

=====

文件夹: class116_Manacher_And_Z_Function

=====

[Markdown 文件]

=====

文件: algorithm_comprehensive_summary.md

=====

Manacher 算法与 Z 函数全面总结

一、算法核心原理

1.1 Manacher 算法

****核心思想**:** 利用回文串的对称性避免重复计算，实现线性时间复杂度。

****关键步骤**:**

1. ****预处理**:** 在字符间插入特殊字符，统一处理奇偶长度回文
2. ****对称性优化**:** 利用已计算的回文信息加速新位置的计算
3. ****边界维护**:** 动态维护当前最右回文边界

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

1.2 Z 函数（扩展 KMP）

****核心思想**:** 计算字符串每个后缀与整个字符串的最长公共前缀。

****关键步骤**:**

1. ****边界维护**:** 维护当前匹配的最右边界
2. ****对称性利用**:** 利用已计算的 Z 值加速新位置的计算
3. ****暴力扩展**:** 当无法利用对称性时进行暴力匹配

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

二、题目分类与解题策略

2.1 基础题目（必须掌握）

LeetCode 5. 最长回文子串

- ****解题思路**:** Manacher 算法直接应用
- ****时间复杂度**:** $O(n)$
- ****空间复杂度**:** $O(n)$
- ****关键点**:** 预处理字符串，维护回文半径数组

LeetCode 647. 回文子串计数

- **解题思路**: Manacher 算法 + 数学公式计算
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **关键点**: 利用半径计算贡献数量 $^{\lceil (radius[i] + 1) / 2 \rceil}$

LeetCode 214. 最短回文串

- **解题思路**: KMP 算法 (LPS 数组)
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **关键点**: 构造 $s + "#" + reverse(s)$, 计算 LPS

2.2 中级题目 (应用扩展)

LeetCode 336. 回文对

- **解题思路**: 字典树 + 回文检查
- **时间复杂度**: $O(n * k^2)$
- **空间复杂度**: $O(n * k)$
- **关键点**: 预处理单词, 检查前缀后缀回文性

LeetCode 131/132. 分割回文串

- **解题思路**: 动态规划 + 回文预处理
- **时间复杂度**: $O(n^2)$
- **空间复杂度**: $O(n^2)$
- **关键点**: 预处理回文信息, 优化状态转移

2.3 高级题目 (竞赛级别)

洛谷 P1659 拉拉队排练

- **解题思路**: Manacher + 统计 + 快速幂
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **关键点**: 统计奇数长度回文, 计算乘积

洛谷 P4555 最长双回文串

- **解题思路**: Manacher + 前后缀预处理
- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **关键点**: 计算每个分割点的左右最长回文

SPOJ PALIN 下一个回文数

- **解题思路**: 数字处理 + 回文构造

- **时间复杂度**: $O(n)$
- **空间复杂度**: $O(n)$
- **关键点**: 处理进位，构造对称数字

三、工程化考量

3.1 异常处理与鲁棒性

```
```java
// 参数验证
if (s == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
}

// 边界情况处理
if (s.length() <= 1) {
 return s;
}
````
```

3.2 性能优化策略

1. **预分配内存**: 避免动态扩容开销
2. **局部变量优化**: 减少数组访问次数
3. **边界检查前置**: 提前处理特殊情况
4. **算法常数优化**: 减少不必要的操作

3.3 多语言实现对比

Java 实现特点

- **字符串处理**: String 不可变，使用 StringBuilder
- **数组操作**: 下标从 0 开始，注意边界检查
- **异常处理**: 完善的异常机制

Python 实现特点

- **动态类型**: 无需声明变量类型
- **字符串切片**: 操作方便但需注意效率
- **内存管理**: 自动垃圾回收

C++实现特点

- **指针操作**: 直接内存操作，性能高
- **STL 容器**: vector 等使用灵活
- **性能优势**: 处理大规模数据更快

四、调试与问题定位

```
#### 4.1 调试技巧
```java
// 中间过程打印
System.out.println("i=" + i + ", c=" + c + ", r=" + r);

// 断言验证
assert len > 0 : "回文半径必须为正整数";

// 性能监控
long startTime = System.nanoTime();
// ... 算法执行 ...
long endTime = System.nanoTime();
```

```

4.2 常见问题与解决方案

内存溢出

- **问题**: 处理长字符串时内存不足
- **解决**: 预分配足够内存，动态调整数组大小

性能瓶颈

- **问题**: 大数据量下性能下降
- **解决**: 启用性能监控，分析瓶颈位置

算法正确性

- **问题**: 边缘情况结果不正确
- **解决**: 运行单元测试，检查边界条件

五、学习路径建议

5.1 初级阶段（1-2 周）

1. **理解算法原理**: 掌握 Manacher 和 Z 函数的核心思想
2. **熟练实现**: 能够独立实现两种算法
3. **基础题目练习**: 解决 LeetCode 5、647、214 等题目

5.2 中级阶段（2-3 周）

1. **应用场景识别**: 识别适合使用这些算法的问题
2. **复杂度分析**: 掌握时间和空间复杂度分析方法
3. **题目变种**: 解决 Codeforces、SPOJ 等平台的进阶题目

5.3 高级阶段（3-4 周）

1. **优化能力**: 根据具体场景优化算法实现

2. **跨语言实践**: 在多种编程语言中实现算法
3. **工程应用**: 将算法应用到实际项目中

六、资源推荐

6.1 在线评测平台

1. **LeetCode**: 提供大量相关题目
2. **Codeforces**: 包含挑战性的字符串算法题目
3. **AtCoder**: 日本算法竞赛平台
4. **SPOJ**: 经典字符串处理问题
5. **洛谷**: 中文算法学习平台

6.2 学习资料

1. **算法导论**: 深入理解算法理论基础
2. **算法竞赛入门经典**: 包含字符串算法详解
3. **编程珠玑**: 提供算法设计思路和技巧
4. **OI Wiki**: 详细的算法讲解和实现
5. **CP-Algorithms**: 高质量的算法教程网站

七、实战技巧

7.1 笔试技巧

1. **模板准备**: 提前准备好算法模板代码
2. **边界处理**: 特别注意空字符串、单字符等情况
3. **性能优化**: 注意时间复杂度的优化

7.2 面试技巧

1. **算法解释**: 清晰解释算法原理和复杂度
2. **代码实现**: 写出清晰、可读的代码
3. **问题分析**: 分析问题的适用算法和优化空间

八、代码模板

8.1 Manacher 算法模板 (Java)

```
```java
public static int manacher(String str) {
 if (str == null || str.isEmpty()) return 0;

 // 预处理
 char[] processed = preprocess(str);
 int n = processed.length;
 int[] p = new int[n];
```

```

int center = 0, right = 0, maxLen = 0;

for (int i = 1; i < n - 1; i++) {
 int mirror = 2 * center - i;

 if (i < right) {
 p[i] = Math.min(right - i, p[mirror]);
 }

 while (processed[i + p[i] + 1] == processed[i - p[i] - 1]) {
 p[i]++;
 }

 if (i + p[i] > right) {
 center = i;
 right = i + p[i];
 }

 maxLen = Math.max(maxLen, p[i]);
}

return maxLen;
}
```

```

```

### 8.2 Z 函数模板 (Python)
```python
def z_function(s: str) -> List[int]:
 n = len(s)
 z = [0] * n
 z[0] = n

 l, r = 0, 0
 for i in range(1, n):
 if i <= r:
 z[i] = min(r - i + 1, z[i - 1])

 while i + z[i] < n and s[z[i]] == s[i + z[i]]:
 z[i] += 1

 if i + z[i] - 1 > r:
 l = i
 r = i + z[i] - 1

```

```
 return z
```

## ## 九、总结

通过系统学习和大量练习，您将能够：

1. \*\*熟练掌握\*\*Manacher 算法和 Z 函数的原理和实现
2. \*\*灵活应用\*\*这些算法解决各种字符串处理问题
3. \*\*优化改进\*\*算法实现，提高代码效率和鲁棒性
4. \*\*工程实践\*\*将算法应用到实际项目中

记住：算法学习是一个持续的过程，需要不断的练习和总结。祝您学习顺利！

---  
\*最后更新：2025 年 10 月 25 日\*

\*作者：算法学习助手\*

=====

文件：algorithm\_summary.md

=====

# Manacher 算法与 Z 函数（扩展 KMP）总结

### ## 1. Manacher 算法

#### ### 1.1 算法概述

Manacher 算法是一种用于在线性时间内查找字符串中所有回文子串的高效算法，由 Glenn K. Manacher 在 1975 年提出。该算法主要用于解决最长回文子串问题，时间复杂度为  $O(n)$ 。

#### ### 1.2 算法原理

1. \*\*预处理\*\*：在原字符串的每个字符之间插入特殊字符'#'，并在首尾也添加'#'，这样可以将奇数长度和偶数长度的回文串统一处理为奇数长度的回文串。
2. \*\*利用回文串的对称性\*\*：维护当前最右回文边界  $r$  和对应的中心  $c$ ，通过已计算的信息加速新位置的计算。
3. \*\*动态扩展\*\*：对于每个位置，首先利用对称性获得一个初始回文半径，然后在此基础上尝试向两边扩展。

#### ### 1.3 核心思想

- 维护最右回文边界：通过记录当前已知的最右回文子串的右边界和中心，避免重复计算。
- 利用对称性：对于当前中心  $i$ ，如果它在已知的最右回文子串内，则可以利用对称点  $2*c - i$  的信息来优化计算。

#### ### 1.4 时间复杂度分析

- 时间复杂度:  $O(n)$ , 其中  $n$  为字符串长度。每个字符最多被访问常数次。
- 空间复杂度:  $O(n)$ , 需要额外数组存储每个位置的回文半径。

#### #### 1.5 适用场景

1. 求最长回文子串 (LeetCode 5)
2. 统计回文子串个数 (LeetCode 647)
3. 构造最短回文串 (LeetCode 214)
4. 求两个不重叠回文子串长度乘积的最大值 (LeetCode 1960)
5. 洛谷 P3805 【模板】manacher
6. UVa 11475 – Extend to Palindrome
7. Codeforces 1326D2 – Prefix–Suffix Palindrome
8. HackerRank – Palindromic Substrings
9. AcWing 141. 周期
10. POJ 3240 – 回文串
11. LeetCode 336. 回文对
12. LeetCode 131. 分割回文串
13. LeetCode 132. 分割回文串 II

#### #### 1.6 算法技巧

1. \*\*预处理技巧\*\*: 通过插入特殊字符'#', 将奇偶长度回文统一处理。
2. \*\*边界处理\*\*: 维护最右回文边界, 利用对称性减少重复计算。
3. \*\*扩展优化\*\*: 从已知半径开始扩展, 避免从 1 开始的低效扩展。

## ## 2. Z 函数 (扩展 KMP)

#### #### 2.1 算法概述

Z 函数 (也称为扩展 KMP 算法) 用于计算字符串  $s$  的每个后缀与整个字符串  $s$  的最长公共前缀 (LCP) 长度。该算法在字符串匹配、周期检测等问题中有广泛应用。

#### #### 2.2 算法原理

1. \*\*维护匹配区间\*\*: 维护一个匹配区间  $[l, r]$ , 表示当前已知的最右匹配区间。
2. \*\*利用已计算信息\*\*: 对于当前位置  $i$ , 如果  $i \leq r$ , 可以利用已计算的信息优化。
3. \*\*对称性优化\*\*: 利用对称性,  $z[i]$  至少为  $\min(r - i + 1, z[i - 1])$ 。
4. \*\*继续扩展\*\*: 在此基础之上继续向右扩展匹配。

#### #### 2.3 核心思想

- 区间维护: 通过维护匹配区间, 避免重复计算已匹配的部分。
- 对称性利用: 利用字符串的对称性, 快速获得初始匹配长度。

#### #### 2.4 时间复杂度分析

- 时间复杂度:  $O(n)$ , 其中  $n$  为字符串长度。每个字符最多被访问常数次。
- 空间复杂度:  $O(n)$ , 需要额外数组存储 Z 函数值。

### ### 2.5 适用场景

1. 字符串匹配问题
2. 寻找既是前缀又是后缀的子串 (Codeforces 126B)
3. 计算字符串得分和 (LeetCode 2223)
4. 计算恢复初始状态所需时间 (LeetCode 3031)
5. 模式匹配相关问题
6. 洛谷 P5410 【模板】扩展 KMP/exKMP (Z 函数)
7. SPOJ - Pattern Find
8. HackerEarth - String Similarity
9. AtCoder ABC141E - Who Says a Pun?
10. USACO 2011 November Contest, Bronze - Cow Photographs
11. 牛客网 NC15051 - 字符串的匹配

### ### 2.6 算法技巧

1. \*\*区间维护\*\*: 通过维护匹配区间  $[l, r]$ , 避免重复计算。
2. \*\*对称性优化\*\*: 利用对称点的信息快速获得初始匹配长度。
3. \*\*扩展策略\*\*: 从已知匹配长度开始扩展, 提高效率。

## ## 3. 算法对比

特性	Manacher 算法	Z 函数 (扩展 KMP)
主要用途	回文串处理	字符串匹配、LCP 计算
核心思想	利用回文对称性	利用匹配区间和对称性
时间复杂度	$O(n)$	$O(n)$
空间复杂度	$O(n)$	$O(n)$
预处理	插入特殊字符	无特殊预处理
典型应用	最长回文子串	字符串匹配

## ## 4. 工程化考量

### ### 4.1 异常处理

1. \*\*空输入处理\*\*: 检查输入字符串是否为空或 null。
2. \*\*边界条件\*\*: 处理字符串长度为 0、1 等特殊情况。
3. \*\*数组越界\*\*: 在扩展匹配时注意数组边界检查。

### ### 4.2 性能优化

1. \*\*避免重复计算\*\*: 利用已计算的信息, 避免重复匹配。
2. \*\*减少内存分配\*\*: 预分配数组空间, 避免动态扩容。
3. \*\*常数优化\*\*: 减少不必要的计算和内存访问。

### ### 4.3 代码可读性

1. \*\*变量命名\*\*: 使用有意义的变量名, 如 center、radius、left、right 等。

2. \*\*注释说明\*\*: 详细注释算法原理和关键步骤。
3. \*\*模块化设计\*\*: 将核心算法封装成独立函数，提高复用性。

## ## 5. 应用场景总结

### #### 5.1 Manacher 算法应用场景

1. \*\*回文检测\*\*: 判断字符串是否为回文串。
2. \*\*最长回文子串\*\*: 找到字符串中的最长回文子串。
3. \*\*回文计数\*\*: 统计字符串中回文子串的个数。
4. \*\*回文构造\*\*: 通过添加字符构造回文串。

### #### 5.2 Z 函数应用场景

1. \*\*字符串匹配\*\*: 在一个字符串中查找另一个字符串的所有出现位置。
2. \*\*周期检测\*\*: 检测字符串的周期性。
3. \*\*前缀后缀匹配\*\*: 查找既是前缀又是后缀的子串。
4. \*\*字符串压缩\*\*: 利用周期性进行字符串压缩。

## ## 6. 工程化与优化深入

### #### 6.1 多语言实现对比

#### #### Java 实现特点

- \*\*字符串处理\*\*: Java 中字符串是不可变的，需要频繁创建新字符串或使用 StringBuilder
- \*\*数组操作\*\*: 数组下标从 0 开始，需要注意边界检查
- \*\*异常处理\*\*: 提供了完善的异常机制，可用于边界条件处理
- \*\*性能优化\*\*: 使用 BufferedReader 和 PrintWriter 提高 I/O 效率

#### #### Python 实现特点

- \*\*动态类型\*\*: 无需提前声明变量类型，代码更简洁
- \*\*字符串处理\*\*: 字符串切片操作方便，但需要注意效率
- \*\*内存管理\*\*: 自动垃圾回收，但可能导致内存占用较高
- \*\*边界处理\*\*: 需要显式检查数组边界，避免越界错误

#### #### C++ 实现特点

- \*\*指针操作\*\*: 可以直接操作内存，性能更高
- \*\*STL 容器\*\*: vector 等容器使用灵活，但需要注意内存管理
- \*\*字符串处理\*\*: string 类提供了丰富的操作方法
- \*\*性能优势\*\*: 在处理大规模数据时通常比 Java 和 Python 更快

### ## 6.2 性能优化策略

#### #### 基础操作效率细节

1. \*\*内存预分配\*\*: 提前分配足够大的数组空间，避免动态扩容

2. \*\*减少函数调用\*\*: 关键循环中的函数调用会影响性能
3. \*\*局部变量优先\*\*: 使用局部变量而非成员变量，减少访问开销
4. \*\*避免重复计算\*\*: 缓存中间计算结果

#### #### 大规模数据优化

1. \*\*分批处理\*\*: 对于超大字符串，可以采用分块处理策略
2. \*\*并行计算\*\*: 在某些场景下可以考虑并行计算 Z 函数或 Manacher 算法的不同部分
3. \*\*内存压缩\*\*: 使用位操作或更紧凑的数据结构减少内存占用

#### #### 常数项优化

1. \*\*循环展开\*\*: 减少循环控制开销
2. \*\*减少分支预测失败\*\*: 优化条件判断顺序
3. \*\*缓存友好\*\*: 优化数据访问模式，提高缓存命中率

### ## 6.3 调试与问题定位

#### #### 中间过程打印

1. \*\*关键变量跟踪\*\*: 在算法执行过程中打印关键变量值
2. \*\*可视化辅助\*\*: 将算法执行过程可视化，帮助理解

#### #### 断言验证

1. \*\*前置条件检查\*\*: 验证输入参数的有效性
2. \*\*后置条件验证\*\*: 确保算法执行结果符合预期
3. \*\*不变量检查\*\*: 验证算法执行过程中的不变量

#### #### 性能分析

1. \*\*时间瓶颈识别\*\*: 使用性能分析工具找出耗时操作
2. \*\*内存使用监控\*\*: 监控内存占用，避免内存泄漏
3. \*\*退化情况测试\*\*: 测试极端输入下的性能表现

### ## 6.4 异常处理与鲁棒性

#### #### 非法输入处理

1. \*\*空字符串检查\*\*: 处理 null 或空字符串输入
2. \*\*边界情况处理\*\*: 处理长度为 0、1 的特殊情况
3. \*\*非法字符处理\*\*: 处理包含特殊字符的输入

#### #### 线程安全改造

1. \*\*无状态设计\*\*: 设计无状态的算法实现，便于多线程使用
2. \*\*同步机制\*\*: 需要时添加适当的同步机制
3. \*\*线程局部变量\*\*: 使用线程局部变量避免共享状态

#### #### 鲁棒性测试

1. \*\*极端输入测试\*\*: 测试空字符串、全相同字符等极端情况
2. \*\*边界值测试\*\*: 测试最大长度限制、最小长度等边界值
3. \*\*随机数据测试\*\*: 使用随机生成的数据进行大量测试

## ## 7. 高级应用与扩展

### ### 7.1 算法变种与扩展

#### #### Manacher 算法变种

1. \*\*最长双回文子串\*\*: 寻找由两个回文子串组成的最长子串
2. \*\*多模式回文匹配\*\*: 同时处理多个模式的回文匹配
3. \*\*动态维护回文信息\*\*: 在字符串动态变化时维护回文信息

#### #### Z 函数扩展应用

1. \*\*多模式字符串匹配\*\*: 使用 Z 函数处理多个模式串
2. \*\*字符串压缩\*\*: 利用周期性进行字符串压缩
3. \*\*近似字符串匹配\*\*: 允许一定数量的不匹配

### ### 7.2 与其他领域的结合

#### #### 机器学习应用

1. \*\*文本分类特征提取\*\*: 使用 Z 函数或 Manacher 算法提取字符串特征
2. \*\*异常检测\*\*: 检测不符合正常模式的字符串
3. \*\*序列比对\*\*: 在生物信息学中的 DNA 序列比对

#### #### 自然语言处理

1. \*\*形态分析\*\*: 分析词语的形态变化
2. \*\*文本分割\*\*: 基于回文或周期性进行文本分割
3. \*\*语言模型\*\*: 构建基于字符串周期性的简单语言模型

#### #### 计算机视觉

1. \*\*模式识别\*\*: 识别具有回文或周期性的模式
2. \*\*图像压缩\*\*: 利用图像中的重复模式进行压缩

### ### 7.3 工程化组件设计

#### #### 可复用接口设计

1. \*\*统一接口\*\*: 设计统一的字符串处理接口
2. \*\*配置选项\*\*: 提供灵活的配置选项适应不同需求
3. \*\*扩展点\*\*: 预留扩展点以便功能扩展

#### #### 文档化

1. \*\*API 文档\*\*: 详细的 API 使用说明

2. \*\*使用示例\*\*: 常见使用场景的示例代码
3. \*\*性能说明\*\*: 算法性能特性的详细说明

#### #### 单元测试

1. \*\*基础功能测试\*\*: 验证基本功能正确性
2. \*\*边界测试\*\*: 测试边界条件
3. \*\*性能测试\*\*: 测试在不同数据规模下的性能

### ## 8. 学习路径与资源

#### ### 8.1 学习建议

##### #### 掌握要点

1. \*\*理解算法原理\*\*: 深入理解 Manacher 算法和 Z 函数的核心思想。
2. \*\*熟练实现\*\*: 能够独立实现两种算法的代码。
3. \*\*应用场景识别\*\*: 能够识别适合使用这两种算法的问题场景。
4. \*\*复杂度分析\*\*: 掌握算法的时间和空间复杂度分析方法。
5. \*\*优化能力\*\*: 能够根据具体场景优化算法实现。
6. \*\*跨语言实践\*\*: 在多种编程语言中实现算法，理解语言特性影响。

##### #### 实践路径

1. \*\*基础实现\*\*: 实现基本的算法框架
2. \*\*题目练习\*\*: 解决各种相关算法题目
3. \*\*优化改进\*\*: 优化算法实现，提高性能
4. \*\*综合应用\*\*: 将算法应用到复杂问题中
5. \*\*教学分享\*\*: 向他人解释算法，加深理解

#### ### 8.2 推荐资源

##### #### 在线评测平台

1. \*\*LeetCode\*\*: 提供大量 Manacher 和 Z 函数相关题目
2. \*\*Codeforces\*\*: 包含更多挑战性的字符串算法题目
3. \*\*AtCoder\*\*: 日本算法竞赛平台，有质量很高的题目
4. \*\*SPOJ\*\*: 包含经典的字符串处理问题
5. \*\*洛谷\*\*: 中文算法学习平台，有丰富的题目资源

##### #### 学习资料

1. \*\*算法导论\*\*: 深入理解算法理论基础
2. \*\*算法竞赛入门经典\*\*: 包含字符串算法详解
3. \*\*编程珠玑\*\*: 提供算法设计的思路和技巧
4. \*\*OI Wiki\*\*: 详细的算法讲解和实现
5. \*\*CP-Algorithms\*\*: 高质量的算法教程网站

#### #### 相关题目链接

- [LeetCode 5. 最长回文子串] (<https://leetcode.com/problems/longest-palindromic-substring/>)
  - [LeetCode 647. 回文子串] (<https://leetcode.com/problems/palindromic-substrings/>)
  - [LeetCode 214. 最短回文串] (<https://leetcode.com/problems/shortest-palindrome/>)
  - [LeetCode 1960. 两个回文子字符串长度的最大乘积] (<https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/>)
  - [LeetCode 2223. 构造字符串的总得分和] (<https://leetcode.com/problems/sum-of-scores-of-built-strings/>)
  - [LeetCode 3031. 将单词恢复初始状态所需的最短时间 II] (<https://leetcode.com/problems/minimum-time-to-revert-word-to-initial-state-ii/>)
  - [Codeforces 126B. Password] (<https://codeforces.com/problemset/problem/126/B>)
  - [Codeforces 1326D2 – Prefix-Suffix Palindrome] (<https://codeforces.com/problemset/problem/1326/D2>)
  - [洛谷 P3805 【模板】manacher] (<https://www.luogu.com.cn/problem/P3805>)
  - [洛谷 P5410 【模板】扩展 KMP/exKMP (Z 函数) ] (<https://www.luogu.com.cn/problem/P5410>)
  - [UVa 11475 – Extend to Palindrome] ([https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=2470](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2470))
  - [SPOJ – Pattern Find] (<https://www.spoj.com/problems/>)
  - [HackerEarth – String Similarity] (<https://www.hackerearth.com/practice/algorithms/string-algorithm/z-algorithm/tutorial/>)
  - [AtCoder ABC141E – Who Says a Pun?] ([https://atcoder.jp/contests/abc141/tasks/abc141\\_e](https://atcoder.jp/contests/abc141/tasks/abc141_e))
  - [USACO 2011 November Contest, Bronze – Cow Photographs] (<http://www.usaco.org/index.php?page=viewproblem2&cpid=95>)
  - [牛客网 NC15051 – 字符串的匹配] (<https://www.nowcoder.com/>)
  - [AcWing 141. 周期] (<https://www.acwing.com/problem/content/143/>)
  - [POJ 3240 – 回文串] (<http://poj.org/problem?id=3240>)
  - [LeetCode 336. 回文对] (<https://leetcode.com/problems/palindrome-pairs/>)
  - [LeetCode 131. 分割回文串] (<https://leetcode.com/problems/palindrome-partitioning/>)
  - [LeetCode 132. 分割回文串 II] (<https://leetcode.com/problems/palindrome-partitioning-ii/>)
- 

文件: comprehensive\_problems.md

---

# Manacher 算法与 Z 函数综合题目集

## 目录

1. [Manacher 算法题目] (#manacher 算法题目)
  - [LeetCode 5. 最长回文子串] (#leetcode-5-最长回文子串)
  - [LeetCode 647. 回文子串] (#leetcode-647-回文子串)
  - [LeetCode 214. 最短回文串] (#leetcode-214-最短回文串)
  - [LeetCode 1960. 两个回文子字符串长度的最大乘积] (#leetcode-1960-两个回文子字符串长度的最大乘

积)

- [洛谷 P3805 【模板】manacher] (#洛谷-p3805-模板 manacher)
  - [UVa 11475 - Extend to Palindrome] (#uva-11475---extend-to-palindrome)
  - [Codeforces 1326D2 - Prefix-Suffix Palindrome] (#codeforces-1326d2---prefix-suffix-palindrome)
  - [HackerRank - Palindromic Substrings] (#hackerrank---palindromic-substrings)
  - [AcWing 141. 周期] (#acwing-141-周期)
  - [POJ 3240 - 回文串] (#poj-3240---回文串)
  - [LeetCode 336. 回文对] (#leetcode-336-回文对)
  - [LeetCode 131. 分割回文串] (#leetcode-131-分割回文串)
  - [LeetCode 132. 分割回文串 II] (#leetcode-132-分割回文串-ii)
  - [SPOJ PALIN - The Next Palindrome] (#spo-j-palin---the-next-palindrome)
  - [HackerRank Build a Palindrome] (#hackerrank-build-a-palindrome)
  - [洛谷 P1659 [国家集训队]拉拉队排练] (#洛谷-p1659-国家集训队拉拉队排练)
  - [洛谷 P4555 [国家集训队]最长双回文串] (#洛谷-p4555-国家集训队最长双回文串)
2. [Z 函数（扩展 KMP）题目] (#z 函数扩展 kmp 题目)
- [LeetCode 2223. 构造字符串的总得分和] (#leetcode-2223-构造字符串的总得分和)
  - [LeetCode 3031. 将单词恢复初始状态所需的最短时间 II] (#leetcode-3031-将单词恢复初始状态所需的最短时间-ii)
  - [Codeforces 126B. Password] (#codeforces-126b-password)
  - [洛谷 P5410 【模板】扩展 KMP/exKMP (Z 函数) ] (#洛谷-p5410-模板扩展 kmppexkmpz-函数)
  - [SPOJ - Pattern Find] (#spo-j---pattern-find)
  - [HackerEarth - String Similarity] (#hackerearth---string-similarity)
  - [AtCoder ABC141E - Who Says a Pun?] (#atcoder-abc141e---who-says-a-pun)
  - [USACO 2011 November Contest, Bronze - Cow Photographs] (#usaco-2011-november-contest-bronze---cow-photographs)
  - [牛客网 NC15051 - 字符串的匹配] (#牛客网-nc15051---字符串的匹配)
3. [高级应用题目] (#高级应用题目)
- [LeetCode 336. 回文对] (#leetcode-336-回文对-1)
  - [LeetCode 131. 分割回文串] (#leetcode-131-分割回文串-1)
  - [LeetCode 132. 分割回文串 II] (#leetcode-132-分割回文串-ii-1)

## Manacher 算法题目

#### 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。

\*\*示例\*\*:

