码排序 - http://acm.hdu.edu.cn/showproblem.php?pid=2000

* 35. LintCode 415 - Valid Palindrome - https://www.lintcode.com/problem/valid-palindrome/

* 36. LintCode 1178 - Valid Palindrome III - https://www.lintcode.com/problem/1178/

* 37. LintCode 125 - Valid Palindrome II - https://www.lintcode.com/problem/125/

* 38. LintCode 200 - Longest Palindromic Substring - https://www.lintcode.com/problem/200/

* 39. LintCode 130 - Heapify - https://www.lintcode.com/problem/130/

* 40. AcWing 901 - 滑雪 - https://www.acwing.com/problem/content/903/

* 41. AcWing 1482 - 动态规划专题 - https://www.acwing.com/activity/content/introduction/19/
*/
public class Code06_SuperPalindromesII {

    /**
     * 方法 1：枚举法
     *
     * @param left 范围左边界（字符串形式）
     * @param right 范围右边界（字符串形式）
     * @return 范围内超级回文数的最小值和最大值，如果不存在返回[-1, -1]
     */
    public static long[] superpalindromesInRange(String left, String right) {
        long l = Long.valueOf(left);
        long r = Long.valueOf(right);

        // 初始化结果数组
        long minValue = Long.MAX_VALUE;
        long maxValue = Long.MIN_VALUE;
        boolean found = false;

        // l...r long
        // x 根号，范围 limit
        long limit = (long) Math.sqrt((double) r);
        // seed : 枚举量很小，10^18 -> 10^9 -> 10^5
    }
}

```

```

// seed : 奇数长度回文、偶数长度回文
long seed = 1;
// num : 根号 x, num^2 -> x
long num = 0;

do {
    // seed 生成偶数长度回文数字
    // 123 -> 123321
    num = evenEnlarge(seed);
    long square = num * num;
    if (square >= 1 && square <= r && isPalindrome(square)) {
        minValue = Math.min(minValue, square);
        maxValue = Math.max(maxValue, square);
        found = true;
    }
}

// seed 生成奇数长度回文数字
// 123 -> 12321
num = oddEnlarge(seed);
square = num * num;
if (square >= 1 && square <= r && isPalindrome(square)) {
    minValue = Math.min(minValue, square);
    maxValue = Math.max(maxValue, square);
    found = true;
}

// 123 -> 124 -> 125
seed++;
} while (num < limit);

if (!found) {
    return new long[] {-1, -1};
}

return new long[] {minValue, maxValue};
}

/**
 * 根据种子扩充到偶数长度的回文数字并返回
 *
 * @param seed 种子数字
 * @return 偶数长度的回文数
 */

```

```
* 例如: seed=123, 返回 123321
*/
public static long evenEnlarge(long seed) {
    long ans = seed;
    while (seed != 0) {
        ans = ans * 10 + seed % 10;
        seed /= 10;
    }
    return ans;
}

/***
 * 根据种子扩充到奇数长度的回文数字并返回
 *
 * @param seed 种子数字
 * @return 奇数长度的回文数
 *
 * 例如: seed=123, 返回 12321
 */
public static long oddEnlarge(long seed) {
    long ans = seed;
    seed /= 10;
    while (seed != 0) {
        ans = ans * 10 + seed % 10;
        seed /= 10;
    }
    return ans;
}

/***
 * 验证 long 类型的数字 num, 是不是回文数字
 *
 * @param num 待检查的数字
 * @return 如果是回文数返回 true, 否则返回 false
 *
 * 算法思路:
 * 1. 通过 offset 定位最高位
 * 2. 比较最高位和最低位是否相等
 * 3. 去掉最高位和最低位, 继续比较
 * 4. 直到所有位都比较完毕
 */
public static boolean isPalindrome(long num) {
    long offset = 1;
```

```

// 注意这么写是为了防止溢出
while (num / offset >= 10) {
    offset *= 10;
}
// num      : 52725
// offset : 10000
// 首尾判断
while (num != 0) {
    if (num / offset != num % 10) {
        return false;
    }
    num = (num % offset) / 10;
    offset /= 100;
}
return true;
}

/**
 * 方法 2：打表法（最优解）
 *
 * @param left 范围左边界（字符串形式）
 * @param right 范围右边界（字符串形式）
 * @return 范围内超级回文数的最小值和最大值，如果不存在返回[-1, -1]
 *
 * 算法思路：
 * 1. 预先计算出所有的超级回文数并存储在数组中
 * 2. 在查询时，找到范围内第一个和最后一个超级回文数的位置
 * 3. 通过位置差计算数量
 *
 * 优势：
 * 1. 时间复杂度最低
 * 2. 避免重复计算
 * 3. 适合多次查询的场景
 */
public static long[] superpalindromesInRange2(String left, String right) {
    long l = Long.parseLong(left);
    long r = Long.parseLong(right);

    // 查找范围内的最小值和最大值
    Long minValue = null;
    Long maxValue = null;

    for (long superPalindrome : record) {

```

```
if (superPalindrome >= l && superPalindrome <= r) {
    if (minVal == null || superPalindrome < minVal) {
        minVal = superPalindrome;
    }
    if (maxVal == null || superPalindrome > maxVal) {
        maxVal = superPalindrome;
    }
}

if (minVal == null) {
    return new long[] {-1, -1};
}

return new long[] {minVal, maxVal};
}

// 预计算的所有超级回文数（已排序）
public static long[] record = new long[] {
    1L,
    4L,
    9L,
    121L,
    484L,
    10201L,
    12321L,
    14641L,
    40804L,
    44944L,
    1002001L,
    1234321L,
    4008004L,
    100020001L,
    102030201L,
    104060401L,
    12102420121L,
    123454321L,
    125686521L,
    400080004L,
    404090404L,
    10000200001L,
    10221412201L,
    12102420121L,
```

12345654321L,
40000800004L,
1000002000001L,
1002003002001L,
1004006004001L,
1020304030201L,
1022325232201L,
1024348434201L,
1210024200121L,
1212225222121L,
1214428244121L,
1232346432321L,
1234567654321L,
4000008000004L,
4004009004004L,
100000020000001L,
100220141022001L,
102012040210201L,
102234363432201L,
121000242000121L,
121242363242121L,
123212464212321L,
123456787654321L,
400000080000004L,
10000000200000001L,
10002000300020001L,
10004000600040001L,
10020210401202001L,
10022212521222001L,
10024214841242001L,
10201020402010201L,
10203040504030201L,
10205060806050201L,
10221432623412201L,
10223454745432201L,
12100002420000121L,
12102202520220121L,
12104402820440121L,
12122232623222121L,
12124434743442121L,
12321024642012321L,
12323244744232321L,
12343456865434321L,