```

输入: s = "babad"

输出: "bab"

解释：“aba” 同样是符合题意的答案。

输入： s = "cbbd"

输出： "bb"

...

解题思路：

使用 Manacher 算法找到最长回文子串，在处理后的字符串中找到最长回文的中心位置，将该位置映射回原字符串，提取对应的子串。

时间复杂度：O(n)

空间复杂度：O(n)

代码实现：

- [Java 版本] (Code01_Manacher.java)

- [Python 版本] (manacher_python.py)

- [C++版本] (manacher_cpp.cpp)

LeetCode 647. 回文子串

题目描述：

给定一个字符串，计算其中回文子串的数目。

示例：

...

输入： "abc"

输出： 3

解释： 三个回文子串： "a", "b", "c"

输入： "aaa"

输出： 6

解释： 六个回文子串： "a", "a", "a", "aa", "aa", "aaa"

...

解题思路：

使用 Manacher 算法计算每个位置的回文半径，每个回文半径可以贡献一定数量的回文子串，由于我们插入了'#'，实际回文子串数量需要转换计算。

时间复杂度：O(n)

空间复杂度：O(n)

LeetCode 214. 最短回文串

题目描述:

给定一个字符串 s ，你可以通过在字符串前面添加字符将其转换为回文串。找到并返回可以用这种方式转换的最短回文串。

示例:

...

输入: $s = "aacecaaa"$

输出: $"aaacecaaa"$

输入: $s = "abcd"$

输出: $"dcbabcd"$

...

解题思路:

此题虽然可以用 Manacher 算法解决，但更常用的是 KMP 算法。我们将字符串与其反转拼接，中间用特殊字符分隔，计算 KMP 的 LPS 数组，找到原字符串的最长回文前缀，在原字符串前添加反转的部分。

时间复杂度: $O(n)$

空间复杂度: $O(n)$

LeetCode 1960. 两个回文子字符串长度的最大乘积

题目描述:

给你一个下标从 0 开始的字符串 s ，你需要找到两个不重叠的回文子字符串，它们的长度都必须为奇数，使得它们长度的乘积最大。

示例:

...

输入: $s = "ababbb"$

输出: 9

解释: 可以选择子串"aba"和"bbb"，它们的长度乘积是 $3*3=9$ 。

输入: $s = "zaaaxbbby"$

输出: 9

解释: 可以选择子串"aaa"和"bbb"，它们的长度乘积是 $3*3=9$ 。

...

解题思路:

使用 Manacher 算法计算所有奇回文信息，预处理前缀和后缀数组，分别记录到每个位置为止的最长回文长度，枚举每个分割点，通过前后缀获取左右两个子串中的最长回文大小，相乘即可。

时间复杂度: $O(n)$

空间复杂度: $O(n)$

****代码实现**:**

- [Java 版本] (LeetCode1960_MaxProduct.java)

洛谷 P3805 【模板】manacher

****题目描述**:**

给出一个只由小写英文字符 a, b, c...y, z 组成的字符串 S, 求 S 中最长回文串的长度。

****解题思路**:**

直接使用 Manacher 算法模板，返回最长回文子串的长度。

****时间复杂度**:** O(n)

****空间复杂度**:** O(n)

UVa 11475 – Extend to Palindrome

****题目描述**:**

给定一个字符串，你需要在其后面添加尽可能少的字符，使其成为一个回文串。

****示例**:**

输入: aaaaa → 输出: aaaaa

输入: abcd → 输出: abcdcba

****解题思路**:**

可以使用 Manacher 算法或者结合 KMP 来解决。我们需要找到字符串的最长前缀，使得该前缀是后缀的回文部分。然后将剩余部分反转添加到字符串后面。

****时间复杂度**:** O(n)

****空间复杂度**:** O(n)

Codeforces 1326D2 – Prefix-Suffix Palindrome

****题目描述**:**

给定一个字符串 s，构造一个长度不超过 $2n$ 的回文串，该回文串必须以 s 为前缀，且要求尽可能长。

****解题思路**:**

首先匹配前后缀中相同的部分，然后在中间部分找到最长的回文前缀或后缀，使得整个字符串构成回文。可以使用 Manacher 算法来高效找到中间部分的最长回文。

****时间复杂度**:** O(n)

****空间复杂度**:** O(n)

HackerRank – Palindromic Substrings

****题目描述**:**

计算一个字符串中所有回文子串的数量。

****解题思路**:**

使用 Manacher 算法可以高效地解决这个问题。对于每个中心位置，计算其回文半径，从而统计回文子串的数量。

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

AcWing 141. 周期

****题目描述**:**

一个字符串的前缀是周期性的，如果它可以由某个子串重复多次组成。例如，“abcabcabc”的前缀长度为 9 的部分可以由“abc”重复 3 次组成。给定一个字符串，求出它的每个前缀的最小周期长度。

****解题思路**:**

这个问题可以使用 Z 函数来解决。对于前缀长度 i ，如果 $i - z[i]$ 是一个周期，那么最小周期就是 $i - z[i]$ ，其中 $z[i]$ 是 Z 函数数组在位置 i 处的值。

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

POJ 3240 – 回文串

****题目描述**:**

给定一个字符串，求其最长回文子串的长度。

****解题思路**:**

直接使用 Manacher 算法求解最长回文子串的长度。

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

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) ，使得两个单词拼接起来是回文串。

****解题思路**:**

1. 使用 Z 函数或 Manacher 算法预处理每个单词

2. 构建字典树或哈希表
3. 枚举所有可能的拼接方式

****时间复杂度**:** $O(n * k^2)$, 其中 n 是单词数量, k 是单词平均长度

****空间复杂度**:** $O(n * k)$

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。返回所有可能的分割方案。

****解题思路**:**

1. 使用 Manacher 算法预处理回文信息
2. 使用回溯法枚举所有分割方案
3. 利用预处理信息快速判断子串是否为回文

****时间复杂度**:** $O(n * 2^n)$

****空间复杂度**:** $O(n^2)$

LeetCode 132. 分割回文串 II

****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。返回符合要求的最少分割次数。

****解题思路**:**

1. 使用 Manacher 算法预处理回文信息
2. 使用动态规划计算最少分割次数
3. 优化状态转移过程

****时间复杂度**:** $O(n^2)$

****空间复杂度**:** $O(n^2)$

SPOJ PALIN - The Next Palindrome

****题目描述**:**

给定一个整数, 找到大于该数的最小回文数。

****解题思路**:**

1. 将数字转换为字符串
2. 使用 Manacher 算法找到当前数字的回文性质
3. 构造下一个回文数

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

HackerRank Build a Palindrome

****题目描述**:**

给定两个字符串 a 和 b , 从 a 中取一个非空前缀, 从 b 中取一个非空后缀, 拼接成一个回文串, 求最长的回文串长度。

****解题思路**:**

1. 使用 Manacher 算法预处理两个字符串
2. 枚举所有可能的前缀后缀组合
3. 检查拼接后的字符串是否为回文

****时间复杂度**:** $O(n^2)$

****空间复杂度**:** $O(n)$

洛谷 P1659 [国家集训队]拉拉队排练

****题目描述**:**

求字符串中所有奇数长度回文串的长度乘积。

****解题思路**:**

1. 使用 Manacher 算法找到所有奇数长度回文串
2. 统计每个长度的回文串数量
3. 计算前 k 大的长度乘积

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

洛谷 P4555 [国家集训队]最长双回文串

****题目描述**:**

求字符串中最长的双回文子串长度 (可以分成两个回文串的字符串)。

****解题思路**:**

1. 使用 Manacher 算法预处理
2. 计算每个位置作为分割点的左右最长回文
3. 枚举所有分割点求最大值

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(n)$

Z 函数 (扩展 KMP) 题目

LeetCode 2223. 构造字符串的总得分和

题目描述:

你需要从空字符串开始构造一个长度为 n 的字符串 s ，构造过程为每次给当前字符串前面添加一个字符。构造过程中得到的所有字符串编号为 1 到 n ，其中长度为 i 的字符串编号为 s_i 。 s_i 的得分为 s_i 和 s_n 的最长公共前缀的长度（注意 $s == s_n$ ）。请你返回每一个 s_i 的得分之和。

示例:

```

输入:  $s = "babab"$

输出: 9

解释:

$s_1 == "b"$ , 得分 1

$s_2 == "ab"$ , 得分 0

$s_3 == "bab"$ , 得分 3

$s_4 == "abab"$ , 得分 0

$s_5 == "babab"$ , 得分 5

总和为  $1+0+3+0+5=9$

```

解题思路:

题目求的就是扩展 KMP (Z 数组) 的所有元素之和。使用 Z 函数计算每个后缀与原字符串的最长公共前缀长度，然后求和。

时间复杂度: $O(n)$

空间复杂度: $O(n)$

代码实现:

- [Java 版本] (Code02_ExpandKMP.java)
- [Python 版本] (z_function_python.py)
- [C++版本] (z_function_cpp.cpp)

LeetCode 3031. 将单词恢复初始状态所需的最短时间 II

题目描述:

给你一个下标从 0 开始的字符串 $word$ 和一个整数 k 。每一秒执行以下操作:

1. 移除 $word$ 的前 k 个字符
2. 在 $word$ 的末尾添加 k 个任意字符

返回将 $word$ 恢复到初始状态所需的最短时间（该时间必须大于零）。

示例:

```

输入: word = "abacaba", k = 3

输出: 2

解释:

第 1 秒后, word 变成"acaba\*\*" (用\*表示添加的字符)

第 2 秒后, word 变成"aba\*\*\*\*"

如果添加的字符分别为"cac"和"caba", word 就恢复为"abacaba"

...

\*\*解题思路\*\*:

使用 Z 函数计算每个后缀与原字符串的最长公共前缀长度, 查找满足条件的最长时间。

\*\*时间复杂度\*\*: O(n)

\*\*空间复杂度\*\*: O(n)

### Codeforces 126B. Password

\*\*题目描述\*\*:

给定一个字符串 s, 找出最长的子串 t, 它既是 s 的前缀, 也是 s 的后缀, 还在 s 的中间出现过。如果存在这样的子串, 输出最长的那个; 否则输出"Just a legend"。

\*\*解题思路\*\*:

使用 Z 函数 (扩展 KMP) 算法解决此问题:

1. 计算字符串 s 的 Z 函数数组 z, 其中 z[i] 表示以位置 i 开始的后缀与原字符串的最长公共前缀长度
2. 遍历 z 数组, 找到既是前缀又是后缀的子串 (即  $z[i] == n-i$  的情况)
3. 同时记录在中间出现过的前缀长度
4. 找到满足所有条件的最长子串

\*\*时间复杂度\*\*: O(n)

\*\*空间复杂度\*\*: O(n)

\*\*代码实现\*\*:

- [Java 版本] (Codeforces126B\_Password.java)
- [Python 版本] (codeforces\_126b\_password.py)
- [C++版本] (codeforces\_126b\_password.cpp)

### 洛谷 P5410 【模板】扩展 KMP/exKMP (Z 函数)

\*\*题目描述\*\*:

给定两个字符串 a, b, 你要求出两个数组:

- b 的 z 函数数组 z, 即 b 与 b 的每一个后缀的 LCP 长度。
- b 与 a 的每一个后缀的 LCP 长度数组 p。

\*\*解题思路\*\*:

使用 Z 函数算法解决此问题。

**\*\*时间复杂度\*\*:**  $O(n + m)$

**\*\*空间复杂度\*\*:**  $O(n + m)$

#### SPOJ - Pattern Find

**\*\*题目描述\*\*:**

给定一个字符串  $S$  和一个模式串  $T$ , 找出  $T$  在  $S$  中的所有出现位置。

**\*\*解题思路\*\*:**

使用 Z 函数来解决字符串匹配问题。将模式串  $T$  与文本串  $S$  拼接, 中间用特殊字符分隔, 然后计算拼接后字符串的 Z 函数数组。Z 函数值等于模式串长度的位置即为匹配位置。

**\*\*时间复杂度\*\*:**  $O(n + m)$ , 其中  $n$  是文本串长度,  $m$  是模式串长度。

**\*\*空间复杂度\*\*:**  $O(n + m)$

**\*\*代码实现\*\*:**

```
``` python
# Python 版本
def z_function(s):
    n = len(s)
    z = [0] * n
    z[0] = n
    l, r = 0, 0
    for i in range(1, n):
        if i <= r:
            z[i] = min(r - i + 1, z[i - 1])
        while i + z[i] < n and s[z[i]] == s[i + z[i]]:
            z[i] += 1
        if i + z[i] - 1 > r:
            l, r = i, i + z[i] - 1
    return z

def pattern_find(pattern, text):
    combined = pattern + '#' + text
    z = z_function(combined)
    pattern_len = len(pattern)
    positions = []

    for i in range(pattern_len + 1, len(combined)):
        if z[i] == pattern_len:
            positions.append(i - pattern_len - 1)
```

```

    return positions
```
HackerEarth - String Similarity
```

#### \*\*题目描述\*\*:

对于字符串 s，定义其相似度为 s 的每个后缀与 s 的最长公共前缀的长度之和。例如，对于 s="ababaa"，其后缀分别为"ababaa"、"babaa"、"abaa"、"baa"、"aa"、"a"，对应的最长公共前缀长度分别为 6、0、3、0、1、1，因此相似度为  $6+0+3+0+1+1=11$ 。

#### \*\*解题思路\*\*:

这道题直接要求计算 Z 函数数组的和，Z[i] 表示从位置 i 开始的后缀与整个字符串的最长公共前缀长度。

**\*\*时间复杂度\*\*:** O(n)

**\*\*空间复杂度\*\*:** O(n)

#### \*\*代码实现\*\*:

```

```cpp
// C++版本
#include <iostream>
#include <vector>
#include <string>
using namespace std;

vector<int> z_function(const string &s) {
    int n = s.size();
    vector<int> z(n);
    z[0] = n;
    int l = 0, r = 0;
    for (int i = 1; i < n; ++i) {
        if (i <= r) {
            z[i] = min(r - i + 1, z[i - 1]);
        }
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
            z[i]++;
        }
        if (i + z[i] - 1 > r) {
            l = i;
            r = i + z[i] - 1;
        }
    }
    return z;
}
```

```
}
```

```
long long string_similarity(const string &s) {
    vector<int> z = z_function(s);
    long long sum = 0;
    for (int val : z) {
        sum += val;
    }
    return sum;
}
```

```
int main() {
    int t;
    cin >> t;
    while (t--) {
        string s;
        cin >> s;
        cout << string_similarity(s) << endl;
    }
    return 0;
}
```

```

### AtCoder ABC141E - Who Says a Pun?

**\*\*题目描述\*\*:**

给定一个长度为 n 的字符串 s，找出两个不重叠的子串，使得它们相等且长度尽可能大。求最大可能的长度。

**\*\*解题思路\*\*:**

可以使用 Z 函数来解决这个问题。对于每个可能的分割点，计算 Z 函数并找到最长的公共前缀。通过遍历所有可能的起始位置，找到最大长度 L，使得存在两个位置 i 和 j，满足  $j - i \geq L$ ，且  $s[i..i+L-1] == s[j..j+L-1]$ 。

**\*\*时间复杂度\*\*:**  $O(n^2)$ ，但在实际应用中效率很高。

**\*\*空间复杂度\*\*:**  $O(n)$

**\*\*代码实现\*\*:**

```
``` java
// Java 版本
import java.util.Scanner;

public class Main {
    public static int[] z_function(String s) {
```

```

int n = s.length();
int[] z = new int[n];
z[0] = n;
int l = 0, r = 0;
for (int i = 1; i < n; i++) {
    if (i <= r) {
        z[i] = Math.min(r - i + 1, z[i - 1]);
    }
    while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
        z[i]++;
    }
    if (i + z[i] - 1 > r) {
        l = i;
        r = i + z[i] - 1;
    }
}
return z;
}

public static int solve(String s) {
    int n = s.length();
    int max_len = 0;

    for (int i = 0; i < n; i++) {
        String sub = s.substring(i);
        int[] z = z_function(sub);

        for (int j = 1; j < z.length; j++) {
            if (z[j] > max_len && j >= z[j]) {
                max_len = Math.max(max_len, z[j]);
            }
        }
    }

    return max_len;
}

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    String s = sc.next();
    System.out.println(solve(s));
}

```

```
}
```

```
...
```

USACO 2011 November Contest, Bronze – Cow Photographs

题目描述:

这道题与 Z 函数的应用相关，主要涉及到字符串的周期性检测。

解题思路:

可以使用 Z 函数来检测字符串的周期性。如果字符串 s 有周期 p，那么对于所有 $i > p$ ， $s[i] = s[i-p]$ 。利用 Z 函数，可以快速找到字符串的最小周期。

时间复杂度: $O(n)$

空间复杂度: $O(n)$

牛客网 NC15051 – 字符串的匹配

题目描述:

给定两个字符串 s 和 t，求出在 s 中出现的 t 的所有位置，并输出这些位置的起始索引。

解题思路:

可以使用 Z 函数来解决字符串匹配问题，类似于 SPOJ 的 Pattern Find 题目。

时间复杂度: $O(n + m)$

空间复杂度: $O(n + m)$

高级应用题目

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) ，使得两个单词拼接起来是回文串。

解题思路:

1. 使用 Z 函数或 Manacher 算法预处理每个单词
2. 构建字典树或哈希表
3. 枚举所有可能的拼接方式

时间复杂度: $O(n * k^2)$ ，其中 n 是单词数量，k 是单词平均长度

空间复杂度: $O(n * k)$

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。返回所有可能的分割方案。

解题思路:

1. 使用 Manacher 算法预处理回文信息
2. 使用回溯法枚举所有分割方案
3. 利用预处理信息快速判断子串是否为回文

时间复杂度: $O(n * 2^n)$

空间复杂度: $O(n^2)$

LeetCode 132. 分割回文串 II

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。返回符合要求的最少分割次数。

解题思路:

1. 使用 Manacher 算法预处理回文信息
2. 使用动态规划计算最少分割次数
3. 优化状态转移过程

时间复杂度: $O(n^2)$

空间复杂度: $O(n^2)$

算法对比总结

算法	主要用途	时间复杂度	空间复杂度	典型应用	适用场景
Manacher	回文串处理	$O(n)$	$O(n)$	最长回文子串、回文计数	需要高效处理回文相关问题的场景
Z 函数	字符串匹配	$O(n)$	$O(n)$	模式匹配、前缀后缀匹配	字符串匹配、周期性检测、LCP 计算

工程化考量

1. 异常处理与鲁棒性

- **空输入处理**: 检查输入字符串是否为空或 null
- **边界条件**: 处理字符串长度为 0、1 等特殊情况
- **数组越界**: 在扩展匹配时注意数组边界检查
- **内存管理**: 预分配足够大的数组空间，避免动态扩容

2. 性能优化策略

- **局部变量优化**: 减少数组访问开销
- **懒加载数组扩展**: 根据实际需求动态调整数组大小
- **边界检查前置**: 提前处理特殊情况，避免无效计算

- **算法常数优化**: 减少不必要的操作，提高执行效率

3. 多语言实现对比

Java 实现特点

- **字符串处理**: Java 中字符串是不可变的，需要频繁创建新字符串或使用 `StringBuilder`
- **数组操作**: 数组下标从 0 开始，需要注意边界检查
- **异常处理**: 提供了完善的异常机制，可用于边界条件处理

Python 实现特点

- **动态类型**: 无需提前声明变量类型，代码更简洁
- **字符串处理**: 字符串切片操作方便，但需要注意效率
- **内存管理**: 自动垃圾回收，但可能导致内存占用较高

C++实现特点

- **指针操作**: 可以直接操作内存，性能更高
- **STL 容器**: `vector` 等容器使用灵活，但需要注意内存管理
- **性能优势**: 在处理大规模数据时通常比 Java 和 Python 更快

4. 调试与问题定位

- **中间过程打印**: 在算法执行过程中打印关键变量值
- **断言验证**: 验证输入参数的有效性和算法执行结果
- **性能分析**: 使用性能分析工具找出耗时操作

学习路径建议

初级阶段（掌握基础）

1. **理解算法原理**: 深入理解 Manacher 算法和 Z 函数的核心思想
2. **熟练实现**: 能够独立实现两种算法的代码
3. **基础题目练习**: 解决 LeetCode 5、647、214 等基础题目

中级阶段（应用扩展）

1. **应用场景识别**: 能够识别适合使用这两种算法的问题场景
2. **复杂度分析**: 掌握算法的时间和空间复杂度分析方法
3. **题目变种**: 解决 Codeforces、SPOJ 等平台的进阶题目

高级阶段（工程化）

1. **优化能力**: 能够根据具体场景优化算法实现
2. **跨语言实践**: 在多种编程语言中实现算法，理解语言特性影响
3. **工程应用**: 将算法应用到实际项目中，处理大规模数据

专家阶段（创新应用）

1. **算法变种**: 理解并实现算法的各种变种和扩展

2. **组合应用**: 将 Manacher 和 Z 函数与其他算法结合解决复杂问题
3. **性能调优**: 针对特定硬件和场景进行深度优化

常见问题与解决方案

内存溢出问题

- **问题**: 处理长字符串时可能出现内存溢出
- **解决**: 使用动态数组分配，根据实际需求调整数组大小

性能瓶颈

- **问题**: 大数据量下性能下降
- **解决**: 启用性能监控，分析瓶颈；确保关闭调试模式

算法正确性

- **问题**: 某些边缘情况下结果不正确
- **解决**: 运行单元测试；检查输入参数是否有效

资源推荐

在线评测平台

1. **LeetCode**: 提供大量 Manacher 和 Z 函数相关题目
2. **Codeforces**: 包含更多挑战性的字符串算法题目
3. **AtCoder**: 日本算法竞赛平台，有质量很高的题目
4. **SPOJ**: 包含经典的字符串处理问题
5. **洛谷**: 中文算法学习平台，有丰富的题目资源

学习资料

1. **算法导论**: 深入理解算法理论基础
2. **算法竞赛入门经典**: 包含字符串算法详解
3. **编程珠玑**: 提供算法设计的思路和技巧
4. **OI Wiki**: 详细的算法讲解和实现
5. **CP-Algorithms**: 高质量的算法教程网站

实战技巧

笔试技巧

1. **模板准备**: 提前准备好 Manacher 和 Z 函数的模板代码
2. **边界处理**: 特别注意空字符串、单字符等边界情况
3. **性能优化**: 在笔试中注意时间复杂度的优化

面试技巧

1. **算法解释**: 能够清晰解释算法的原理和复杂度
2. **代码实现**: 写出清晰、可读的代码

3. **问题分析**: 分析问题的适用算法和优化空间

文件: FINAL_SUMMARY.md

Manacher 算法与 Z 函数完全掌握指南

项目完成总结

一、已完成的工作

1.1 代码实现完善

Java 版本: 已完成所有高级题目的实现，包括:

- LeetCode 336. 回文对
- LeetCode 131. 分割回文串
- LeetCode 132. 分割回文串 II
- 洛谷 P1659 [国家集训队]拉拉队排练
- 洛谷 P4555 [国家集训队]最长双回文串
- SPOJ PALIN - The Next Palindrome
- HackerRank Build a Palindrome
- AtCoder ABC141E - Who Says a Pun?

Python 版本: 已完成所有高级题目的实现，代码已通过测试

C++版本: 已完成所有高级题目的实现

1.2 文档完善

算法总结文档: 包含详细的算法原理、时间复杂度分析、工程化考量

题目分类文档: 按难度级别分类，提供解题策略

学习路径指南: 从初级到专家的完整学习路径

二、代码质量保证

2.1 编译测试结果

- **Python 代码**: 编译通过，测试运行正常
- **Java 代码**: 编译通过，测试运行正常
- **C++代码**: 代码结构完整，需要相应编译环境

2.2 代码特性

- **详细注释**: 每个函数都有完整的注释说明

- **异常处理**: 完善的参数验证和边界条件处理
- **性能优化**: 考虑了各种优化策略
- **多语言实现**: Java、Python、C++三种语言版本

三、题目覆盖范围

3.1 基础题目（必须掌握）

1. **最长回文子串** (LeetCode 5)
2. **回文子串计数** (LeetCode 647)
3. **最短回文串** (LeetCode 214)

3.2 中级题目（应用扩展）

1. **回文对** (LeetCode 336)
2. **分割回文串** (LeetCode 131/132)

3.3 高级题目（竞赛级别）

1. **拉拉队排练** (洛谷 P1659)
2. **最长双回文串** (洛谷 P4555)
3. **下一个回文数** (SPOJ PALIN)
4. **构建回文串** (HackerRank)
5. **重复子串** (AtCoder ABC141E)

四、工程化特性

4.1 代码质量

- **模块化设计**: 每个功能独立封装
- **可复用性**: 提供通用的算法模板
- **可维护性**: 清晰的代码结构和注释

4.2 性能考量

- **时间复杂度优化**: 所有实现都是最优解
- **空间复杂度控制**: 合理使用内存空间
- **边界条件处理**: 完善的异常防御

4.3 跨语言兼容

- **Java**: 面向对象，企业级应用
- **Python**: 简洁高效，数据分析
- **C++**: 高性能，系统级开发

五、学习价值

5.1 算法深度

- **Manacher 算法**: 深入理解回文串处理

- **Z 函数**: 掌握字符串匹配的高级技巧
- **动态规划**: 复杂问题的分解与解决

5.2 工程实践

- **代码规范**: 学习工业级代码标准
- **调试技巧**: 掌握问题定位方法
- **性能分析**: 理解算法效率优化

5.3 面试准备

- **题目覆盖**: 涵盖各大公司面试高频题
- **解题思路**: 提供系统的解题方法论
- **代码实现**: 展示高质量的编码能力

六、使用指南

6.1 快速开始

```
```bash
Python 版本
cd class103
python advanced_manacher_python.py
```

```
Java 版本
cd class103
javac AdvancedManacherProblems.java
java AdvancedManacherProblems
```
```

6.2 学习顺序

1. 先阅读 `algorithm_comprehensive_summary.md` 理解算法原理
2. 运行基础题目的代码示例
3. 尝试解决中级和高级题目
4. 对比不同语言的实现差异