```
12345678987654321L,
40000000800000004L,
40004000900040004L,
100000000200000001L,
1000220014100220001L,
1002003004003002001L,
1002223236323222001L,
1020100204020010201L,
1020322416142230201L,
1022123226223212201L,
1022345658565432201L,
1210000024200000121L,
1210242036302420121L,
1212203226223022121L,
1212445458545442121L,
1232100246420012321L,
1232344458544432321L,
1234323468643234321L,
4000000008000000004L
} ;
```

```
// 测试函数
public static void main(String[] args) {
    // 测试用例 1
    String left1 = "4";
    String right1 = "1000";
    long[] result1 = superpalindromesInRange(left1, right1);
    long[] result2 = superpalindromesInRange2(left1, right1);
    System.out.println("测试用例 1:");
    System.out.println("范围: [" + left1 + ", " + right1 + "]");
    System.out.println("枚举法结果: [" + result1[0] + ", " + result1[1] + "]");
    System.out.println("打表法结果: [" + result2[0] + ", " + result2[1] + "]");

    // 测试用例 2
    String left2 = "1";
    String right2 = "2";
    long[] result3 = superpalindromesInRange(left2, right2);
    long[] result4 = superpalindromesInRange2(left2, right2);
    System.out.println("\n 测试用例 2:");
    System.out.println("范围: [" + left2 + ", " + right2 + "]");
    System.out.println("枚举法结果: [" + result3[0] + ", " + result3[1] + "]");
    System.out.println("打表法结果: [" + result4[0] + ", " + result4[1] + "]");
```

```

// 测试用例 3: 无超级回文数
String left3 = "10000000000000000000";
String right3 = "20000000000000000000";
long[] result5 = superpalindromesInRange(left3, right3);
long[] result6 = superpalindromesInRange2(left3, right3);
System.out.println("\n 测试用例 3:");
System.out.println("范围: [" + left3 + ", " + right3 + "]");
System.out.println("枚举法结果: [" + result5[0] + ", " + result5[1] + "]");
System.out.println("打表法结果: [" + result6[0] + ", " + result6[1] + "]");
}

/**
 * 补充训练题目
 *
 * 1. LeetCode 479. 最大回文数乘积
 * 题目描述: 给定一个整数 n , 返回 可表示为两个 n 位整数乘积的 最大回文整数 。因为答案可能非常大, 所以返回它对 1337 取余。
 * 解题思路: 生成可能的回文数, 然后检查是否可以分解为两个 n 位数的乘积
 */
public static int largestPalindrome(int n) {
    if (n == 1) return 9;
    int max = (int) Math.pow(10, n) - 1;
    int min = (int) Math.pow(10, n-1);

    for (int i = max; i >= min; i--) {
        long palindrome = createPalindrome(i);
        for (long j = max; j*j >= palindrome; j--) {
            if (palindrome % j == 0 && palindrome / j <= max && palindrome / j >= min) {
                return (int) (palindrome % 1337);
            }
        }
    }
    return -1; // 不应该到达这里
}

// 辅助方法: 根据数字创建回文数
private static long createPalindrome(int num) {
    String s = Integer.toString(num);
    StringBuilder sb = new StringBuilder(s);
    return Long.parseLong(s + sb.reverse().toString());
}

/**

```

```

* 2. LeetCode 680. 验证回文字符串 II
*   题目描述: 给定一个非空字符串 s, 最多删除一个字符, 判断是否能成为回文字符串。
*   解题思路: 使用双指针, 如果遇到不匹配的字符, 尝试删除左边或右边的字符
*/
public static boolean validPalindrome(String s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            // 尝试删除左边或右边的字符
            return isPalindromeRange(s, left + 1, right) || isPalindromeRange(s, left, right
- 1);
        }
        left++;
        right--;
    }

    return true;
}

private static boolean isPalindromeRange(String s, int left, int right) {
    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }
        left++;
        right--;
    }

    return true;
}

/**
* 3. CodeChef PALIN - The Next Palindrome
*   题目描述: 给定一个数字 N, 找到大于 N 的最小回文数。
*   解题思路: 构造下一个回文数, 可以通过处理中间位来高效实现
*/
public static String nextPalindrome(String N) {
    char[] arr = N.toCharArray();
    int n = arr.length;
    int mid = n / 2;

    // 检查是否是全 9 的情况
    boolean all9 = true;

```

```

for (char c : arr) {
    if (c != '9') {
        all9 = false;
        break;
    }
}

if (all9) {
    StringBuilder sb = new StringBuilder();
    sb.append('1');
    for (int i = 0; i < n - 1; i++) {
        sb.append('0');
    }
    sb.append('1');
    return sb.toString();
}

// 构造回文
boolean leftIsSmaller = false;
int i = mid - 1;
int j = (n % 2 == 0) ? mid : mid + 1;

while (i >= 0 && arr[i] == arr[j]) {
    i--;
    j++;
}

if (i < 0 || arr[i] < arr[j]) {
    leftIsSmaller = true;
}

// 镜像左半部分到右半部分
while (i >= 0) {
    arr[j++] = arr[i--];
}

// 如果左半部分小于右半部分，需要增加中间部分
if (leftIsSmaller) {
    int carry = 1;
    i = mid - 1;
    if (n % 2 == 1) {
        arr[mid] += carry;
        carry = (arr[mid] - '0') / 10;
        arr[mid] = (char)((arr[mid] - '0') % 10 + '0');
    }
}

```

```

        j = mid + 1;
    } else {
        j = mid;
    }

    while (i >= 0 && carry > 0) {
        arr[i] += carry;
        carry = (arr[i] - '0') / 10;
        arr[i] = (char)((arr[i] - '0') % 10 + '0');
        arr[j++] = arr[i--];
    }
}

return new String(arr);
}

/**
 * 4. HackerRank - The Longest Palindromic Subsequence
 * 题目描述：给定一个字符串 s，找到最长回文子序列的长度。子序列可以通过删除字符来形成，但不能改变剩余字符的相对顺序。
 * 解题思路：使用动态规划，dp[i][j]表示 s[i...j] 的最长回文子序列长度
 */
public static int longestPalindromeSubseq(String s) {
    int n = s.length();
    int[][] dp = new int[n][n];

    // 单个字符的回文长度为 1
    for (int i = 0; i < n; i++) {
        dp[i][i] = 1;
    }

    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[0][n - 1];
}

```

```

}

/***
 * 5. LintCode 1178 - Valid Palindrome III
 * 题目描述：给定一个字符串 s 和一个整数 k，判断是否可以通过最多删除 k 个字符使得 s 成为回文字
字符串。
 * 解题思路：使用动态规划，dp[i][j]表示将 s[i...j] 变成回文所需删除的最少字符数
*/
public static boolean isValidPalindrome(String s, int k) {
    int n = s.length();
    int[][] dp = new int[n][n];

    for (int len = 2; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s.charAt(i) == s.charAt(j)) {
                dp[i][j] = dp[i + 1][j - 1];
            } else {
                dp[i][j] = Math.min(dp[i + 1][j], dp[i][j - 1]) + 1;
            }
        }
    }

    return dp[0][n - 1] <= k;
}

/***
 * 6. LeetCode 647. 回文子串
 * 题目描述：给你一个字符串 s，请你统计并返回这个字符串中 回文子串 的数目。
 * 回文字符串 是正着读和倒过来读一样的字符串。
 * 子字符串 是字符串中的由连续字符组成的一个序列。
 * 解题思路：使用中心扩展法，遍历每个可能的中心位置，向两边扩展检查回文
*/
public static int countSubstrings(String s) {
    int count = 0;
    int n = s.length();

    // 检查每个可能的中心位置
    for (int i = 0; i < n; i++) {
        // 以单个字符为中心的回文（奇数长度）
        count += expandAroundCenter(s, i, i);
        // 以两个字符之间为中心的回文（偶数长度）
        count += expandAroundCenter(s, i, i + 1);
    }
}

```

```

}

return count;
}

private static int expandAroundCenter(String s, int left, int right) {
    int count = 0;
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
        count++;
        left--;
        right++;
    }
    return count;
}

/***
 * 7. LeetCode 336. 回文对
 * 题目描述：给定一组唯一的单词，返回所有不同的索引对 (i, j)，使得单词 words[i] + words[j]
 * 是回文串。
 * 解题思路：使用哈希表存储每个单词的逆序，然后检查每个单词的前缀和后缀
 */
public static List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> result = new ArrayList<>();
    Map<String, Integer> map = new HashMap<>();

    // 存储单词及其索引
    for (int i = 0; i < words.length; i++) {
        map.put(new StringBuilder(words[i]).reverse().toString(), i);
    }

    // 处理空字符串
    if (map.containsKey("")) {
        int emptyIndex = map.get("");
        for (int i = 0; i < words.length; i++) {
            if (i != emptyIndex && isPalindromeString(words[i])) {
                result.add(Arrays.asList(i, emptyIndex));
                result.add(Arrays.asList(emptyIndex, i));
            }
        }
    }

    for (int i = 0; i < words.length; i++) {
        String word = words[i];

```

```

// 分割单词为前缀和后缀
for (int j = 1; j < word.length(); j++) {
    String prefix = word.substring(0, j);
    String suffix = word.substring(j);

    // 前缀是回文，检查后缀的逆序是否存在
    if (isPalindromeString(prefix) && map.containsKey(reverseString(suffix))) {
        int index = map.get(reverseString(suffix));
        if (index != i) {
            result.add(Arrays.asList(index, i));
        }
    }

    // 后缀是回文，检查前缀的逆序是否存在
    if (isPalindromeString(suffix) && map.containsKey(reverseString(prefix))) {
        int index = map.get(reverseString(prefix));
        if (index != i) {
            result.add(Arrays.asList(i, index));
        }
    }
}

// 检查整个单词的逆序是否存在
String reversed = reverseString(word);
if (map.containsKey(reversed) && map.get(reversed) != i) {
    result.add(Arrays.asList(i, map.get(reversed)));
}
}

return result;
}

private static boolean isPalindromeString(String s) {
    int left = 0, right = s.length() - 1;
    while (left < right) {
        if (s.charAt(left++) != s.charAt(right--)) {
            return false;
        }
    }
    return true;
}

```

```

private static String reverseString(String s) {
    return new StringBuilder(s).reverse().toString();
}

/***
 * 8. HackerRank - Split the String
 * 题目描述：给定一个字符串，要求将其分割成最多 K 个部分，使得每个部分都包含相同数量的' a' 和
 * ' b' 字符。
 * 解题思路：贪心算法，遍历字符串并统计 a 和 b 的数量，当两者相等时进行分割
 */
public static int splitTheString(String s, int k) {
    int aCount = 0, bCount = 0;
    int splits = 0;

    for (char c : s.toCharArray()) {
        if (c == 'a') aCount++;
        else bCount++;

        // 当 a 和 b 的数量相等时，进行分割
        if (aCount == bCount && aCount > 0) {
            splits++;
            aCount = 0;
            bCount = 0;

            // 如果已经达到最大分割次数，提前结束
            if (splits == k) {
                break;
            }
        }
    }

    // 返回分割次数（不超过 K）
    return splits;
}

/***
 * 9. LeetCode 5. 最长回文子串
 * 题目描述：给你一个字符串 s，找到 s 中最长的回文子串。
 * 解题思路：中心扩展法，枚举每个可能的中心点并向两边扩展
 */
public static String longestPalindrome(String s) {
    if (s == null || s.length() < 1) return "";
    int start = 0, end = 0;

```

```

for (int i = 0; i < s.length(); i++) {
    // 奇数长度回文
    int len1 = expandAroundCenter(s, i, i);
    // 偶数长度回文
    int len2 = expandAroundCenter(s, i, i + 1);
    int len = Math.max(len1, len2);

    if (len > end - start) {
        start = i - (len - 1) / 2;
        end = i + len / 2;
    }
}

return s.substring(start, end + 1);
}

/*
private static int expandAroundCenter(String s, int left, int right) {
    while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
        left--;
        right++;
    }
    return right - left - 1;
}
*/
/***
 * 10. LeetCode 125. 验证回文串
 * 题目描述：给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。
 * 解题思路：双指针，从两端向中间移动并比较字符
 */
public static boolean isPalindromeSentence(String s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        // 跳过非字母数字字符
        while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {
            left++;
        }
        while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {
            right--;
        }
    }
}

```

```

    }

    // 比较字符（忽略大小写）
    if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right)))
{
    return false;
}

left++;
right--;
}

return true;
}

/***
 * 11. LeetCode 680. 验证回文字符串 II
 * 题目描述：给定一个非空字符串 s，最多删除一个字符，判断是否能成为回文字符串。
 * 解题思路：双指针，当遇到不匹配时，尝试删除左边或右边的字符继续检查
 */
/*
public static boolean validPalindrome(String s) {
    int left = 0, right = s.length() - 1;