6.3 进阶学习

1. 尝试优化现有算法的性能
2. 实现更多相关题目
3. 将算法应用到实际项目中

七、项目结构

```
```
class103/
 └── Code01_Manacher.java # 基础 Manacher 实现
```

```
└── Code02_ExpandKMP.java # Z 函数实现
└── AdvancedManacherProblems.java # 高级题目 Java 版
└── advanced_manacher_python.py # 高级题目 Python 版
└── advanced_manacher_cpp.cpp # 高级题目 C++ 版
└── comprehensive_problems.md # 题目详解
└── algorithm_comprehensive_summary.md # 算法总结
└── FINAL_SUMMARY.md # 本项目总结
``
```

## ### 八、后续建议

### #### 8.1 进一步学习

1. \*\*更多字符串算法\*\*: KMP、AC 自动机、后缀数组等
2. \*\*算法竞赛\*\*: 参加 Codeforces、LeetCode 周赛等
3. \*\*实际应用\*\*: 将算法应用到文本处理、数据分析等领域

### #### 8.2 代码优化

1. \*\*性能测试\*\*: 对不同规模数据进行性能分析
2. \*\*内存优化\*\*: 进一步优化空间复杂度
3. \*\*并行计算\*\*: 考虑多线程优化方案

### #### 8.3 扩展功能

1. \*\*可视化工具\*\*: 展示算法执行过程
2. \*\*测试用例生成\*\*: 自动生成测试数据
3. \*\*性能对比\*\*: 不同语言实现的性能对比

## ### 九、总结

本项目全面覆盖了 Manacher 算法和 Z 函数的相关题目，提供了三种编程语言的完整实现。通过本项目的学习，您将能够：

1. \*\*深入理解\*\*字符串处理算法的核心原理
2. \*\*熟练应用\*\*这些算法解决实际问题
3. \*\*掌握工程化\*\*的代码编写和优化技巧
4. \*\*具备应对\*\*算法面试和竞赛的能力

希望本项目对您的算法学习之旅有所帮助！

---

\*项目完成时间：2025 年 10 月 25 日\*

\*最后更新：2025 年 10 月 25 日\*

=====

文件: README.md

---

## # Manacher 算法与 Z 函数（扩展 KMP）

本目录包含 Manacher 算法和 Z 函数（扩展 KMP）的高效实现和详细讲解，以及相关题目的解决方案。这两种算法都是处理字符串匹配和回文问题的强大工具，时间复杂度均为  $O(n)$ 。

### ## 目录结构

- **algorithm\_summary.md**: 算法原理、核心思想、复杂度分析、适用场景等详细总结
- **comprehensive\_problems.md**: 综合题目集，包含 LeetCode、洛谷等平台的相关题目
- **Code01\_Manacher.java**: Manacher 算法的 Java 实现，包含工程化特性
- **Code02\_ExpandKMP.java**: Z 函数（扩展 KMP）的 Java 实现，包含工程化特性
- **README.md**: 本说明文件

### ## 算法简介

#### #### Manacher 算法

Manacher 算法用于高效查找字符串中的最长回文子串，时间复杂度为  $O(n)$ 。它通过利用已计算的回文信息来避免重复计算，主要特点：

- 统一处理奇数和偶数长度的回文串
- 利用回文的对称性加速计算
- 维护最右回文边界来优化扩展过程

#### #### Z 函数（扩展 KMP）

Z 函数计算字符串的每个后缀与原字符串的最长公共前缀长度，时间复杂度为  $O(n)$ 。主要应用于：

- 模式匹配问题
- 重复子串检测
- 字符串周期性分析

### ## 工程化特性

我们的实现包含以下工程化特性：

1. **参数验证与异常处理**: 全面的输入验证和异常捕获机制
2. **性能监控与日志记录**: 可配置的性能统计和详细日志
3. **调试支持**: 可开关的调试模式，输出关键中间状态
4. **单元测试**: 内置测试用例，确保算法正确性

5. \*\*边界情况处理\*\*: 完善的空字符串、单字符等特殊情况处理
6. \*\*资源管理\*\*: 安全的文件和资源关闭机制

## 使用方法

#### Manacher 算法

```
``` java
// 创建 Manacher 算法实例
Code01_Manacher manacher = new Code01_Manacher();

// 设置调试模式
manacher.setDebugMode(true);

// 查找最长回文子串
String s = "babad";
String longestPalindrome = manacher.longestPalindrome(s);

// 统计回文子串数量
int count = manacher.countSubstrings(s);

// 计算最短回文串
String shortestPalindrome = manacher.shortestPalindrome(s);

// 运行单元测试
manacher.runUnitTests();
```
```

#### Z 函数（扩展 KMP）

```
``` java
// 创建 Z 函数实例
Code02_ExpandKMP zFunction = new Code02_ExpandKMP();

// 设置性能监控
zFunction.setPerformanceMonitoring(true);

// 计算构造字符串的总得分和 (LeetCode 2223)
long score = zFunction.sumScores("babab");

// 计算恢复初始状态所需时间 (LeetCode 3031)
int time = zFunction.minimumTimeToInitialState("abacaba", 3);
```

```
// 运行单元测试  
zFunction.runUnitTests();  
```
```

## ## 命令行参数

运行程序时支持以下命令行参数：

- `--debug`：启用调试模式，输出详细日志
- `--performance`：启用性能监控，统计操作耗时
- `--test`：运行内置单元测试

## ## 性能优化

1. **局部变量优化**：减少数组访问开销
2. **懒加载数组扩展**：根据实际需求动态调整数组大小
3. **边界检查前置**：提前处理特殊情况，避免无效计算
4. **算法常数优化**：减少不必要的操作，提高执行效率

## ## 常见问题与解决方案

### ### 内存溢出

- **问题**：处理长字符串时可能出现内存溢出
- **解决**：使用`ensureArrayCapacity`方法动态调整数组大小

### ### 性能问题

- **问题**：大数据量下性能下降
- **解决**：启用性能监控，分析瓶颈；确保关闭调试模式

### ### 算法正确性

- **问题**：某些边缘情况下结果不正确
- **解决**：运行单元测试；检查输入参数是否有效

## ## 算法应用场景与相关题目

### ### Manacher 算法应用场景

1. **回文检测**：判断字符串是否为回文串
2. **最长回文子串**：找到字符串中的最长回文子串
3. **回文计数**：统计字符串中回文子串的个数
4. **回文构造**：通过添加字符构造回文串

### ### Z 函数应用场景

1. **字符串匹配**：在一个字符串中查找另一个字符串的所有出现位置

2. \*\*周期检测\*\*: 检测字符串的周期性
3. \*\*前缀后缀匹配\*\*: 查找既是前缀又是后缀的子串
4. \*\*字符串压缩\*\*: 利用周期性进行字符串压缩

### ### 相关题目链接

#### #### LeetCode 题目

- [LeetCode 5. 最长回文子串] (<https://leetcode.com/problems/longest-palindromic-substring/>)
- [LeetCode 647. 回文子串] (<https://leetcode.com/problems/palindromic-substrings/>)
- [LeetCode 214. 最短回文串] (<https://leetcode.com/problems/shortest-palindrome/>)
- [LeetCode 1960. 两个回文子字符串长度的最大乘积] (<https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/>)
- [LeetCode 2223. 构造字符串的总得分和] (<https://leetcode.com/problems/sum-of-scores-of-built-strings/>)
- [LeetCode 3031. 将单词恢复初始状态所需的最短时间 II] (<https://leetcode.com/problems/minimum-time-to-revert-word-to-initial-state-ii/>)
- [LeetCode 336. 回文对] (<https://leetcode.com/problems/palindrome-pairs/>)
- [LeetCode 131. 分割回文串] (<https://leetcode.com/problems/palindrome-partitioning/>)
- [LeetCode 132. 分割回文串 II] (<https://leetcode.com/problems/palindrome-partitioning-ii/>)

#### #### Codeforces 题目

- [Codeforces 126B. Password] (<https://codeforces.com/problemset/problem/126/B>)
- [Codeforces 1326D2 – Prefix-Suffix Palindrome] (<https://codeforces.com/problemset/problem/1326/D2>)

#### #### 洛谷题目

- [洛谷 P3805 【模板】manacher] (<https://www.luogu.com.cn/problem/P3805>)
- [洛谷 P5410 【模板】扩展 KMP/exKMP (Z 函数) ] (<https://www.luogu.com.cn/problem/P5410>)

#### #### 其他平台题目

- [UVa 11475 – Extend to Palindrome] ([https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=2470](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2470))
- [SPOJ – Pattern Find] (<https://www.spoj.com/problems/>)
- [HackerEarth – String Similarity] (<https://www.hackerearth.com/practice/algorithms/string-algorithm/z-algorithm/tutorial/>)
- [AtCoder ABC141E – Who Says a Pun?] ([https://atcoder.jp/contests/abc141/tasks/abc141\\_e](https://atcoder.jp/contests/abc141/tasks/abc141_e))
- [USACO 2011 November Contest, Bronze – Cow Photographs] (<http://www.usaco.org/index.php?page=viewproblem2&cpid=95>)
- [牛客网 NC15051 – 字符串的匹配] (<https://www.nowcoder.com/>)
- [AcWing 141. 周期] (<https://www.acwing.com/problem/content/143/>)
- [POJ 3240 – 回文串] (<http://poj.org/problem?id=3240>)

## ## 学习资源

- 详细算法原理请参考`algorithm\_summary.md`
- 实战题目练习请参考`comprehensive\_problems.md`
- 更多字符串算法请查看本项目其他相关章节

## ## 贡献指南

欢迎提交问题反馈和代码改进！如果发现 bug 或有更好的实现方式，请提交 issue 或 pull request。

## ## 许可证

本项目采用 MIT 许可证。

## [代码文件]

文件: AdvancedManacherProblems.java

```
// package class103; // 注释掉包声明，便于直接运行
```

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

/**
 * 高级 Manacher 算法题目实现
 * 包含更多复杂的回文串处理问题和 Z 函数应用
 *
 * 本文件实现了以下高级题目：
 * 1. LeetCode 336. 回文对
 * 2. LeetCode 131. 分割回文串
 * 3. LeetCode 132. 分割回文串 II
 * 4. 洛谷 P1659 [国家集训队]拉拉队排练
 * 5. 洛谷 P4555 [国家集训队]最长双回文串
 * 6. SPOJ PALIN - The Next Palindrome
 * 7. HackerRank Build a Palindrome
 * 8. AtCoder ABC141E - Who Says a Pun?
 *
 * 时间复杂度分析: O(n) 到 O(n2) 不等，取决于具体问题
 * 空间复杂度分析: O(n) 到 O(n2) 不等，取决于具体问题
 */
```

```
public class AdvancedManacherProblems {

 // 日志记录器 - 简化实现，避免复杂的日志依赖
 // private static final Logger logger =
 Logger.getLogger(AdvancedManacherProblems.class.getName());

 /**
 * LeetCode 336. 回文对
 * 给定一组互不相同的单词，找出所有不同的索引对(i, j)，使得两个单词拼接起来是回文串。
 *
 * 解题思路：
 * 1. 使用字典树存储所有单词及其反转
 * 2. 对于每个单词，检查其前缀和后缀是否为回文
 * 3. 在字典树中查找剩余部分的匹配
 *
 * 时间复杂度：O(n * k2)，其中 n 是单词数量，k 是单词平均长度
 * 空间复杂度：O(n * k)
 *
 * @param words 单词列表
 * @return 回文对列表
 */
 public static List<List<Integer>> palindromePairs(String[] words) {
 List<List<Integer>> result = new ArrayList<>();
 if (words == null || words.length < 2) return result;

 // 构建字典树
 TrieNode root = new TrieNode();
 for (int i = 0; i < words.length; i++) {
 insertWord(root, words[i], i);
 }

 // 查找回文对
 for (int i = 0; i < words.length; i++) {
 String word = words[i];
 TrieNode node = root;

 // 检查整个单词是否在字典树中
 for (int j = 0; j < word.length(); j++) {
 char c = word.charAt(j);
 if (node.children[c - 'a'] == null) {
 break;
 }
 }
 }
 }
}
```

```

node = node.children[c - 'a'];

 // 如果当前节点是某个单词的结尾，且剩余部分是回文
 if (node.wordIndex != -1 && node.wordIndex != i &&
 isPalindrome(word, j + 1, word.length() - 1)) {
 result.add(Arrays.asList(i, node.wordIndex));
 }
 }

 // 检查单词的反转
 String reversed = new StringBuilder(word).reverse().toString();
 TrieNode revNode = root;
 for (int j = 0; j < reversed.length(); j++) {
 char c = reversed.charAt(j);
 if (revNode.children[c - 'a'] == null) {
 break;
 }
 revNode = revNode.children[c - 'a'];
 }

 // 如果当前节点是某个单词的结尾，且剩余部分是回文
 if (revNode.wordIndex != -1 && revNode.wordIndex != i &&
 isPalindrome(reversed, j + 1, reversed.length() - 1)) {
 result.add(Arrays.asList(revNode.wordIndex, i));
 }
}

return result;
}

/**
 * 字典树节点
 */
static class TrieNode {
 TrieNode[] children;
 int wordIndex; // 存储单词索引，-1 表示不是单词结尾

 public TrieNode() {
 children = new TrieNode[26];
 wordIndex = -1;
 }
}

```

```
/**
 * 向字典树中插入单词
 */

private static void insertWord(TrieNode root, String word, int index) {
 TrieNode node = root;
 for (char c : word.toCharArray()) {
 if (node.children[c - 'a'] == null) {
 node.children[c - 'a'] = new TrieNode();
 }
 node = node.children[c - 'a'];
 }
 node.wordIndex = index;
}
```

```
/**
 * 判断子串是否为回文
 */

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;
}
```

```
/**
 * LeetCode 131. 分割回文串
 * 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。返回所有可能的分割方案。
 *
 * 解题思路：
 * 1. 使用 Manacher 算法预处理回文信息
 * 2. 使用回溯法枚举所有分割方案
 * 3. 利用预处理信息快速判断子串是否为回文
 *
 * 时间复杂度：O(n * 2^n)
 * 空间复杂度：O(n^2)
 *
 * @param s 输入字符串
 * @return 所有分割方案
 */
```

```

public static List<List<String>> partition(String s) {
 List<List<String>> result = new ArrayList<>();
 if (s == null || s.length() == 0) return result;

 // 预处理回文信息
 boolean[][] isPalindrome = preprocessPalindrome(s);

 // 回溯法枚举所有分割方案
 backtrack(s, 0, new ArrayList<>(), result, isPalindrome);

 return result;
}

/**
 * 预处理回文信息
 */
private static boolean[][] preprocessPalindrome(String s) {
 int n = s.length();
 boolean[][] dp = new boolean[n][n];

 // 单个字符都是回文
 for (int i = 0; i < n; i++) {
 dp[i][i] = true;
 }

 // 两个字符的情况
 for (int i = 0; i < n - 1; i++) {
 dp[i][i + 1] = (s.charAt(i) == s.charAt(i + 1));
 }

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

 return dp;
}

/**
 * 回溯法实现

```

```

*/
private static void backtrack(String s, int start, List<String> current,
 List<List<String>> result, boolean[][] isPalindrome) {
 if (start == s.length()) {
 result.add(new ArrayList<>(current));
 return;
 }

 for (int end = start; end < s.length(); end++) {
 if (isPalindrome[start][end]) {
 current.add(s.substring(start, end + 1));
 backtrack(s, end + 1, current, result, isPalindrome);
 current.remove(current.size() - 1);
 }
 }
}

/**
 * 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。
 *
 * 解题思路：
 * 1. 使用 Manacher 算法预处理回文信息
 * 2. 使用动态规划计算最少分割次数
 * 3. 优化状态转移过程
 *
 * 时间复杂度：O(n2)
 * 空间复杂度：O(n2)
 *
 * @param s 输入字符串
 * @return 最少分割次数
 */
public static int minCut(String s) {
 int n = s.length();
 if (n <= 1) return 0;

 // 预处理回文信息
 boolean[][] isPalindrome = preprocessPalindrome(s);

 // 动态规划计算最少分割次数
 int[] dp = new int[n];
 Arrays.fill(dp, Integer.MAX_VALUE);

```

```

for (int i = 0; i < n; i++) {
 if (isPalindrome[0][i]) {
 dp[i] = 0; // 整个子串是回文，不需要分割
 } else {
 for (int j = 0; j < i; j++) {
 if (isPalindrome[j + 1][i]) {
 dp[i] = Math.min(dp[i], dp[j] + 1);
 }
 }
 }
}

return dp[n - 1];
}

/***
 * 洛谷 P1659 [国家集训队]拉拉队排练
 * 求字符串中所有奇数长度回文串的长度乘积。
 *
 * 解题思路：
 * 1. 使用 Manacher 算法找到所有奇数长度回文串
 * 2. 统计每个长度的回文串数量
 * 3. 计算前 k 大的长度乘积
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @param k 前 k 大的长度
 * @return 长度乘积（取模）
 */
public static long longestPalindromeProduct(String s, int k) {
 int n = s.length();
 if (n == 0) return 0;

 // 使用 Manacher 算法计算回文半径
 int[] radius = manacherOdd(s);

 // 统计每个长度的回文串数量
 int[] count = new int[n + 1];
 for (int i = 0; i < n; i++) {
 int len = 2 * radius[i] + 1;
 count[len]++;
 }
}

```

```
}

// 计算前 k 大的长度乘积
long product = 1;
long mod = 1000000007;
int remaining = k;

for (int len = n; len >= 1 && remaining > 0; len -= 2) {
 if (count[len] > 0) {
 int take = Math.min(count[len], remaining);
 product = (product * pow(len, take, mod)) % mod;
 remaining -= take;
 }
}

return product;
}
```

```
/***
 * 快速幂计算
 */
private static long pow(long base, int exponent, long mod) {
 long result = 1;
 while (exponent > 0) {
 if ((exponent & 1) == 1) {
 result = (result * base) % mod;
 }
 base = (base * base) % mod;
 exponent >>= 1;
 }
 return result;
}
```

```
/***
 * Manacher 算法计算奇回文串
 */
private static int[] manacherOdd(String s) {
 int n = s.length();
 int[] radius = new int[n];

 for (int i = 0, l = 0, r = -1; i < n; i++) {
 int k = (i > r) ? 1 : Math.min(radius[l + r - i], r - i + 1);
```

```

 while (0 <= i - k && i + k < n && s.charAt(i - k) == s.charAt(i + k)) {
 k++;
 }

 radius[i] = k - 1;

 if (i + k - 1 > r) {
 l = i - k + 1;
 r = i + k - 1;
 }
 }

 return radius;
}

/***
 * 洛谷 P4555 [国家集训队]最长双回文串
 * 输入字符串 s, 求 s 的最长双回文子串 t 的长度 (双回文子串就是可以分成两个回文串的字符串)
 *
 * 解题思路:
 * 1. 使用 Manacher 算法预处理
 * 2. 计算每个位置作为分割点的左右最长回文
 * 3. 枚举所有分割点求最大值
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * @param s 输入字符串
 * @return 最长双回文串长度
 */
public static int longestDoublePalindrome(String s) {
 int n = s.length();
 if (n < 2) return 0;

 // 预处理字符串
 String processed = preprocess(s);
 char[] chars = processed.toCharArray();
 int len = chars.length;
 int[] p = new int[len];

 int center = 0, right = 0;
 for (int i = 1; i < len - 1; i++) {
 int mirror = 2 * center - i;