    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            // 尝试删除左边或右边的字符
            return isPalindromeRange(s, left + 1, right) || isPalindromeRange(s, left, right - 1);
        }
        left++;
        right--;
    }

    return true;
}

private static boolean isPalindromeRange(String s, int left, int right) {
    while (left < right) {
        if (s.charAt(left) != s.charAt(right)) {
            return false;
        }
        left++;
    }
}

```

```

        right--;
    }
    return true;
}
*/
/***
 * 12. LeetCode 131. 分割回文串
 * 题目描述：给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是 回文串 。返回 s 所有可能的分割方案。
 * 解题思路：回溯算法，递归地尝试所有可能的分割方式
*/
public static List<List<String>> partition(String s) {
    List<List<String>> result = new ArrayList<>();
    List<String> current = new ArrayList<>();
    backtrackPartition(s, 0, current, result);
    return result;
}

private static void backtrackPartition(String s, int start, List<String> current,
List<List<String>> result) {
    if (start >= s.length()) {
        result.add(new ArrayList<>(current));
        return;
    }

    for (int end = start; end < s.length(); end++) {
        if (isPalindromeString(s.substring(start, end + 1))) {
            current.add(s.substring(start, end + 1));
            backtrackPartition(s, end + 1, current, result);
            current.remove(current.size() - 1); // 回溯
        }
    }
}

/**
 * 13. LeetCode 132. 分割回文串 II
 * 题目描述：给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是回文。返回符合要求的最少分割次数。
 * 解题思路：动态规划，dp[i]表示 s[0...i]的最少分割次数
*/
public static int minCut(String s) {
    int n = s.length();

```

```

// dp[i]表示 s[0... i]的最少分割次数
int[] dp = new int[n];

// 初始化: 最多需要 i 次分割 (每个字符单独成一个回文串)
for (int i = 0; i < n; i++) {
    dp[i] = i;
}

// 预处理判断是否为回文串
boolean[][] isPal = new boolean[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j <= i; j++) {
        if (s.charAt(i) == s.charAt(j) && (i - j <= 2 || isPal[j + 1][i - 1])) {
            isPal[j][i] = true;
        }
    }
}

// 动态规划填表
for (int i = 0; i < n; i++) {
    if (isPal[0][i]) {
        dp[i] = 0; // 不需要分割
        continue;
    }

    for (int j = 0; j < i; j++) {
        if (isPal[j + 1][i]) {
            dp[i] = Math.min(dp[i], dp[j] + 1);
        }
    }
}

return dp[n - 1];
}

/***
 * 14. LeetCode 516. 最长回文子序列
 * 题目描述: 给你一个字符串 s , 找出其中最长的回文子序列，并返回该序列的长度。
 * 解题思路: 动态规划, dp[i][j]表示 s[i... j]范围内的最长回文子序列长度
 */
/*
public static int longestPalindromeSubseq(String s) {
    int n = s.length();

```

```

int[][] dp = new int[n][n];

// 初始化对角线
for (int i = 0; i < n; i++) {
    dp[i][i] = 1;
}

// 填充 dp 数组
for (int len = 2; len <= n; len++) {
    for (int i = 0; i + len <= n; i++) {
        int j = i + len - 1;
        if (s.charAt(i) == s.charAt(j)) {
            dp[i][j] = dp[i + 1][j - 1] + 2;
        } else {
            dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
        }
    }
}

return dp[0][n - 1];
}
*/

```

/**

* 15. Codeforces 1335D – Anti-Sudoku

* 题目描述：给定一个 9x9 的数独，将其转换为反数独，即每个行、列和 3x3 子网格中至少包含两个相同的数字。

* 解题思路：简单地将所有的 5 改为 6 即可（或者其他类似的简单变换）

*/

```

public static char[][] antiSudoku(char[][] grid) {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (grid[i][j] == '5') {
                grid[i][j] = '6';
            }
        }
    }
    return grid;
}

```

/**

* 超级回文数算法的 Python 实现

*/

```
/*
# Python 版本

def is_palindrome(x):
    """
    判断一个数是否是回文数
    时间复杂度: O(log x) - 取决于数字的位数
    空间复杂度: O(1)
    """

    if x < 0:
        return False
    original = x
    reversed_num = 0
    while x > 0:
        reversed_num = reversed_num * 10 + x % 10
        x = x // 10
    return original == reversed_num

def even_enlarge(seed):
    """
    根据种子扩充到偶数长度的回文数字
    例如: seed=123, 返回 123321
    时间复杂度: O(log seed)
    空间复杂度: O(1)
    """

    ans = seed
    temp = seed
    while temp > 0:
        ans = ans * 10 + temp % 10
        temp = temp // 10
    return ans

def odd_enlarge(seed):
    """
    根据种子扩充到奇数长度的回文数字
    例如: seed=123, 返回 12321
    时间复杂度: O(log seed)
    空间复杂度: O(1)
    """

    ans = seed
    temp = seed // 10
    while temp > 0:
        ans = ans * 10 + temp % 10
```

```

        temp = temp // 10
    return ans

# 预计算的超级回文数
SUPER_PALINDROMES = [
    1, 4, 9, 121, 484, 10201, 12321, 14641, 40804, 44944, 1002001, 1234321,
    4008004, 100020001, 102030201, 104060401, 121242121, 123454321, 125686521,
    400080004, 404090404, 10000200001, 10221412201, 12102420121, 12345654321,
    40000800004, 1000002000001, 1002003002001, 1004006004001, 1020304030201,
    1022325232201, 1024348434201, 1210024200121, 1212225222121, 1214428244121,
    1232346432321, 1234567654321, 4000008000004, 4004009004004, 100000020000001,
    10002000300020001, 10004000600040001, 10020210401202001, 10022212521222001,
    10024214841242001, 10201020402010201, 10203040504030201, 10205060806050201,
    10221232623212201, 10223252725232201, 10225272827252201, 121000242000121,
    12102024342020121, 12104024642040121, 12122232623222121, 12124234743242121,
    12126236863262121, 123003464300321, 1232234654322321, 1234434664344321,
    1232345665432321, 123456787654321, 400000080000004, 40004000900040004
]

def super_palindromes_in_range(left, right):
    """
    使用打表法查找范围内的超级回文数
    时间复杂度: O(log K), K 为预计算数组的大小
    空间复杂度: O(K)
    """
    count = 0
    left_num = int(left)
    right_num = int(right)

    for num in SUPER_PALINDROMES:
        if left_num <= num <= right_num:
            count += 1

    return count

def super_palindromes_in_range_enum(left, right):
    """
    使用枚举法查找范围内的超级回文数
    时间复杂度: O(√R * log R), 其中 R 为右边界
    空间复杂度: O(1)
    """
    left_num = int(left)
    right_num = int(right)

```

```

count = 0

# 计算种子的上界
limit = int(right_num ** 0.5) + 1
seed_limit = 10**5 # 对于 10^18 范围，种子只需到 10^5 级别

# 枚举所有可能的种子
for seed in range(1, seed_limit):
    # 生成偶数长度的回文数
    even_pal = even_enlarge(seed)
    square_even = even_pal * even_pal
    if square_even > right_num:
        continue
    if square_even >= left_num and is_palindrome(square_even):
        count += 1

    # 生成奇数长度的回文数
    odd_pal = odd_enlarge(seed)
    square_odd = odd_pal * odd_pal
    if square_odd > right_num:
        continue
    if square_odd >= left_num and is_palindrome(square_odd):
        count += 1

return count

```

```

# 补充训练题目的 Python 实现
def largest_palindrome_py(n):
    """
    LeetCode 479. 最大回文数乘积
    时间复杂度: O(10^2n)，但实际运行远快于理论值
    空间复杂度: O(1)
    """
    if n == 1:
        return 9
    max_num = 10**n - 1
    min_num = 10**n - 1

    for i in range(max_num, min_num - 1, -1):
        # 创建回文数
        s = str(i)
        palindrome = int(s + s[::-1])

```

```

# 检查是否可以分解为两个 n 位数的乘积
for j in range(max_num, int(palindrome**0.5) - 1, -1):
    if palindrome % j == 0:
        if min_num <= palindrome // j <= max_num:
            return palindrome % 1337
return -1

def palindrome_pairs_py(words):
    """
    LeetCode 336. 回文对
    时间复杂度: O(n * k^2), 其中 n 是单词数量, k 是单词的最大长度
    空间复杂度: O(n * k)
    """

    result = []
    word_map = {word: i for i, word in enumerate(words)}

    # 处理空字符串
    if "" in word_map:
        empty_idx = word_map[""]
        for i, word in enumerate(words):
            if i != empty_idx and word == word[::-1]:
                result.append([i, empty_idx])
                result.append([empty_idx, i])

    for i, word in enumerate(words):
        # 检查整个单词的逆序
        rev_word = word[::-1]
        if rev_word in word_map and word_map[rev_word] != i:
            result.append([i, word_map[rev_word]])

    # 分割单词为前缀和后缀
    for j in range(1, len(word)):
        prefix = word[:j]
        suffix = word[j:]

        # 前缀是回文, 检查后缀的逆序
        if prefix == prefix[::-1]:
            rev_suffix = suffix[::-1]
            if rev_suffix in word_map and word_map[rev_suffix] != i:
                result.append([word_map[rev_suffix], i])

        # 后缀是回文, 检查前缀的逆序
        if suffix == suffix[::-1]:
            rev_prefix = prefix[::-1]
            if rev_prefix in word_map and word_map[rev_prefix] != i:
                result.append([word_map[rev_prefix], i])

```

```

        rev_prefix = prefix[::-1]
        if rev_prefix in word_map and word_map[rev_prefix] != i:
            result.append([i, word_map[rev_prefix]])

    return result
*/



/***
 * 超级回文数算法的 C++实现
 */
/*
#include <iostream>
#include <string>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <cmath>

using namespace std;

// 判断一个数是否是回文数
bool isPalindrome(long long x) {
    if (x < 0) return false;
    long long original = x;
    long long reversedNum = 0;
    while (x > 0) {
        reversedNum = reversedNum * 10 + x % 10;
        x = x / 10;
    }
    return original == reversedNum;
}

// 根据种子扩充到偶数长度的回文数字
long long evenEnlarge(long long seed) {
    long long ans = seed;
    long long temp = seed;
    while (temp > 0) {
        ans = ans * 10 + temp % 10;
        temp = temp / 10;
    }
    return ans;
}

```

```

// 根据种子扩充到奇数长度的回文数字
long long oddEnlarge(long long seed) {
    long long ans = seed;
    long long temp = seed / 10;
    while (temp > 0) {
        ans = ans * 10 + temp % 10;
        temp = temp / 10;
    }
    return ans;
}

// 预计算的超级回文数
vector<long long> SUPER_PALINDROMES = {
    1, 4, 9, 121, 484, 10201, 12321, 14641, 40804, 44944, 1002001, 1234321,
    4008004, 100020001, 102030201, 104060401, 121242121, 123454321, 125686521,
    400080004, 404090404, 10000200001, 10221412201, 12102420121, 12345654321,
    40000800004, 1000002000001, 1002003002001, 1004006004001, 1020304030201,
    1022325232201, 1024348434201, 1210024200121, 1212225222121, 1214428244121,
    1232346432321, 1234567654321, 4000008000004, 4004009004004, 100000020000001,
    10002000300020001, 10004000600040001, 10020210401202001, 10022212521222001,
    10024214841242001, 10201020402010201, 10203040504030201, 10205060806050201,
    10221232623212201, 10223252725232201, 10225272827252201, 121000242000121,
    12102024342020121, 12104024642040121, 12122232623222121, 12124234743242121,
    12126236863262121, 123003464300321, 1232234654322321, 1234434664344321,
    1232345665432321, 123456787654321, 400000080000004, 40004000900040004
};

// 使用打表法查找范围内的超级回文数
int superPalindromesInRange(string left, string right) {
    long long leftNum = stoll(left);
    long long rightNum = stoll(right);
    int count = 0;

    for (long long num : SUPER_PALINDROMES) {
        if (leftNum <= num && num <= rightNum) {
            count++;
        }
    }

    return count;
}

// 使用枚举法查找范围内的超级回文数

```

```

int superPalindromesInRangeEnum(string left, string right) {
    long long leftNum = stoll(left);
    long long rightNum = stoll(right);
    int count = 0;

    // 枚举所有可能的种子
    const long long SEED_LIMIT = 100000; // 对于 10^18 范围，种子只需到 10^5 级别
    for (long long seed = 1; seed < SEED_LIMIT; seed++) {
        // 生成偶数长度的回文数
        long long evenPal = evenEnlarge(seed);
        long long squareEven = evenPal * evenPal;
        if (squareEven > rightNum) {
            continue;
        }
        if (squareEven >= leftNum && isPalindrome(squareEven)) {
            count++;
        }
    }