```

```

if (i < right) {
 p[i] = Math.min(right - i, p[mirror]);
}

while (i + p[i] + 1 < len && i - p[i] - 1 >= 0 &&
 chars[i + p[i] + 1] == chars[i - p[i] - 1]) {
 p[i]++;
}

if (i + p[i] > right) {
 center = i;
 right = i + p[i];
}
}

// 计算每个位置的最长回文长度
int[] leftMax = new int[n];
int[] rightMax = new int[n];

// 从左到右计算每个位置的最长回文前缀
int maxLen = 0;
for (int i = 0; i < n; i++) {
 int pos = 2 * i + 1;
 maxLen = Math.max(maxLen, p[pos]);
 leftMax[i] = maxLen;
}

// 从右到左计算每个位置的最长回文后缀
maxLen = 0;
for (int i = n - 1; i >= 0; i--) {
 int pos = 2 * i + 1;
 maxLen = Math.max(maxLen, p[pos]);
 rightMax[i] = maxLen;
}

// 枚举分割点，计算最长双回文串
int result = 0;
for (int i = 0; i < n - 1; i++) {
 result = Math.max(result, leftMax[i] + rightMax[i + 1]);
}

return result;

```

```
}

/***
 * 预处理函数，用于在字符间插入'#'
 */
private static String preprocess(String s) {
 int n = s.length();
 if (n == 0) return "^$";
 StringBuilder sb = new StringBuilder(2 * n + 3);
 sb.append("^");
 for (int i = 0; i < n; i++) {
 sb.append("#").append(s.charAt(i));
 }
 sb.append("#$");
 return sb.toString();
}

/***
 * SPOJ PALIN - The Next Palindrome
 * 给定一个整数，找到大于该数的最小回文数。
 *
 * 解题思路：
 * 1. 将数字转换为字符串
 * 2. 构造下一个回文数
 * 3. 处理进位等特殊情况
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param numStr 数字字符串
 * @return 下一个回文数
 */
public static String nextPalindrome(String numStr) {
 int n = numStr.length();
 char[] num = numStr.toCharArray();
 // 处理全 9 的情况
 if (isAllNines(num)) {
 StringBuilder result = new StringBuilder("1");
 for (int i = 0; i < n - 1; i++) {
```

```
 result.append("0");
 }
 result.append("1");
 return result.toString();
}

// 构造下一个回文数
int mid = n / 2;
boolean leftSmaller = false;
int i = mid - 1;
int j = (n % 2 == 0) ? mid : mid + 1;

// 跳过已经相同的部分
while (i >= 0 && num[i] == num[j]) {
 i--;
 j++;
}

// 检查是否需要增加中间部分
if (i < 0 || num[i] < num[j]) {
 leftSmaller = true;
}

// 复制左半部分到右半部分
while (i >= 0) {
 num[j] = num[i];
 i--;
 j++;
}

// 如果需要增加中间部分
if (leftSmaller) {
 int carry = 1;
 i = mid - 1;

 if (n % 2 == 1) {
 int midNum = num[mid] - '0' + carry;
 carry = midNum / 10;
 num[mid] = (char)(midNum % 10 + '0');
 j = mid + 1;
 } else {
 j = mid;
 }
}
```

```

// 处理进位
while (i >= 0) {
 int digit = num[i] - '0' + carry;
 carry = digit / 10;
 num[i] = (char)(digit % 10 + '0');
 num[j] = num[i];
 i--;
 j++;
}
}

return new String(num);
}

/***
 * 检查数字是否全为 9
 */
private static boolean isAllNines(char[] num) {
 for (char c : num) {
 if (c != '9') {
 return false;
 }
 }
 return true;
}

/***
 * HackerRank Build a Palindrome
 * 给定两个字符串 a 和 b，从 a 中取一个非空前缀，从 b 中取一个非空后缀，拼接成一个回文串，求最长的回文串长度。
*
* 解题思路：
* 1. 使用 Manacher 算法预处理两个字符串
* 2. 枚举所有可能的前缀后缀组合
* 3. 检查拼接后的字符串是否为回文
*
* 时间复杂度：O(n2)
* 空间复杂度：O(n)
*
* @param a 字符串 a
* @param b 字符串 b
* @return 最长回文串长度
*/

```

```

*/
public static int buildPalindrome(String a, String b) {
 int maxLen = 0;
 int n = a.length(), m = b.length();

 // 预处理两个字符串的回文信息
 boolean[][] isPalindromeA = preprocessPalindrome(a);
 boolean[][] isPalindromeB = preprocessPalindrome(b);

 // 枚举所有可能的前缀后缀组合
 for (int i = 0; i < n; i++) {
 for (int j = 0; j < m; j++) {
 // 从 a 取前缀，从 b 取后缀
 String prefix = a.substring(0, i + 1);
 String suffix = b.substring(m - j - 1);
 String combined = prefix + suffix;

 // 检查是否为回文
 if (isPalindrome(combined, 0, combined.length() - 1)) {
 maxLen = Math.max(maxLen, combined.length());
 }
 }
 }
}

```

```
return maxLen;
```

```
}
```

```
/**
```

```
* AtCoder ABC141E - Who Says a Pun?
```

```
* 给定一个长度为 n 的字符串 s，找出两个不重叠的子串，使得它们相等且长度尽可能大。
```

```
*
```

```
* 解题思路：
```

```
* 1. 使用 Z 函数计算每个后缀的匹配情况
```

```
* 2. 遍历所有可能的分割点
```

```
* 3. 找到满足条件的最长子串
```

```
*
```

```
* 时间复杂度：O(n2)
```

```
* 空间复杂度：O(n)
```

```
*
```

```
* @param s 输入字符串
```

```
* @return 最大长度
```

```
*/
```

```
public static int whoSaysPun(String s) {
```

```

int n = s.length();
int maxLen = 0;

// 对于每个可能的起始位置
for (int i = 0; i < n; i++) {
 // 计算从位置 i 开始的 Z 函数
 int[] z = zFunction(s.substring(i));

 // 查找不重叠的相同子串
 for (int j = 1; j < z.length; j++) {
 if (z[j] > maxLen && j >= z[j]) {
 maxLen = Math.max(maxLen, z[j]);
 }
 }
}

return maxLen;
}

/**
 * Z 函数计算
 */
private static int[] zFunction(String s) {
 int n = s.length();
 int[] z = new int[n];
 z[0] = n;

 for (int i = 1, l = 0, r = 0; i < n; i++) {
 if (i <= r) {
 z[i] = Math.min(r - i + 1, z[i - 1]);
 }

 while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
 z[i]++;
 }

 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
 }

 return z;
}

```

```
}

/**
 * 单元测试方法
 */
public static void runUnitTests() {
 System.out.println("===== 高级 Manacher 算法题目测试 =====");

 // 测试回文对
 System.out.println("\n1. 回文对测试:");
 String[] words1 = {"abcd", "dcba", "lls", "s", "sssll"};
 List<List<Integer>> result1 = palindromePairs(words1);
 System.out.println("回文对结果: " + result1);

 // 测试分割回文串
 System.out.println("\n2. 分割回文串测试:");
 String s2 = "aab";
 List<List<String>> result2 = partition(s2);
 System.out.println("分割方案数量: " + result2.size());

 // 测试最少分割次数
 System.out.println("\n3. 最少分割次数测试:");
 String s3 = "aab";
 int result3 = minCut(s3);
 System.out.println("最少分割次数: " + result3);

 // 测试最长双回文串
 System.out.println("\n4. 最长双回文串测试:");
 String s4 = "baacaabbacabb";
 int result4 = longestDoublePalindrome(s4);
 System.out.println("最长双回文串长度: " + result4);

 // 测试下一个回文数
 System.out.println("\n5. 下一个回文数测试:");
 String num = "12345";
 String result5 = nextPalindrome(num);
 System.out.println("下一个回文数: " + result5);

 System.out.println("\n===== 测试完成 =====");
}

/**
 * 主方法

```

```
 */
public static void main(String[] args) {
 runUnitTests();
}
=====
```

文件: advanced\_manacher\_cpp.cpp

```
=====
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <unordered_map>
#include <climits>
#include <cmath>
#include <memory>
#include <functional>

using namespace std;
```

```
/**
 * 高级 Manacher 算法题目实现 - C++版本
 * 包含更多复杂的回文串处理问题和 Z 函数应用
 *
 * 本文件实现了以下高级题目:
 * 1. LeetCode 336. 回文对
 * 2. LeetCode 131. 分割回文串
 * 3. LeetCode 132. 分割回文串 II
 * 4. 洛谷 P1659 [国家集训队]拉拉队排练
 * 5. 洛谷 P4555 [国家集训队]最长双回文串
 * 6. SPOJ PALIN - The Next Palindrome
 * 7. HackerRank Build a Palindrome
 * 8. AtCoder ABC141E - Who Says a Pun?
 *
 * 时间复杂度分析: O(n) 到 O(n2) 不等, 取决于具体问题
 * 空间复杂度分析: O(n) 到 O(n2) 不等, 取决于具体问题
 */
```

```
class AdvancedManacherCPP {
private:
 // 字典树节点
```

```

struct TrieNode {
 unordered_map<char, shared_ptr<TrieNode>> children;
 int wordIndex;

 TrieNode() : wordIndex(-1) {}
};

shared_ptr<TrieNode> root;

public:
 AdvancedManacherCPP() {
 root = make_shared<TrieNode>();
 }

 /**
 * LeetCode 336. 回文对
 * 给定一组互不相同的单词，找出所有不同的索引对(i, j)，使得两个单词拼接起来是回文串。
 *
 * 解题思路：
 * 1. 使用字典树存储所有单词及其反转
 * 2. 对于每个单词，检查其前缀和后缀是否为回文
 * 3. 在字典树中查找剩余部分的匹配
 *
 * 时间复杂度：O(n * k2)，其中 n 是单词数量，k 是单词平均长度
 * 空间复杂度：O(n * k)
 *
 * @param words 单词列表
 * @return 回文对列表
 */
 vector<vector<int>> palindromePairs(vector<string>& words) {
 vector<vector<int>> result;
 if (words.size() < 2) return result;

 // 构建字典树
 for (int i = 0; i < words.size(); i++) {
 insertWord(words[i], i);
 }

 // 查找回文对
 for (int i = 0; i < words.size(); i++) {
 string word = words[i];
 shared_ptr<TrieNode> node = root;
 }
 }
}

```

```

// 检查整个单词是否在字典树中
for (int j = 0; j < word.length(); j++) {
 char c = word[j];
 if (node->children.find(c) == node->children.end()) {
 break;
 }
 node = node->children[c];

 // 如果当前节点是某个单词的结尾，且剩余部分是回文
 if (node->wordIndex != -1 && node->wordIndex != i &&
 isPalindrome(word, j + 1, word.length() - 1)) {
 result.push_back({i, node->wordIndex});
 }
}

// 检查单词的反转
string reversedWord = word;
reverse(reversedWord.begin(), reversedWord.end());
shared_ptr<TrieNode> revNode = root;
for (int j = 0; j < reversedWord.length(); j++) {
 char c = reversedWord[j];
 if (revNode->children.find(c) == revNode->children.end()) {
 break;
 }
 revNode = revNode->children[c];

 // 如果当前节点是某个单词的结尾，且剩余部分是回文
 if (revNode->wordIndex != -1 && revNode->wordIndex != i &&
 isPalindrome(reversedWord, j + 1, reversedWord.length() - 1)) {
 result.push_back({revNode->wordIndex, i});
 }
}

return result;
}

private:
/***
 * 向字典树中插入单词
 */
void insertWord(const string& word, int index) {
 shared_ptr<TrieNode> node = root;

```

```

for (char c : word) {
 if (node->children.find(c) == node->children.end()) {
 node->children[c] = make_shared<TrieNode>();
 }
 node = node->children[c];
}
node->wordIndex = index;
}

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

public:
/***
 * LeetCode 131. 分割回文串
 * 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。返回所有可能的分割方案。
 *
 * 解题思路：
 * 1. 使用 Manacher 算法预处理回文信息
 * 2. 使用回溯法枚举所有分割方案
 * 3. 利用预处理信息快速判断子串是否为回文
 *
 * 时间复杂度：O(n * 2^n)
 * 空间复杂度：O(n^2)
 *
 * @param s 输入字符串
 * @return 所有分割方案
 */
vector<vector<string>> partition(string s) {
 vector<vector<string>> result;
 if (s.empty()) return result;

```

```

// 预处理回文信息
vector<vector<bool>> isPalindrome = preprocessPalindrome(s);

// 回溯法枚举所有分割方案
vector<string> current;
backtrack(s, 0, current, result, isPalindrome);

return result;
}

private:
/***
 * 预处理回文信息
 */
vector<vector<bool>> preprocessPalindrome(const string& s) {
 int n = s.length();
 vector<vector<bool>> dp(n, vector<bool>(n, false));

 // 单个字符都是回文
 for (int i = 0; i < n; i++) {
 dp[i][i] = true;
 }

 // 两个字符的情况
 for (int i = 0; i < n - 1; i++) {
 dp[i][i + 1] = (s[i] == s[i + 1]);
 }

 // 长度大于 2 的情况
 for (int len = 3; len <= n; len++) {
 for (int i = 0; i <= n - len; i++) {
 int j = i + len - 1;
 dp[i][j] = (s[i] == s[j] && dp[i + 1][j - 1]);
 }
 }

 return dp;
}

/***
 * 回溯法实现
 */
void backtrack(const string& s, int start, vector<string>& current,

```

```

 vector<vector<string>>& result, const vector<vector<bool>>& isPalindrome) {
 if (start == s.length()) {
 result.push_back(current);
 return;
 }

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

public:
 /**
 * 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。
 *
 * 解题思路：
 * 1. 使用 Manacher 算法预处理回文信息
 * 2. 使用动态规划计算最少分割次数
 * 3. 优化状态转移过程
 *
 * 时间复杂度：O(n2)
 * 空间复杂度：O(n2)
 *
 * @param s 输入字符串
 * @return 最少分割次数
 */
 int minCut(string s) {
 int n = s.length();
 if (n <= 1) return 0;

 // 预处理回文信息
 vector<vector<bool>> isPalindrome = preprocessPalindrome(s);

 // 动态规划计算最少分割次数
 vector<int> dp(n, INT_MAX);

 for (int i = 0; i < n; i++) {
 if (isPalindrome[0][i]) {

```

```

 dp[i] = 0; // 整个子串是回文，不需要分割
 } else {
 for (int j = 0; j < i; j++) {
 if (isPalindrome[j + 1][i]) {
 dp[i] = min(dp[i], dp[j] + 1);
 }
 }
 }

 return dp[n - 1];
}

```

```

/**
 * 洛谷 P1659 [国家集训队]拉拉队排练
 * 求字符串中所有奇数长度回文串的长度乘积。
 *
 * 解题思路：
 * 1. 使用 Manacher 算法找到所有奇数长度回文串
 * 2. 统计每个长度的回文串数量
 * 3. 计算前 k 大的长度乘积
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @param k 前 k 大的长度
 * @return 长度乘积（取模）
 */

```

```

long long longestPalindromeProduct(const string& s, int k) {
 int n = s.length();
 if (n == 0) return 0;

 // 使用 Manacher 算法计算回文半径
 vector<int> radius = manacherOdd(s);

 // 统计每个长度的回文串数量
 vector<int> count(n + 1, 0);
 for (int i = 0; i < n; i++) {
 int len = 2 * radius[i] + 1;
 count[len]++;
 }
}
```

```

// 计算前 k 大的长度乘积
long long product = 1;
long long mod = 1000000007;
int remaining = k;

for (int len = n; len >= 1 && remaining > 0; len -= 2) {
 if (count[len] > 0) {
 int take = min(count[len], remaining);
 product = (product * fastPower(len, take, mod)) % mod;
 remaining -= take;
 }
}

return product;
}

private:
/***
 * Manacher 算法计算奇回文串
 */
vector<int> manacherOdd(const string& s) {
 int n = s.length();
 vector<int> radius(n, 0);
 int l = 0, r = -1;

 for (int i = 0; i < n; i++) {
 int k = (i > r) ? 1 : min(radius[l + r - i], r - i + 1);

 while (i - k >= 0 && i + k < n && s[i - k] == s[i + k]) {
 k++;
 }

 radius[i] = k - 1;

 if (i + k - 1 > r) {
 l = i - k + 1;
 r = i + k - 1;
 }
 }

 return radius;
}

```

```

/**
 * 快速幂计算
 */
long long fastPower(long long base, int exponent, long long mod) {
 long long result = 1;
 while (exponent > 0) {
 if (exponent & 1) {
 result = (result * base) % mod;
 }
 base = (base * base) % mod;
 exponent >>= 1;
 }
 return result;
}

public:
 /**
 * 洛谷 P4555 [国家集训队]最长双回文串
 * 输入字符串 s，求 s 的最长双回文子串 t 的长度（双回文子串就是可以分成两个回文串的字符串）
 *
 * 解题思路：
 * 1. 使用 Manacher 算法预处理
 * 2. 计算每个位置作为分割点的左右最长回文
 * 3. 枚举所有分割点求最大值
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @return 最长双回文串长度
 */
int longestDoublePalindrome(const string& s) {
 int n = s.length();
 if (n < 2) return 0;

 // 预处理字符串
 string processed = preprocessString(s);
 int len = processed.length();
 vector<int> p(len, 0);

 int center = 0, right = 0;
 for (int i = 1; i < len - 1; i++) {
 int mirror = 2 * center - i;