    // 生成奇数长度的回文数
    long long oddPal = oddEnlarge(seed);
    long long squareOdd = oddPal * oddPal;
    if (squareOdd > rightNum) {
        continue;
    }
    if (squareOdd >= leftNum && isPalindrome(squareOdd)) {
        count++;
    }
}

return count;
}

// 补充训练题目的 C++ 实现
int largestPalindrome(int n) {
    if (n == 1) return 9;
    long long maxNum = pow(10, n) - 1;
    long long minNum = pow(10, n - 1);

    for (long long i = maxNum; i >= minNum; i--) {
        // 创建回文数
        string s = to_string(i);
        string rev_s = s;
        reverse(rev_s.begin(), rev_s.end());

```

```

long long palindrome = stoll(s + rev_s);

// 检查是否可以分解为两个 n 位数的乘积
for (long long j = maxNum; j * j >= palindrome; j--) {
    if (palindrome % j == 0) {
        long long factor = palindrome / j;
        if (factor >= minNum && factor <= maxNum) {
            return palindrome % 1337;
        }
    }
}
return -1;
}

// 判断字符串是否是回文
bool isPalindromeString(const string& s) {
    int left = 0, right = s.size() - 1;
    while (left < right) {
        if (s[left++] != s[right--]) {
            return false;
        }
    }
    return true;
}

// 反转字符串
string reverseString(const string& s) {
    string rev = s;
    reverse(rev.begin(), rev.end());
    return rev;
}

// LeetCode 336. 回文对
vector<vector<int>> palindromePairs(vector<string>& words) {
    vector<vector<int>> result;
    unordered_map<string, int> wordMap;

    // 存储单词及其索引
    for (int i = 0; i < words.size(); i++) {
        wordMap[words[i]] = i;
    }
}

```

```

// 处理空字符串
if (wordMap.find("") != wordMap.end()) {
    int emptyIdx = wordMap[""];
    for (int i = 0; i < words.size(); i++) {
        if (i != emptyIdx && isPalindromeString(words[i])) {
            result.push_back({i, emptyIdx});
            result.push_back({emptyIdx, i});
        }
    }
}

for (int i = 0; i < words.size(); i++) {
    const string& word = words[i];

    // 检查整个单词的逆序
    string reversed = reverseString(word);
    if (wordMap.find(reversed) != wordMap.end() && wordMap[reversed] != i) {
        result.push_back({i, wordMap[reversed]});
    }
}

// 分割单词为前缀和后缀
for (int j = 1; j < word.size(); j++) {
    string prefix = word.substr(0, j);
    string suffix = word.substr(j);

    // 前缀是回文，检查后缀的逆序
    if (isPalindromeString(prefix)) {
        string revSuffix = reverseString(suffix);
        if (wordMap.find(revSuffix) != wordMap.end() && wordMap[revSuffix] != i) {
            result.push_back({wordMap[revSuffix], i});
        }
    }
}

// 后缀是回文，检查前缀的逆序
if (isPalindromeString(suffix)) {
    string revPrefix = reverseString(prefix);
    if (wordMap.find(revPrefix) != wordMap.end() && wordMap[revPrefix] != i) {
        result.push_back({i, wordMap[revPrefix]});
    }
}
}

```

```
    return result;
}
*/
/***
 * 超级回文数与回文序列生成的算法技巧总结
 *
 * 算法设计与策略选择:
 * 1. 问题分析框架:
 *     - 超级回文数的定义特征: 双重回文性质 (自身回文且为回文数的平方)
 *     - 搜索空间优化: 从平方根而非原始数入手, 大幅减少搜索范围
 *     - 对称性利用: 利用回文数的对称性特征构造候选解
 *
 * 2. 算法选择原则:
 *     - 单次查询场景: 枚举法更节省内存, 实现简单
 *     - 多次查询场景: 打表法性能最优, 时间复杂度接近  $O(1)$ 
 *     - 大数据范围: 需要考虑数值溢出问题和效率优化
 *
 * 回文数生成技术:
 * 1. 种子生成法:
 *     - 种子到完整回文数的转换策略
 *     - 偶数长度与奇数长度回文数的不同生成方式
 *     - 种子范围的数学推导与优化 (对于  $10^{18}$  范围, 种子仅需到  $10^5$  级别)
 *
 * 2. 高效实现技巧:
 *     - 数学方法避免字符串转换开销
 *     - 使用位移和取模操作进行位提取
 *     - 位运算优化 (适用于某些特定场景)
 *
 * 复杂度分析与优化:
 * 1. 时间复杂度深入分析:
 *     - 枚举法:  $O(\sqrt{R} * \log R)$ , 其中  $\log R$  为回文判断的时间
 *     - 打表法: 预处理  $O(K * \log K)$ , 查询  $O(\log K)$ 
 *     - 种子生成法的实际复杂度远低于理论上界
 *
 * 2. 空间复杂度考量:
 *     - 枚举法:  $O(1)$ , 常数额外空间
 *     - 打表法:  $O(K)$ , 其中  $K$  约为 70, 空间消耗极小
 *     - 权衡空间与时间的最佳平衡点
 *
 * 高级优化技术:
 * 1. 上下界剪枝:
 *     - 快速判断种子生成的回文数平方是否可能在范围内
```

- * - 数学上界计算避免不必要的计算
- *
- * 2. 并行与并发优化:
 - 多线程并行生成和验证回文数
 - 任务分割策略与负载均衡
 - 原子操作确保结果正确性
- *
- * 3. 缓存与预算策略:
 - LRU 缓存设计与实现
 - 区间查询结果缓存
 - 动态规划预处理回文属性
- *
- * 工程化最佳实践:
- * 1. 健壮性设计:
 - 大数输入处理（字符串转长整型）
 - 溢出检测与处理机制
 - 边界条件全面测试
- *
- * 2. 代码结构优化:
 - 模块化设计与功能解耦
 - 策略模式实现不同算法切换
 - 接口抽象与实现分离
- *
- * 3. 性能监控与调优:
 - 热点代码识别与优化
 - 内存使用分析
 - 执行时间基准测试
- *
- * 调试与测试技术:
- * 1. 测试用例设计:
 - 边界值测试: 最小/最大输入、空范围
 - 特殊情况测试: 无超级回文数的范围
 - 性能测试: 大规模输入下的执行效率
- *
- * 2. 调试技巧:
 - 中间结果可视化
 - 性能瓶颈分析
 - 断言与防御式编程
- *
- * 跨语言实现比较:
- * 1. 语言特性影响:
 - C++: 数值运算性能最佳, 无符号整数处理更灵活
 - Java: 大整数支持良好, 线程安全特性

- * - Python: 内置大整数, 代码简洁但性能较低
- * - JavaScript: 需要特殊处理大整数, 精度问题需注意
- *
- * 2. 实现策略差异:
 - 不同语言的哈希表/字典实现效率
 - 数值类型范围与溢出处理机制
 - 并行编程模型与性能
- *
- * 与现代技术的融合:
 - 1. 与机器学习的结合:
 - 模式识别: 自动发现回文数生成模式
 - 预测模型: 预测特定范围内超级回文数的分布
 - 启发式搜索: 基于学习的优化搜索策略
 - *
 - 2. 高性能计算应用:
 - GPU 加速回文数生成与验证
 - 分布式计算大规模范围查询
 - SIMD 指令集优化数值运算
 - *
 - 3. 密码学与安全应用:
 - 超级回文数在某些加密算法中的应用
 - 数论研究中的特殊性质利用
 - 安全协议中的数学基础
 - *
- * 进阶研究方向:
 - 1. 数学理论扩展:
 - 超级回文数的分布规律研究
 - 推广到其他数论函数的超级形式
 - 多维回文结构探索
 - *
 - 2. 算法创新:
 - 更快的回文数生成算法
 - 更高效的范围查询方法
 - 新型数据结构支持
 - *
 - 3. 实际应用拓展:
 - 文本处理中的模式匹配
 - 生物序列分析中的回文结构识别
 - 网络安全中的应用场景
- */

{}

=====

文件: Code06_SuperPalindromesII.py

=====

超级回文数 II 问题 - Python 版本

问题描述:

如果一个正整数自身是回文数，而且它也是一个回文数的平方，那么我们称这个数为超级回文数。

给定两个正整数 L 和 R (以字符串形式表示)，返回包含在范围 [L, R] 中的超级回文数的最小值和最大值。如果不存在超级回文数，返回 [-1, -1]。

算法思路:

方法 1: 枚举法 - 通过生成回文数来优化搜索

方法 2: 打表法 - 预计算所有可能的超级回文数

时间复杂度分析:

- 枚举法: $O(\sqrt{R} * \log R)$, 其中 \sqrt{R} 为平方根范围, $\log R$ 为回文判断时间
- 打表法: 预处理 $O(K * \log K)$, 查询 $O(\log K)$, 其中 K 为超级回文数个数 (约 70)

空间复杂度分析:

- 枚举法: $O(1)$, 常数额外空间
- 打表法: $O(K)$, 存储预计算的超级回文数

工程化考量:

1. 大数处理: 使用 Python 内置大整数支持
2. 边界处理: 处理 L 和 R 的边界情况
3. 性能优化: 选择合适的算法策略
4. 可测试性: 设计全面的测试用例

=====

```
from typing import List, Tuple
```

```
class SuperPalindromesIISolver:
```

```
    def __init__(self):
```

```
        # 预计算的超级回文数数组
```

```
        self.super_palindromes = [
```

```
            1, 4, 9, 121, 484, 10201, 12321, 14641, 40804, 44944,
            1002001, 1234321, 4008004, 100020001, 102030201, 104060401,
            121242121, 123454321, 125686521, 400080004, 404090404,
            10000200001, 10221412201, 12102420121, 12345654321, 40000800004,
            1000002000001, 1002003002001, 1004006004001, 1020304030201,
            1022325232201, 1024348434201, 1210024200121, 1212225222121,
            1214428244121, 1232346432321, 1234567654321, 4000008000004,
```

```
4004009004004, 100000020000001, 100220141022001, 102012040210201,
102234363432201, 121000242000121, 121242363242121, 123212464212321,
123456787654321, 400000080000004, 10000000200000001, 10002000300020001,
10004000600040001, 10020210401202001, 10022212521222001, 10024214841242001,
10201020402010201, 10203040504030201, 10205060806050201, 10221432623412201,
10223454745432201, 12100002420000121, 12102202520220121, 12104402820440121,
12122232623222121, 12124434743442121, 12321024642012321, 12323244744232321,
12343456865434321, 12345678987654321, 40000000800000004, 40004000900040004
]
```

```
def superpalindromes_in_range(self, left: str, right: str) -> List[int]:
```

```
"""
```

方法 1：打表法

使用预计算的超级回文数列表进行查询

```
"""
```

```
L = int(left)
```

```
R = int(right)
```

```
min_val = float('inf')
```

```
max_val = float('-inf')
```

```
found = False
```

```
# 查找范围内的最小值和最大值
```

```
for num in self.super_palindromes:
```

```
    if L <= num <= R:
```

```
        found = True
```

```
        if num < min_val:
```

```
            min_val = num
```

```
        if num > max_val:
```

```
            max_val = num
```

```
if not found:
```

```
    return [-1, -1]
```

```
return [int(min_val), int(max_val)]
```

```
def superpalindromes_in_range_generate(self, left: str, right: str) -> List[int]:
```

```
"""
```

方法 2：枚举生成法

通过生成回文数种子来构造超级回文数

```
"""
```

```
L = int(left)
```

```
R = int(right)
```

```
min_val = float('inf')
```

```

max_val = float('-inf')
found = False

seed_limit = 100000 # 10^5

# 枚举所有可能的种子
for seed in range(1, seed_limit):
    # 生成偶数长度的回文数
    even_pal = self.even_enlarge(seed)
    square_even = even_pal * even_pal

    # 检查是否在范围内且是回文数
    if L <= square_even <= R and self.is_palindrome(square_even):
        found = True
        if square_even < min_val:
            min_val = square_even
        if square_even > max_val:
            max_val = square_even

    # 生成奇数长度的回文数
    odd_pal = self.odd_enlarge(seed)
    square_odd = odd_pal * odd_pal

    # 检查是否在范围内且是回文数
    if L <= square_odd <= R and self.is_palindrome(square_odd):
        found = True
        if square_odd < min_val:
            min_val = square_odd
        if square_odd > max_val:
            max_val = square_odd

    # 如果生成的平方已经超过 R, 可以提前终止
    if square_even > R and square_odd > R:
        break

if not found:
    return [-1, -1]

return [int(min_val), int(max_val)]
```

def even_enlarge(self, seed: int) -> int:

""""

根据种子扩充到偶数长度的回文数字

例如: seed=123, 返回 123321

"""

```
ans = seed
temp = seed
while temp > 0:
    ans = ans * 10 + temp % 10
    temp //= 10
return ans
```

def odd_enlarge(self, seed: int) -> int:

"""

根据种子扩充到奇数长度的回文数字

例如: seed=123, 返回 12321

"""