```

```

if (i < right) {
 p[i] = min(right - i, p[mirror]);
}

while (i + p[i] + 1 < len && i - p[i] - 1 >= 0 &&
 processed[i + p[i] + 1] == processed[i - p[i] - 1]) {
 p[i]++;
}

if (i + p[i] > right) {
 center = i;
 right = i + p[i];
}
}

// 计算每个位置的最长回文长度
vector<int> leftMax(n, 0);
vector<int> rightMax(n, 0);

// 从左到右计算每个位置的最长回文前缀
int maxLen = 0;
for (int i = 0; i < n; i++) {
 int pos = 2 * i + 1;
 maxLen = max(maxLen, p[pos]);
 leftMax[i] = maxLen;
}

// 从右到左计算每个位置的最长回文后缀
maxLen = 0;
for (int i = n - 1; i >= 0; i--) {
 int pos = 2 * i + 1;
 maxLen = max(maxLen, p[pos]);
 rightMax[i] = maxLen;
}

// 枚举分割点，计算最长双回文串
int result = 0;
for (int i = 0; i < n - 1; i++) {
 result = max(result, leftMax[i] + rightMax[i + 1]);
}

return result;

```

```

}

private:
/***
 * 预处理字符串，插入特殊字符
 */
string preprocessString(const string& s) {
 if (s.empty()) return "^$";

 string result = "^";
 for (char c : s) {
 result += "#";
 result += c;
 }
 result += "#$";

 return result;
}

public:
/***
 * SPOJ PALIN - The Next Palindrome
 * 给定一个整数，找到大于该数的最小回文数。
 *
 * 解题思路：
 * 1. 将数字转换为字符串
 * 2. 构造下一个回文数
 * 3. 处理进位等特殊情况
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param numStr 数字字符串
 * @return 下一个回文数
 */
string nextPalindrome(const string& numStr) {
 if (numStr.empty()) return "1";

 string num = numStr;
 int n = num.length();

 // 处理全 9 的情况
 if (all_of(num.begin(), num.end(), [](char c) { return c == '9'; })) {

```

```
 return "1" + string(n - 1, '0') + "1";
 }

// 构造下一个回文数
int mid = n / 2;
bool leftSmaller = false;
int i = mid - 1;
int j = (n % 2 == 0) ? mid : mid + 1;

// 跳过已经相同的部分
while (i >= 0 && num[i] == num[j]) {
 i--;
 j++;
}

// 检查是否需要增加中间部分
if (i < 0 || num[i] < num[j]) {
 leftSmaller = true;
}

// 复制左半部分到右半部分
while (i >= 0) {
 num[j] = num[i];
 i--;
 j++;
}

// 如果需要增加中间部分
if (leftSmaller) {
 int carry = 1;
 i = mid - 1;

 if (n % 2 == 1) {
 int midNum = (num[mid] - '0') + carry;
 carry = midNum / 10;
 num[mid] = (midNum % 10) + '0';
 j = mid + 1;
 } else {
 j = mid;
 }

 // 处理进位
 while (i >= 0) {
```

```

 int digit = (num[i] - '0') + carry;
 carry = digit / 10;
 num[i] = (digit % 10) + '0';
 num[j] = num[i];
 i--;
 j++;
 }
}

return num;
}

/***
 * HackerRank Build a Palindrome
 * 给定两个字符串 a 和 b，从 a 中取一个非空前缀，从 b 中取一个非空后缀，拼接成一个回文串，求最长的回文串长度。
 *
 * 解题思路：
 * 1. 使用 Manacher 算法预处理两个字符串
 * 2. 枚举所有可能的前缀后缀组合
 * 3. 检查拼接后的字符串是否为回文
 *
 * 时间复杂度：O(n2)
 * 空间复杂度：O(n)
 *
 * @param a 字符串 a
 * @param b 字符串 b
 * @return 最长回文串长度
 */
int buildPalindrome(const string& a, const string& b) {
 int maxLen = 0;
 int n = a.length(), m = b.length();

 // 枚举所有可能的前缀后缀组合
 for (int i = 0; i < n; i++) {
 for (int j = 0; j < m; j++) {
 // 从 a 取前缀，从 b 取后缀
 string prefix = a.substr(0, i + 1);
 string suffix = b.substr(m - j - 1);
 string combined = prefix + suffix;

 // 检查是否为回文
 if (isPalindrome(combined, 0, combined.length() - 1)) {

```

```

 maxLen = max(maxLen, (int)combined.length());
 }
}

return maxLen;
}

/***
 * AtCoder ABC141E - Who Says a Pun?
 * 给定一个长度为 n 的字符串 s，找出两个不重叠的子串，使得它们相等且长度尽可能大。
 *
 * 解题思路：
 * 1. 使用 Z 函数计算每个后缀的匹配情况
 * 2. 遍历所有可能的分割点
 * 3. 找到满足条件的最长子串
 *
 * 时间复杂度：O(n2)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @return 最大长度
 */
int whoSaysPun(const string& s) {
 int n = s.length();
 int maxLen = 0;

 // 对于每个可能的起始位置
 for (int i = 0; i < n; i++) {
 // 计算从位置 i 开始的 Z 函数
 string substr = s.substr(i);
 vector<int> z = zFunction(substr);

 // 查找不重叠的相同子串
 for (int j = 1; j < z.size(); j++) {
 if (z[j] > maxLen && j >= z[j]) {
 maxLen = max(maxLen, z[j]);
 }
 }
 }

 return maxLen;
}

```

```

private:
 /**
 * Z 函数计算
 */
 vector<int> zFunction(const string& s) {
 int n = s.length();
 vector<int> z(n, 0);
 z[0] = n;

 int l = 0, r = 0;
 for (int i = 1; i < n; i++) {
 if (i <= r) {
 z[i] = min(r - i + 1, z[i - 1]);
 }

 while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
 z[i]++;
 }

 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
 }

 return z;
 }

public:
 /**
 * 运行单元测试
 */
 void runUnitTests() {
 cout << "===== C++版本高级 Manacher 算法题目测试 =====" << endl;

 // 测试回文对
 cout << "\n1. 回文对测试:" << endl;
 vector<string> words = {"abcd", "dcba", "lls", "s", "sssll"};
 vector<vector<int>> result1 = palindromePairs(words);
 cout << "回文对结果: ";
 for (auto& pair : result1) {
 cout << "[" << pair[0] << "," << pair[1] << "] ";
 }
 }

```

```

 }

 cout << endl;

 // 测试分割回文串
 cout << "\n2. 分割回文串测试:" << endl;
 string s2 = "aab";
 vector<vector<string>> result2 = partition(s2);
 cout << "分割方案数量: " << result2.size() << endl;

 // 测试最少分割次数
 cout << "\n3. 最少分割次数测试:" << endl;
 string s3 = "aab";
 int result3 = minCut(s3);
 cout << "最少分割次数: " << result3 << endl;

 // 测试最长双回文串
 cout << "\n4. 最长双回文串测试:" << endl;
 string s4 = "baacaabbacabb";
 int result4 = longestDoublePalindrome(s4);
 cout << "最长双回文串长度: " << result4 << endl;

 // 测试下一个回文数
 cout << "\n5. 下一个回文数测试:" << endl;
 string num = "12345";
 string result5 = nextPalindrome(num);
 cout << "下一个回文数: " << result5 << endl;

 cout << "\n===== 测试完成 =====" << endl;
}

};

int main() {
 AdvancedManacherCPP solver;
 solver.runUnitTests();
 return 0;
}

```

=====

文件: advanced\_manacher\_python.py

=====

```

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

```

"""

## 高级 Manacher 算法题目实现 – Python 版本 包含更多复杂的回文串处理问题和 Z 函数应用

本文件实现了以下高级题目：

1. LeetCode 336. 回文对
2. LeetCode 131. 分割回文串
3. LeetCode 132. 分割回文串 II
4. 洛谷 P1659 [国家集训队]拉拉队排练
5. 洛谷 P4555 [国家集训队]最长双回文串
6. SPOJ PALIN – The Next Palindrome
7. HackerRank Build a Palindrome
8. AtCoder ABC141E – Who Says a Pun?

时间复杂度分析： $O(n)$  到  $O(n^2)$  不等，取决于具体问题

空间复杂度分析： $O(n)$  到  $O(n^2)$  不等，取决于具体问题

"""

```
import sys
import math
from typing import List, Tuple

class AdvancedManacherPython:
 """
 Python 版本的高级 Manacher 算法实现类
 """

 def __init__(self):
 self.logger = self._setup_logger()

 def _setup_logger(self):
 """设置简单的日志记录器"""
 import logging
 logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')
 return logging.getLogger(__name__)

 def palindrome_pairs(self, words: List[str]) -> List[List[int]]:
 """
 LeetCode 336. 回文对
 给定一组互不相同的单词，找出所有不同的索引对(i, j)，使得两个单词拼接起来是回文串。
 """

 # Implementation of the Advanced Manacher algorithm to solve the problem.
```

解题思路：

1. 使用字典树存储所有单词及其反转
2. 对于每个单词，检查其前缀和后缀是否为回文
3. 在字典树中查找剩余部分的匹配

时间复杂度:  $O(n * k^2)$ ，其中  $n$  是单词数量， $k$  是单词平均长度

空间复杂度:  $O(n * k)$

Args:

words: 单词列表

Returns:

回文对列表

"""

```
if not words or len(words) < 2:
 return []

构建字典树
trie = self.TrieNode()
for i, word in enumerate(words):
 self._insert_word(trie, word, i)

result = []

查找回文对
for i, word in enumerate(words):
 # 检查整个单词是否在字典树中
 node = trie
 for j, char in enumerate(word):
 if char not in node.children:
 break
 node = node.children[char]

 # 如果当前节点是某个单词的结尾，且剩余部分是回文
 if node.word_index != -1 and node.word_index != i and self._is_palindrome(word, j
+ 1, len(word) - 1):
 result.append([i, node.word_index])
```

# 检查单词的反转

```
reversed_word = word[::-1]
rev_node = trie
for j, char in enumerate(reversed_word):
 if char not in rev_node.children:
 break
```

```

 rev_node = rev_node.children[char]

 # 如果当前节点是某个单词的结尾，且剩余部分是回文
 if rev_node.word_index != -1 and rev_node.word_index != i and
self._is_palindrome(reversed_word, j + 1, len(reversed_word) - 1):
 result.append([rev_node.word_index, i])

return result

class TrieNode:
 """字典树节点"""
 def __init__(self):
 self.children = {}
 self.word_index = -1

def _insert_word(self, root: TrieNode, word: str, index: int):
 """向字典树中插入单词"""
 node = root
 for char in word:
 if char not in node.children:
 node.children[char] = self.TrieNode()
 node = node.children[char]
 node.word_index = index

def _is_palindrome(self, s: str, left: int, right: int) -> bool:
 """判断子串是否为回文"""
 while left < right:
 if s[left] != s[right]:
 return False
 left += 1
 right -= 1
 return True

def partition(self, s: str) -> List[List[str]]:
 """
 LeetCode 131. 分割回文串
 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。返回所有可能的分割方案。
 """

```

解题思路：

1. 使用 Manacher 算法预处理回文信息
2. 使用回溯法枚举所有分割方案
3. 利用预处理信息快速判断子串是否为回文

时间复杂度:  $O(n * 2^n)$

空间复杂度:  $O(n^2)$

Args:

s: 输入字符串

Returns:

所有分割方案

""""

if not s:

    return []

# 预处理回文信息

is\_palindrome = self.\_preprocess\_palindrome(s)

result = []

self.\_backtrack(s, 0, [], result, is\_palindrome)

return result

def \_preprocess\_palindrome(self, s: str) -> List[List[bool]]:

"""预处理回文信息"""

n = len(s)

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

# 单个字符都是回文

for i in range(n):

    dp[i][i] = True

# 两个字符的情况

for i in range(n - 1):

    dp[i][i + 1] = (s[i] == s[i + 1])

# 长度大于 2 的情况

for length in range(3, n + 1):

    for i in range(n - length + 1):

        j = i + length - 1

        dp[i][j] = (s[i] == s[j] and dp[i + 1][j - 1])

return dp

def \_backtrack(self, s: str, start: int, current: List[str],

                  result: List[List[str]], is\_palindrome: List[List[bool]]):

"""回溯法实现"""

```

if start == len(s):
 result.append(current[:])
 return

for end in range(start, len(s)):
 if is_palindrome[start][end]:
 current.append(s[start:end + 1])
 self._backtrack(s, end + 1, current, result, is_palindrome)
 current.pop()

```

def min\_cut(self, 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。

解题思路：

1. 使用 Manacher 算法预处理回文信息
2. 使用动态规划计算最少分割次数
3. 优化状态转移过程

时间复杂度：O(n<sup>2</sup>)

空间复杂度：O(n<sup>2</sup>)

Args:

s: 输入字符串

Returns:

最少分割次数

"""

if len(s) <= 1:

return 0

# 预处理回文信息

is\_palindrome = self.\_preprocess\_palindrome(s)

n = len(s)

dp = [float('inf')] \* n

for i in range(n):

if is\_palindrome[0][i]:

dp[i] = 0 # 整个子串是回文，不需要分割

else:

for j in range(i):

```

 if is_palindrome[j + 1][i]:
 dp[i] = min(dp[i], dp[j] + 1)

 return int(dp[n - 1]) # 确保返回 int 类型

def longest_palindrome_product(self, s: str, k: int) -> int:
 """
 洛谷 P1659 [国家集训队]拉拉队排练
 求字符串中所有奇数长度回文串的长度乘积。
 """

```

解题思路：

1. 使用 Manacher 算法找到所有奇数长度回文串
2. 统计每个长度的回文串数量
3. 计算前 k 大的长度乘积

时间复杂度：O(n)

空间复杂度：O(n)

Args:

- s: 输入字符串
- k: 前 k 大的长度

Returns:

长度乘积（取模）

```

 """
if not s:
 return 0

n = len(s)
使用 Manacher 算法计算回文半径
radius = self._manacher_odd(s)

统计每个长度的回文串数量
count = [0] * (n + 1)
for i in range(n):
 length = 2 * radius[i] + 1
 count[length] += 1

```

# 计算前 k 大的长度乘积

```

mod = 10**9 + 7
product = 1
remaining = k

```

```

for length in range(n, 0, -2):
 if count[length] > 0 and remaining > 0:
 take = min(count[length], remaining)
 product = (product * self._fast_power(length, take, mod)) % mod
 remaining -= take

return product

def _manacher_odd(self, s: str) -> List[int]:
 """Manacher 算法计算奇回文串"""
 n = len(s)
 radius = [0] * n
 l, r = 0, -1

 for i in range(n):
 k = 1 if i > r else min(radius[1 + r - i], r - i + 1)

 while i - k >= 0 and i + k < n and s[i - k] == s[i + k]:
 k += 1

 radius[i] = k - 1

 if i + k - 1 > r:
 l = i - k + 1
 r = i + k - 1

 return radius

def _fast_power(self, base: int, exponent: int, mod: int) -> int:
 """快速幂计算"""
 result = 1
 while exponent > 0:
 if exponent & 1:
 result = (result * base) % mod
 base = (base * base) % mod
 exponent >>= 1
 return result

def longest_double_palindrome(self, s: str) -> int:
 """
 洛谷 P4555 [国家集训队]最长双回文串
 输入字符串 s，求 s 的最长双回文子串 t 的长度（双回文子串就是可以分成两个回文串的字符串）
 """

```

解题思路：

1. 使用 Manacher 算法预处理
2. 计算每个位置作为分割点的左右最长回文
3. 枚举所有分割点求最大值

时间复杂度：O(n)

空间复杂度：O(n)

Args:

s: 输入字符串

Returns:

最长双回文串长度

"""

```
if len(s) < 2:
 return 0
```

# 预处理字符串

```
processed = self._preprocess_string(s)
n = len(processed)
p = [0] * n
```

```
center, right = 0, 0
```

```
for i in range(1, n - 1):
 mirror = 2 * center - i
```

```
 if i < right:
 p[i] = min(right - i, p[mirror])
```

```
 while (i + p[i] + 1 < n and i - p[i] - 1 >= 0 and
 processed[i + p[i] + 1] == processed[i - p[i] - 1]):
 p[i] += 1
```

```
 if i + p[i] > right:
 center = i
 right = i + p[i]
```

# 计算每个位置的最长回文长度

```
left_max = [0] * len(s)
right_max = [0] * len(s)
```

# 从左到右计算每个位置的最长回文前缀

```
max_len = 0
```

```

for i in range(len(s)):
 pos = 2 * i + 1
 max_len = max(max_len, p[pos])
 left_max[i] = max_len

从右到左计算每个位置的最长回文后缀
max_len = 0
for i in range(len(s) - 1, -1, -1):
 pos = 2 * i + 1
 max_len = max(max_len, p[pos])
 right_max[i] = max_len

枚举分割点，计算最长双回文串
result = 0
for i in range(len(s) - 1):
 result = max(result, left_max[i] + right_max[i + 1])

return result

def _preprocess_string(self, s: str) -> str:
 """预处理字符串，插入特殊字符"""
 if not s:
 return '^$'

 result = ['^']
 for char in s:
 result.extend(['#', char])
 result.extend(['#', '$'])

 return ''.join(result)

def next_palindrome(self, num_str: str) -> str:
 """
 SPOJ PALIN - The Next Palindrome
 给定一个整数，找到大于该数的最小回文数。
 """

```

解题思路：

1. 将数字转换为字符串
2. 构造下一个回文数
3. 处理进位等特殊情况

时间复杂度：O(n)

空间复杂度：O(n)

Args:

num\_str: 数字字符串

Returns:

下一个回文数

"""

```
if not num_str:
 return "1"
```

```
num = list(num_str)
n = len(num)
```

# 处理全 9 的情况

```
if all(char == '9' for char in num):
 return '1' + '0' * (n - 1) + '1'
```

# 构造下一个回文数

```
mid = n // 2
left_smaller = False
i = mid - 1
j = mid if n % 2 == 0 else mid + 1
```

# 跳过已经相同的部分

```
while i >= 0 and num[i] == num[j]:
 i -= 1
 j += 1
```

# 检查是否需要增加中间部分

```
if i < 0 or num[i] < num[j]:
 left_smaller = True
```

# 复制左半部分到右半部分

```
while i >= 0:
 num[j] = num[i]
 i -= 1
 j += 1
```

# 如果需要增加中间部分

```
if left_smaller:
 carry = 1
 i = mid - 1
```

```

if n % 2 == 1:
 mid_num = int(num[mid]) + carry
 carry = mid_num // 10
 num[mid] = str(mid_num % 10)
 j = mid + 1
else:
 j = mid

处理进位
while i >= 0:
 digit = int(num[i]) + carry
 carry = digit // 10
 num[i] = str(digit % 10)
 num[j] = num[i]
 i -= 1
 j += 1

return ''.join(num)

```

def build\_palindrome(self, a: str, b: str) -> int:

"""

HackerRank Build a Palindrome

给定两个字符串 a 和 b，从 a 中取一个非空前缀，从 b 中取一个非空后缀，拼接成一个回文串，求最长的回文串长度。

解题思路：

1. 使用 Manacher 算法预处理两个字符串
2. 枚举所有可能的前缀后缀组合
3. 检查拼接后的字符串是否为回文

时间复杂度：O(n<sup>2</sup>)

空间复杂度：O(n)

Args:

- a: 字符串 a
- b: 字符串 b

Returns:

最长回文串长度

"""

max\_len = 0

n, m = len(a), len(b)

```

枚举所有可能的前缀后缀组合
for i in range(n):
 for j in range(m):
 # 从 a 取前缀，从 b 取后缀
 prefix = a[:i + 1]
 suffix = b[m - j - 1:]
 combined = prefix + suffix

 # 检查是否为回文
 if self._is_palindrome(combined, 0, len(combined) - 1):
 max_len = max(max_len, len(combined))

return max_len

```

```

def who_says_pun(self, s: str) -> int:
 """

```

AtCoder ABC141E - Who Says a Pun?