```
ans = seed
temp = seed // 10
while temp > 0:
    ans = ans * 10 + temp % 10
    temp //= 10
return ans
```

def is_palindrome(self, x: int) -> bool:

"""

判断一个数是否是回文数

使用数学方法避免字符串转换

"""

```
if x < 0:
    return False
if x < 10:
    return True
```

original = x

reversed_num = 0

while x > 0:

```
    reversed_num = reversed_num * 10 + x % 10
    x //= 10
```

return original == reversed_num

补充训练题目 - Python 实现

def largest_palindrome_product(n: int) -> int:

```
"""
```

LeetCode 479. 最大回文数乘积

给定一个整数 n，返回可表示为两个 n 位整数乘积的最大回文整数

```
"""
```

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

```
    return 9
```

```
max_num = 10**n - 1
```

```
min_num = 10**n - 1)
```

```
for i in range(max_num, min_num - 1, -1):
```

```
    # 创建回文数
```

```
    s = str(i)
```

```
    palindrome = int(s + s[::-1])
```

```
    # 检查是否可以分解为两个 n 位数的乘积
```

```
    for j in range(max_num, int(palindrome**0.5) - 1, -1):
```

```
        if palindrome % j == 0:
```

```
            factor = palindrome // j
```

```
            if min_num <= factor <= max_num:
```

```
                return palindrome
```

```
return -1
```

```
def count_super_palindromes(left: str, right: str) -> int:
```

```
"""
```

LeetCode 906. 超级回文数（统计版本）

统计范围内的超级回文数个数

```
"""
```

```
L = int(left)
```

```
R = int(right)
```

```
count = 0
```

```
super_palindromes = [
```

```
    1, 4, 9, 121, 484, 10201, 12321, 14641, 40804, 44944,
```

```
    1002001, 1234321, 4008004, 100020001, 102030201, 104060401,
```

```
    121242121, 123454321, 125686521, 400080004, 404090404,
```

```
    1000020001, 10221412201, 12102420121, 12345654321, 40000800004
```

```
]
```

```
for num in super_palindromes:
```

```
    if L <= num <= R:
```

```
        count += 1
```

```
    return count

def is_palindrome_big(s: str) -> bool:
    """
    LeetCode 9. 回文数（大数版本）
    支持大数判断的回文数函数
    """
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

def generate_palindromes_in_range(start: int, end: int) -> List[int]:
    """
    生成指定范围内的所有回文数
    """
    result = []

    for i in range(start, end + 1):
        if str(i) == str(i)[::-1]:
            result.append(i)

    return result

def next_palindrome(n: int) -> int:
    """
    寻找大于 n 的最小回文数
    """
    def is_palindrome(x: int) -> bool:
        s = str(x)
        return s == s[::-1]

    while True:
        n += 1
        if is_palindrome(n):
            return n

    def prev_palindrome(n: int) -> int:
        """

```

```
寻找小于 n 的最大回文数
"""
def is_palindrome(x: int) -> bool:
    s = str(x)
    return s == s[::-1]

while n > 0:
    n -= 1
    if is_palindrome(n):
        return n
return 0

def palindrome_generator(limit: int):
    """
    回文数生成器
    生成小于等于 limit 的所有回文数
    """
    for i in range(1, limit + 1):
        if str(i) == str(i)[::-1]:
            yield i

# 测试函数
def test_super_palindromes_ii():
    """测试函数"""
    solver = SuperPalindromesIISolver()

    # 测试用例 1
    left1 = "4"
    right1 = "1000"
    result1 = solver.superpalindromes_in_range(left1, right1)
    result1_gen = solver.superpalindromes_in_range_generate(left1, right1)

    print("测试用例 1:")
    print(f"范围: [{left1}, {right1}]")
    print(f"打表法结果: [{result1[0]}, {result1[1]}]")
    print(f"生成法结果: [{result1_gen[0]}, {result1_gen[1]}]")
    print()

    # 测试用例 2: 无超级回文数的情况
    left2 = "10000000000000000000"
    right2 = "10000000000000000000"
    result2 = solver.superpalindromes_in_range(left2, right2)
```

```
print("测试用例 2:")
print(f"范围: [{left2}, {right2}]")
print(f"结果: [{result2[0]}, {result2[1]}]")
print()

# 测试补充题目
print("== 补充训练题目测试 ==")

# 测试最大回文数乘积
print(f"最大回文数乘积(n=2): {largest_palindrome_product(2)}")

# 测试超级回文数统计
print(f"超级回文数统计: 范围[4, 1000] -> {count_super_palindromes('4', '1000')}")

# 测试大数回文判断
big_num = "12345678987654321"
print(f"大数回文判断: {big_num} -> {'是' if is_palindrome_big(big_num) else '否'}")

# 测试回文数生成
palindromes = generate_palindromes_in_range(100, 200)
print(f"回文数生成(100-200): 数量={len(palindromes)}")

# 测试下一个回文数
n = 123
print(f"下一个回文数: {n} -> {next_palindrome(n)}")

# 测试上一个回文数
print(f"上一个回文数: {n} -> {prev_palindrome(n)}")

# 测试回文数生成器
print("回文数生成器(1-100):", end=" ")
count = 0
for p in palindrome_generator(100):
    count += 1
print(f"数量={count}")

if __name__ == "__main__":
    test_super_palindromes_ii()

"""

算法技巧总结 - Python 版本
```

核心概念:

1. 回文数生成技术:
 - 种子生成法: 通过种子数字构造回文数
 - 对称性利用: 利用回文数的对称特征
 - 数学方法: 避免字符串转换的开销
2. Python 特有优势:
 - 内置大整数支持: 无需担心数值溢出
 - 字符串操作简便: `[::-1]`快速反转字符串
 - 生成器表达式: 节省内存空间
3. 算法选择策略:
 - 小范围查询: 枚举法更节省内存
 - 多次查询: 打表法性能最优
 - 大数据范围: 需要数学优化和边界处理

调试技巧:

1. 使用 `pdb` 进行调试
2. 打印中间状态变量
3. 使用 `assert` 进行条件验证

性能优化:

1. 避免不必要的字符串转换
2. 使用局部变量减少属性查找
3. 利用 Python 内置函数的高效实现

工程化实践:

1. 类型注解: 提高代码可读性
2. 异常处理: 确保程序健壮性
3. 单元测试: 保证代码质量
4. 文档字符串: 提供清晰的接口说明

边界情况处理:

1. 空范围: 返回 `[-1, -1]`
2. 单个数字: 检查该数字是否是超级回文数
3. 大数范围: 使用适当的算法避免性能问题
4. 边界值: 测试最小/最大可能的值

"""

=====