给定一个长度为 n 的字符串 s，找出两个不重叠的子串，使得它们相等且长度尽可能大。

解题思路：

1. 使用 Z 函数计算每个后缀的匹配情况
2. 遍历所有可能的分割点
3. 找到满足条件的最长子串

时间复杂度：O( $n^2$ )

空间复杂度：O(n)

Args:

s: 输入字符串

Returns:

最大长度

"""

n = len(s)

max\_len = 0

# 对于每个可能的起始位置

for i in range(n):

# 计算从位置 i 开始的 Z 函数

z = self.\_z\_function(s[i:])

# 查找不重叠的相同子串

for j in range(1, len(z)):

```

 if z[j] > max_len and j >= z[j]:
 max_len = max(max_len, z[j])

 return max_len

def _z_function(self, s: str) -> List[int]:
 """Z 函数计算"""
 n = len(s)
 z = [0] * n
 z[0] = n

 l, r = 0, 0
 for i in range(1, n):
 if i <= r:
 z[i] = min(r - i + 1, z[i - 1])

 while i + z[i] < n and s[z[i]] == s[i + z[i]]:
 z[i] += 1

 if i + z[i] - 1 > r:
 l = i
 r = i + z[i] - 1

 return z

def run_unit_tests(self):
 """运行单元测试"""
 print("===== Python 版本高级 Manacher 算法题目测试 =====")

 # 测试回文对
 print("\n1. 回文对测试:")
 words = ["abcd", "dcba", "lls", "s", "sssll"]
 result = self.palindrome_pairs(words)
 print(f"回文对结果: {result}")

 # 测试分割回文串
 print("\n2. 分割回文串测试:")
 s = "aab"
 result = self.partition(s)
 print(f"分割方案数量: {len(result)}")

 # 测试最少分割次数
 print("\n3. 最少分割次数测试:")

```

```

s = "aab"
result = self.min_cut(s)
print(f"最少分割次数: {result}")

测试最长双回文串
print("\n4. 最长双回文串测试:")
s = "baacaabbacabb"
result = self.longest_double_palindrome(s)
print(f"最长双回文串长度: {result}")

测试下一个回文数
print("\n5. 下一个回文数测试:")
num = "12345"
result = self.next_palindrome(num)
print(f"下一个回文数: {result}")

print("\n===== 测试完成 =====")

```

```

def main():
 """主函数"""
 solver = AdvancedManacherPython()
 solver.run_unit_tests()

if __name__ == "__main__":
 main()

```

=====

文件: AlgorithmTester.java

```

=====
package class103;

import java.util.Arrays;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * 算法测试器 - 用于验证 Manacher 算法和 Z 函数实现的正确性
 * 提供便捷的测试入口和比较功能
 */
public class AlgorithmTester {
 private static final Logger logger = Logger.getLogger(AlgorithmTester.class.getName());

```

```
public static void main(String[] args) {
 logger.info("开始测试 Manacher 算法和 Z 函数...");

 // 测试 Manacher 算法
 testManacher();

 // 测试 Z 函数
 testZFunction();

 logger.info("所有测试完成!");
}
```

```
/**
 * 测试 Manacher 算法的功能
 */
private static void testManacher() {
 logger.info("===== 测试 Manacher 算法 =====");

 try {
 // 启用调试模式
 Code01_Manacher.setDebugMode(true);

 // 测试用例
 String[] testCases = {
 "babad",
 "cbbd",
 "a",
 "ac",
 "racecar",
 "",
 "aaa"
 };
```

```
 // 预期的最长回文子串结果
 String[] expectedPalindromes = {
 "bab", // 或 "aba"
 "bb",
 "a",
 "a", // 或 "c"
 "racecar",
 "",
 "aaa"
 };
```

```

// 预期的回文子串数量
int[] expectedCounts = {
 7, // b, a, b, a, d, bab, aba
 5, // c, b, b, d, bb
 1, // a
 2, // a, c
 10, // r, a, c, e, c, a, r, aceca, cec, racecar
 0, // 空字符串
 6 // a, a, aa, aa, aaa
};

Code01_Manacher manacher = new Code01_Manacher();

for (int i = 0; i < testCases.length; i++) {
 String s = testCases[i];
 logger.info("测试用例[" + (i+1) + "]: \"" + s + "\"");

 // 测试最长回文子串
 String result = manacher.longestPalindrome(s);
 String expected = expectedPalindromes[i];
 boolean passed = result.equals(expected) ||
 (s.equals("babad") && result.equals("aba")) ||
 (s.equals("ac") && result.equals("c"));

 logger.info("最长回文子串: " + result + " (" + (passed ? "通过" : "失败") +
")");

 // 测试回文子串计数
 int countResult = manacher.countSubstrings(s);
 boolean countPassed = countResult == expectedCounts[i];
 logger.info("回文子串数量: " + countResult + " (" + (countPassed ? "通过" : "失败") +
")");

 // 测试最短回文串 (对于非空字符串)
 if (!s.isEmpty()) {
 String shortestPalindrome = manacher.shortestPalindrome(s);
 boolean isPalindrome = isPalindrome(shortestPalindrome);
 boolean startsWith = shortestPalindrome.endsWith(s);
 logger.info("最短回文串: " + shortestPalindrome + " (" + (isPalindrome ? "是回文: " +
isPalindrome + ", 以原串结尾: " + startsWith + ")"));
 }
}

```

```
 logger.info("-----");
 }

 // 运行内置单元测试
 logger.info("运行 Manacher 内置单元测试... ");
 manacher.runUnitTests();

} catch (Exception e) {
 logger.log(Level.SEVERE, "Manacher 算法测试出错: " + e.getMessage(), e);
} finally {
 // 关闭调试模式
 Code01_Manacher.setDebugMode(false);
}

}

/***
 * 测试 Z 函数的功能
 */
private static void testZFunction() {
 logger.info("===== 测试 Z 函数 (扩展 KMP) =====");

 try {
 // 启用调试模式
 Code02_ExpandKMP.setDebugMode(true);

 // 测试 Z 数组计算
 testZArrayCalculation();

 // 测试 LeetCode 2223
 testLeetCode2223();

 // 测试 LeetCode 3031
 testLeetCode3031();

 // 运行内置单元测试
 logger.info("运行 Z 函数内置单元测试... ");
 Code02_ExpandKMP.runUnitTests();

 } catch (Exception e) {
 logger.log(Level.SEVERE, "Z 函数测试出错: " + e.getMessage(), e);
 } finally {
 // 关闭调试模式
 Code02_ExpandKMP.setDebugMode(false);
 }
}
```

```
 }

 }

/***
 * 测试 Z 数组计算
 */
private static void testZArrayCalculation() {
 logger.info("测试 Z 数组计算:");

 String[] testCases = {
 "aaaaa",
 "ababc",
 "babab",
 "abcdef",
 "aabaa"
 };

 int[][] expectedResults = {
 {5, 4, 3, 2, 1},
 {5, 0, 2, 0, 1},
 {5, 0, 3, 0, 1},
 {6, 0, 0, 0, 0},
 {5, 1, 0, 1, 0}
 };
}

// 确保数组足够大
if (Code02_ExpandKMP.MAXN < 100) {
 Code02_ExpandKMP.MAXN = 100;
 Code02_ExpandKMP.z = Arrays.copyOf(Code02_ExpandKMP.z, Code02_ExpandKMP.MAXN);
}

for (int i = 0; i < testCases.length; i++) {
 String s = testCases[i];
 char[] chars = s.toCharArray();
 int n = s.length();

 logger.info("字符串: \"" + s + "\"");

 // 计算 Z 数组
 Code02_ExpandKMP.zArray(chars, n);

 // 获取计算结果
 int[] result = Arrays.copyOf(Code02_ExpandKMP.z, n);
}
```

```

int[] expected = expectedResults[i];

boolean passed = Arrays.equals(result, expected);
logger.info("Z 数组: " + Arrays.toString(result));
logger.info("结果: " + (passed ? "通过" : "失败"));

if (!passed) {
 logger.info("期望: " + Arrays.toString(expected));
}

logger.info("-----");
}

/**
 * 测试 LeetCode 2223 - 构造字符串的总得分和
 */
private static void testLeetCode2223() {
 logger.info("测试 LeetCode 2223 - 构造字符串的总得分和:");

 String[] testCases = {
 "babab",
 "azbazbzaz",
 "a",
 "aaa"
 };

 long[] expectedResults = {
 9,
 14,
 1,
 6
 };

 for (int i = 0; i < testCases.length; i++) {
 String s = testCases[i];
 long result = Code02_ExpandKMP.sumScores(s);
 long expected = expectedResults[i];
 boolean passed = result == expected;

 logger.info("字符串: \\" + s + "\\" -> 得分: " + result + " (" + (passed ? "通过" : "失败") + ")");
 }
}

```

```
 if (!passed) {
 logger.info("期望得分: " + expected);
 }
 }

 logger.info("-----");
}

/***
 * 测试 LeetCode 3031 - 将单词恢复初始状态所需的最短时间 II
 */
private static void testLeetCode3031() {
 logger.info("测试 LeetCode 3031 - 将单词恢复初始状态所需的最短时间 II:");

 String[] words = {
 "abacaba",
 "abacaba",
 "abcdef",
 "a",
 "aaa"
 };

 int[] ks = {
 3,
 4,
 2,
 1,
 2
 };

 int[] expectedResults = {
 2,
 1,
 3,
 1,
 1
 };

 for (int i = 0; i < words.length; i++) {
 String word = words[i];
 int k = ks[i];
 int result = Code02_ExpandKMP.minimumTimeToInitialState(word, k);
 int expected = expectedResults[i];
 if (result != expected) {
 logger.error("Test failed for word: " + word + ", k: " + k + ". Expected: " + expected + ", Result: " + result);
 } else {
 logger.info("Test passed for word: " + word + ", k: " + k + ". Result: " + result);
 }
 }
}
```

```
 boolean passed = result == expected;

 logger.info("word: " + word + "\", k: " + k + " -> 时间: " + result + " (" +
(passed ? "通过" : "失败") + ")");
 }

 if (!passed) {
 logger.info("期望时间: " + expected);
 }
}

logger.info("-----");
}

/**
 * 辅助方法: 检查字符串是否为回文
 */
private static boolean isPalindrome(String s) {
 if (s == null || s.isEmpty()) {
 return true;
 }

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

 while (left < right) {
 if (s.charAt(left) != s.charAt(right)) {
 return false;
 }
 left++;
 right--;
 }

 return true;
}

/**
 * 性能对比测试
 */
public static void performanceComparison() {
 logger.info("===== 性能对比测试 =====");

 try {
 // 启用性能监控

```

```

Code01_Manacher.setPerformanceMonitoring(true);
Code02_ExpandKMP.setPerformanceMonitoring(true);

// 生成测试数据
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 100000; i++) {
 sb.append((char) ('a' + (i % 26)));
}
String longString = sb.toString();

logger.info("测试字符串长度: " + longString.length());

// 测试 Manacher 性能
long startTime = System.currentTimeMillis();
Code01_Manacher manacher = new Code01_Manacher();
manacher.longestPalindrome(longString);
long manacherTime = System.currentTimeMillis() - startTime;
logger.info("Manacher 算法耗时: " + manacherTime + " ms");

// 测试 Z 函数性能
startTime = System.currentTimeMillis();
Code02_ExpandKMP.sumScores(longString);
long zFunctionTime = System.currentTimeMillis() - startTime;
logger.info("Z 函数耗时: " + zFunctionTime + " ms");

} catch (Exception e) {
 logger.log(Level.SEVERE, "性能测试出错: " + e.getMessage(), e);
} finally {
 // 关闭性能监控
 Code01_Manacher.setPerformanceMonitoring(false);
 Code02_ExpandKMP.setPerformanceMonitoring(false);
}
}
}

=====

文件: Code01_Manacher.java
=====

package class103;

// Manacher 算法模版 - 高效解决回文子串问题
// 求字符串 s 中最长回文子串的长度

```

```
// 测试链接 : https://www.luogu.com.cn/problem/P3805
// 提交时请把类名改成"Main", 可以直接通过

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.logging.Logger;
import java.util.logging.Level;

/**
 * Manacher 算法的全面实现类
 * 提供高效的回文子串检测、计数和处理功能
 * 包含多个算法题目的最优解实现
 * 时间复杂度: O(n) - 线性时间复杂度, 每个字符最多被访问两次
 * 空间复杂度: O(n) - 需要预处理和辅助数组
 *
 * 算法应用场景:
 * 1. LeetCode 5. 最长回文子串 - https://leetcode.com/problems/longest-palindromic-substring/
 * 2. LeetCode 647. 回文子串 - https://leetcode.com/problems/palindromic-substrings/
 * 3. LeetCode 214. 最短回文串 - https://leetcode.com/problems/shortest-palindrome/
 * 4. 洛谷 P3805 【模板】manacher - https://www.luogu.com.cn/problem/P3805
 * 5. UVa 11475 - Extend to Palindrome -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2470
 * 6. Codeforces 1326D2 - Prefix-Suffix Palindrome -
https://codeforces.com/problemset/problem/1326/D2
 * 7. HackerRank - Palindromic Substrings
 * 8. AcWing 141. 周期 - https://www.acwing.com/problem/content/143/
 * 9. POJ 3240 - 回文串
 * 10. LeetCode 336. 回文对 - https://leetcode.com/problems/palindrome-pairs/
 * 11. LeetCode 131. 分割回文串 - https://leetcode.com/problems/palindrome-partitioning/
 * 12. LeetCode 132. 分割回文串 II - https://leetcode.com/problems/palindrome-partitioning-ii/
*/
public class Code01_Manacher {
 // 日志记录器 - 用于调试和问题定位
 private static final Logger logger = Logger.getLogger(Code01_Manacher.class.getName());
 // 性能监控开关
 private static boolean performanceMonitoring = false;
```

```
// 最大字符串长度常量 - 预分配内存以提高性能
public static int MAXN = 11000001;

// 预处理后的字符串数组
public static char[] ss = new char[MAXN << 1];

// 回文半径数组
public static int[] p = new int[MAXN << 1];

// 当前处理的字符串长度
public static int n;

/**
 * 主方法 - 用于在线评测系统
 * 使用高效的输入输出方式以处理大规模数据
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
 // 高效 IO 处理 - 处理大规模输入时性能优化
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

 // 读取输入并处理
 out.println(manacher(in.readLine()));

 // 确保所有输出被刷新
 out.flush();
 out.close();
 in.close();
}

/**
 * 增强版主方法 - 用于本地测试和调试
 * 提供更丰富的测试功能和调试信息
 * @param args 命令行参数
 */
public static void enhancedMain(String[] args) {
 testAllFunctionality();
}

/**
```

```

* 全面功能测试方法
* 测试所有实现的算法功能
*/
public static void testAllFunctionality() {
 System.out.println("===== Manacher 算法功能测试 =====");

 // 测试用例集
 String[] testCases = {
 "abc12321cba", // 包含回文的字符串
 "", // 空字符串
 "a", // 单个字符
 "aaaaa", // 全相同字符
 "abcdefg" // 无回文
 };

 // 测试最长回文子串
 System.out.println("\n1. 最长回文子串测试:");
 for (String s : testCases) {
 System.out.println("输入: \"" + s + "\"");
 System.out.println("结果: \"" + longestPalindrome(s) + "\"");
 }

 // 测试回文子串计数
 System.out.println("\n2. 回文子串计数测试:");
 for (String s : testCases) {
 System.out.println("输入: \"" + s + "\"");
 System.out.println("结果: " + countSubstrings(s));
 }

 // 测试最短回文串
 System.out.println("\n3. 最短回文串测试:");
 String[] shortestTestCases = {"aacecaaa", "abcd", ""};
 for (String s : shortestTestCases) {
 System.out.println("输入: \"" + s + "\"");
 System.out.println("结果: \"" + shortestPalindrome(s) + "\"");
 }

 System.out.println("\n===== 测试完成 =====");
}

/**
 * Manacher 算法主函数，用于计算字符串中最长回文子串的长度
 * 这是算法竞赛中常用的优化实现版本

```

```

*
* 算法核心原理:
* 1. 预处理: 在原字符串的每个字符之间插入特殊字符'#'，并在首尾也添加'#'
* 这样可以将奇数长度和偶数长度的回文串统一处理为奇数长度的回文串
* 2. 利用回文串的对称性，避免重复计算 - 这是算法线性时间复杂度的关键
* 3. 维护当前最右回文边界 r 和对应的中心 c，通过已计算的信息加速新位置的计算
*
* 时间复杂度证明:
* - 每个字符最多被访问两次: 第一次作为中心扩展，第二次作为右侧字符被检查
* - 虽然有嵌套循环，但内层循环的总执行次数被右边界 r 的单调增长所限制
* - 因此整体时间复杂度为 O(n)
*
* 空间复杂度: O(n)，需要预处理字符串数组和回文半径数组
*
* @param str 输入字符串
* @return 最长回文子串的长度
* @throws IllegalArgumentException 如果输入为 null
*/
public static int manacher(String str) {
 // 参数验证 - 防御性编程
 if (str == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }

 // 边界情况快速处理
 if (str.isEmpty()) {
 return 0;
 }

 long startTime = 0;
 if (performanceMonitoring) {
 startTime = System.nanoTime();
 }

 // 预处理字符串
 manacherss(str.toCharArray());

 int max = 0;
 // c: 当前最右回文子串的中心
 // r: 当前最右回文子串的右边界
 // len: 当前位置 i 为中心的回文半径
 for (int i = 0, c = 0, r = 0, len; i < n; i++) {
 // 调试辅助 - 记录中间变量状态
 }
}

```

```

if (logger.isLoggable(Level.FINER)) {
 logger.finer("i=" + i + ", c=" + c + ", r=" + r);
}

// 算法核心优化步骤：利用回文对称性减少重复计算
// 有三种情况：
// 1. i 在 r 外：无法利用对称性，初始半径为 1
// 2. i 在 r 内且对称点的回文完全在 c 的回文内：直接使用对称点的回文半径
// 3. i 在 r 内但对称点的回文部分超出 c 的回文：使用 r-i 作为初始半径
len = r > i ? Math.min(p[2 * c - i], r - i) : 1;

// 断言验证 - 确保 len 的有效性（调试时使用）
assert len > 0 : "回文半径必须为正整数";

// 尝试扩展回文串
// 从当前半径开始，尝试向两边扩展
while (i + len < n && i - len >= 0 && ss[i + len] == ss[i - len]) {
 len++;
 // 性能监控点 - 极端情况下可以监控扩展次数
 if (performanceMonitoring && len > 10000) {
 logger.info("大规模回文扩展：" + len);
 }
}

// 更新最右回文边界和中心
// 这里体现了算法的贪心策略：总是维护最右的回文边界
if (i + len > r) {
 r = i + len;
 c = i;
}

// 更新最大回文半径
if (len > max) {
 max = len;
 // 记录最长回文的位置信息（可选）
 if (logger.isLoggable(Level.FINE)) {
 logger.fine("找到更长回文，中心位置：" + i + "，半径：" + len);
 }
}

// 保存当前位置的回文半径
p[i] = len;
}

```

```
// 性能统计
if (performanceMonitoring) {
 long endTime = System.nanoTime();
 logger.info("Manacher 算法执行时间: " + (endTime - startTime) + " 纳秒");
}

// 由于我们插入了'#'字符，实际回文长度是半径减 1
return max - 1;
}

/**
 * 预处理函数，将原字符串转换为插入'#'的格式
 * 例如: "abc" -> "#a#b#c#"
 *
 * 预处理的目的:
 * 1. 统一处理奇数长度和偶数长度的回文串
 * 2. 避免处理边界情况时的数组越界检查
 * 3. 确保每个回文串的中心都是一个字符，而不是字符之间的间隙
 *
 * @param a 原字符串的字符数组
 * @throws IllegalArgumentException 如果输入为 null
 */
public static void manacherss(char[] a) {
 // 参数验证
 if (a == null) {
 throw new IllegalArgumentException("输入字符数组不能为 null");
 }

 // 计算预处理后的字符串长度
 // 原长度 n -> 2n+1 (每个字符间插入'#')
 n = a.length * 2 + 1;

 // 边界检查 - 避免数组越界
 if (n > ss.length) {
 // 如果预分配的数组不够大，需要重新分配
 // 注意：在实际竞赛中通常预分配足够大的空间
 ss = new char[n];
 p = new int[n];
 }

 // 填充预处理后的字符串
 // 使用位运算 (i & 1) 比取模运算 (i % 2) 效率更高
```

```

 for (int i = 0, j = 0; i < n; i++) {
 // 偶数位置放'#'，奇数位置放原字符
 ss[i] = (i & 1) == 0 ? '#' : a[j++];
 }

 // 调试信息
 if (logger.isLoggable(Level.FINER)) {
 logger.finer("预处理后的字符串: " + new String(ss, 0, n));
 }
 }

 /**
 * 设置性能监控开关
 * @param enabled 是否启用性能监控
 */
 public static void setPerformanceMonitoring(boolean enabled) {
 performanceMonitoring = enabled;
 }

 /**
 * 调试辅助方法：打印算法执行的详细过程
 * 用于算法学习和问题定位
 *
 * @param str 输入字符串
 */
 public static void debugProcess(String str) {
 System.out.println("\n===== Manacher 算法执行过程调试 =====");
 System.out.println("原始字符串: " + str);

 // 预处理字符串
 char[] a = str.toCharArray();
 manacherss(a);
 System.out.println("预处理后: " + new String(ss, 0, n));

 // 重置 p 数组
 Arrays.fill(p, 0);

 int max = 0;
 for (int i = 0, c = 0, r = 0, len; i < n; i++) {
 System.out.println("\n处理位置 i = " + i + ", 字符 = '" + ss[i] + "'");
 System.out.println("当前状态: c = " + c + ", r = " + r);

 len = r > i ? Math.min(p[2 * c - i], r - i) : 1;
 if (ss[i] != '#') {
 if (r <= i + len - 1) {
 p[2 * c - i] = len;
 r = i + len;
 } else {
 p[2 * c - i] = len - (r - i);
 r = i + len - p[2 * c - i];
 }
 } else {
 if (r <= i + len - 1) {
 p[2 * c - i] = len;
 r = i + len;
 } else {
 p[2 * c - i] = len - (r - i);
 r = i + len - p[2 * c - i];
 }
 }
 }
 }
}

```

```

System.out.println("初始 len = " + len + " (基于对称性优化)");

while (i + len < n && i - len >= 0 && ss[i + len] == ss[i - len]) {

 len++;

 System.out.println("扩展 len 到 " + len);

}

if (i + len > r) {

 r = i + len;

 c = i;

 System.out.println("更新 r = " + r + ", c = " + c);

}

if (len > max) {

 max = len;

 System.out.println("更新最大回文半径: " + max);

}

p[i] = len;

System.out.println("p[" + i + "] = " + len);

}

System.out.println("\n最终最大回文长度: " + (max - 1));

System.out.println("===== 调试结束 =====");
}

/**

 * LeetCode 5. 最长回文子串

 * 给你一个字符串 s，找到 s 中最长的回文子串

 *

 * 示例 1:

 * 输入: s = "babad"

 * 输出: "bab" 或 "aba"

 *

 * 示例 2:

 * 输入: s = "cbbd"

 * 输出: "bb"

 *

 * 算法分析:

 * - 时间复杂度: O(n) - Manacher 算法的线性时间复杂度

 * - 空间复杂度: O(n) - 需要存储预处理字符串和回文半径数组

 *

 * 解题思路:

```

```
* 1. 预处理字符串，插入特殊字符统一处理奇偶数长度回文
* 2. 使用 Manacher 算法计算每个位置的回文半径
* 3. 找到最长回文的中心位置和半径
* 4. 将处理后的位置映射回原字符串，提取对应的子串
*
* 优化点：
* - 使用 try-catch 处理边界情况，提高鲁棒性
* - 预处理时添加^和$边界符，避免额外的边界检查
*
* @param s 输入字符串
* @return 最长回文子串
* @throws IllegalArgumentException 如果输入为 null
*/
public static String longestPalindrome(String s) {
 // 参数验证
 if (s == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }

 // 边界情况快速处理
 if (s.length() <= 1) {
 return s;
 }

 // 使用 Manacher 算法处理
 String processed = preprocess(s);
 char[] chars = processed.toCharArray();
 int len = chars.length;
 int[] radius = new int[len];

 int center = 0, right = 0;
 int maxLen = 0, centerIndex = 0;

 for (int i = 0; i < len; i++) {
 // 算法核心：利用回文对称性减少重复计算
 int mirror = 2 * center - i;

 if (i < right) {
 // 三种情况的统一处理：
 // 1. 对称点的回文完全在当前回文内部
 // 2. 对称点的回文部分超出当前回文
 // 3. 对称点的回文恰好到达当前回文边界
 radius[i] = Math.min(right - i, radius[mirror]);
 }
 }
}
```

```

}

// 尝试扩展回文
// 使用 try-catch 处理边界情况，比每次检查更高效
try {
 while (i - radius[i] - 1 >= 0 &&
 i + radius[i] + 1 < len &&
 chars[i - radius[i] - 1] == chars[i + radius[i] + 1]) {
 radius[i]++;
 }
} catch (ArrayIndexOutOfBoundsException e) {
 // 边界情况处理 - 当数组越界时停止扩展
 // 在生产环境中，应该记录异常信息
 if (logger.isLoggable(Level.FINEST)) {
 logger.finest("边界扩展异常: " + e.getMessage());
 }
}

// 更新最右边界和中心
if (i + radius[i] > right) {
 center = i;
 right = i + radius[i];
}

// 更新最长回文信息
if (radius[i] > maxLen) {
 maxLen = radius[i];
 centerIndex = i;
}
}

// 从处理后的字符串中提取原始回文子串
// 映射公式：原始起始位置 = (处理后的中心 - 最大半径) / 2
int start = (centerIndex - maxLen) / 2;

// 安全检查 - 确保索引有效
start = Math.max(0, start);
int end = Math.min(s.length(), start + maxLen);

String result = s.substring(start, end);

if (logger.isLoggable(Level.FINE)) {
 logger.fine("最长回文子串: " + result + ", 长度: " + result.length());
}

```

```

 }

 return result;
}

/***
 * LeetCode 647. 回文子串
 * 给定一个字符串，计算其中回文子串的数目
 *
 * 示例：
 * 输入： "abc" -> 输出： 3 ("a", "b", "c")
 * 输入： "aaa" -> 输出： 6 ("a", "a", "a", "aa", "aa", "aaa")
 *
 * 算法分析：
 * - 时间复杂度：O(n) - Manacher 算法的线性时间复杂度
 * - 空间复杂度：O(n) - 需要存储预处理字符串和回文半径数组
 *
 * 数学原理：
 * 对于处理后的字符串中的每个位置 i，回文半径为 r：
 * - 如果 i 是奇数位置（对应原字符串的字符），则贡献 r 个回文子串
 * - 如果 i 是偶数位置（对应插入的'#'），则贡献 r 个回文子串
 * 通过公式 (radius[i] + 1) / 2 可以统一计算贡献数量
 *
 * 优化点：
 * 利用预处理和 Manacher 算法避免了暴力枚举的 O(n2) 时间复杂度
 * 使用异常处理简化边界检查逻辑
 *
 * @param s 输入字符串
 * @return 回文子串的数目
 * @throws IllegalArgumentException 如果输入为 null
 */
public static int countSubstrings(String s) {
 // 参数验证
 if (s == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }

 // 边界情况快速处理
 if (s.isEmpty()) {
 return 0;
 }
 if (s.length() == 1) {
 return 1;
 }
}

```

```
}
```

```
// 预处理字符串
String processed = preprocess(s);
char[] chars = processed.toCharArray();
int len = chars.length;
int[] radius = new int[len];
```

```
int center = 0, right = 0;
int count = 0;
```

```
for (int i = 0; i < len; i++) {
 // 利用回文对称性优化
 int mirror = 2 * center - i;

 if (i < right) {
 radius[i] = Math.min(right - i, radius[mirror]);
 }
```

```
// 扩展回文
try {
 while (i - radius[i] - 1 >= 0 &&
 i + radius[i] + 1 < len &&
 chars[i - radius[i] - 1] == chars[i + radius[i] + 1]) {
 radius[i]++;
 }
} catch (ArrayIndexOutOfBoundsException e) {
 // 边界处理
}
```

```
// 更新右边界
if (i + radius[i] > right) {
 center = i;
 right = i + radius[i];
}
```

```
// 计算当前位置贡献的回文子串数量
// 数学公式: (radius[i] + 1) / 2
// 例如: radius=3 时, 贡献 2 个回文子串
// radius=4 时, 贡献 2 个回文子串
int contribution = (radius[i] + 1) / 2;
count += contribution;
```

```

 if (logger.isLoggable(Level.FINEST)) {
 logger.finest("位置 i=" + i + ", 半径=" + radius[i] + ", 贡献=" + contribution);
 }
 }

 if (logger.isLoggable(Level.FINE)) {
 logger.fine("字符串: " + s + ", 回文子串总数: " + count);
 }

 return count;
}

/**
 * LeetCode 214. 最短回文串
 * 给定一个字符串 s，可以通过在字符串前面添加字符将其转换为回文串。
 * 找到并返回可以用这种方式转换的最短回文串。
 *
 * 示例：
 * 输入： "aacecaaa" -> 输出： "aaacecaaa"
 * 输入： "abcd" -> 输出： "dcbabcd"
 *
 * 算法分析：
 * - 时间复杂度：O(n) - KMP 算法的线性时间复杂度
 * - 空间复杂度：O(n) - 需要存储组合字符串和 LPS 数组
 *
 * 解题思路（KMP 方法）：
 * 1. 问题转化：找到 s 的最长前缀，该前缀也是 s 的后缀
 * 2. 构造字符串：s + "#" + reverse(s)，使用 '#' 确保匹配不会跨边界
 * 3. 计算 LPS 数组：找到最长公共前后缀
 * 4. 构建结果：将未匹配部分反转后添加到原字符串前
 *
 * 为什么 KMP 比 Manacher 更适合此题：
 * - 题目要求的是前缀回文，而不是任意位置的回文
 * - KMP 算法能更直接地找到字符串的前缀后缀匹配关系
 * - 实现更简洁，且时间复杂度同样为 O(n)
 *
 * @param s 输入字符串
 * @return 最短回文串
 * @throws IllegalArgumentException 如果输入为 null
 */
public static String shortestPalindrome(String s) {
 // 参数验证
 if (s == null) {

```

```
 throw new IllegalArgumentException("输入字符串不能为 null");
 }

 // 边界情况快速处理
 if (s.length() <= 1) {
 return s;
 }

 // 核心思路：找到 s 的最长前缀，该前缀也是 s 的后缀
 // 构造组合字符串：原字符串 + 特殊字符 + 反转后的字符串
 // 使用'#'作为分隔符，确保匹配不会跨过原字符串和反转字符串的边界
 String reversed = new StringBuilder(s).reverse().toString();
 String combined = s + "#" + reversed;

 // 计算 LPS 数组
 int[] lps = computeLPS(combined);

 // LPS 数组的最后一个元素表示原字符串的最长前缀回文长度
 int maxPrefixPalindromeLen = lps[combined.length() - 1];

 // 构建最短回文串：
 // 1. 取出原字符串中不属于最长前缀回文的部分
 // 2. 反转这部分
 // 3. 添加到原字符串前面
 String suffixToReverse = s.substring(maxPrefixPalindromeLen);
 String reversedPrefix = new StringBuilder(suffixToReverse).reverse().toString();
 String result = reversedPrefix + s;

 if (logger.isLoggable(Level.FINE)) {
 logger.fine("原始字符串：" + s);
 logger.fine("最长前缀回文长度：" + maxPrefixPalindromeLen);
 logger.fine("最短回文串：" + result);
 }

 return result;
}

/**
 * KMP 算法中的 LPS 数组计算（最长公共前后缀数组）
 * LPS[i] 表示字符串 pattern[0...i] 的最长公共前后缀长度
 *
 * 算法原理：
 * - i 从 1 开始遍历字符串

```

```
* - j 表示当前已匹配的前缀长度
* - 当字符匹配时， j 增加并记录到 lps[i]
* - 当字符不匹配时，回退 j 到 lps[j-1]
*
* 时间复杂度: O(m)，其中 m 为模式串长度
*
* @param pattern 模式字符串
* @return LPS 数组
* @throws IllegalArgumentException 如果输入为 null
*/
private static int[] computeLPS(String pattern) {
 // 参数验证
 if (pattern == null) {
 throw new IllegalArgumentException("模式字符串不能为 null");
 }

 int len = pattern.length();
 int[] lps = new int[len];

 // 边界情况处理
 if (len == 0) {
 return lps;
 }

 // i: 当前处理的位置
 // j: 当前已匹配的前缀长度
 int i = 1, j = 0;

 while (i < len) {
 if (pattern.charAt(i) == pattern.charAt(j)) {
 // 字符匹配，增加已匹配长度并记录
 lps[i++] = ++j;
 } else {
 // 字符不匹配
 if (j != 0) {
 // 回退到前一个可能的匹配位置
 j = lps[j - 1];
 } else {
 // 无法回退，当前位置 LPS 值为 0
 lps[i++] = 0;
 }
 }
 }
}
```

```
if (logger.isLoggable(Level.FINER)) {
 logger.finest("模式串: " + pattern);
 logger.finest("LPS 数组: " + Arrays.toString(lps));
}

return lps;
}

/**
 * 预处理函数，用于在字符间插入'#'
 * 同时在首尾添加^和$作为边界标记
 *
 * 预处理格式: "abc" -> "^#a#b#c#$"
 *
 * 预处理的优点:
 * 1. 统一处理奇数长度和偶数长度的回文串
 * 2. ^和$边界符避免了扩展时的边界检查
 * 3. 确保回文扩展不会越界
 *
 * @param s 原始字符串
 * @return 预处理后的字符串
 * @throws IllegalArgumentException 如果输入为 null
 */
private static String preprocess(String s) {
 // 参数验证
 if (s == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }

 int n = s.length();
 if (n == 0) {
 return "^$";
 }

 // 使用 StringBuilder 提高字符串拼接效率
 StringBuilder sb = new StringBuilder(2 * n + 3); // 预分配足够空间
 sb.append("^"); // 开头边界符

 for (int i = 0; i < n; i++) {
 sb.append("#").append(s.charAt(i));
 }
}
```

```

 sb.append("#$").append("$"); // 结尾边界符

 return sb.toString();
}

/**
 * 洛谷 P3805 【模板】manacher
 * 题目描述：给出一个只由小写英文字符 a, b, c, ..., y, z 组成的字符串 S ,
 * 求 S 中最长回文串的长度 。
 *
 * 输入格式：一行小写英文字符 a, b, c, ..., y, z 组成的字符串 S
 * 输出格式：一个整数表示答案
 *
 * 时间复杂度：O(n) - 线性时间算法
 * 空间复杂度：O(n) - 需要 O(n) 的辅助空间
 *
 * 解题思路：
 * 1. 直接使用 Manacher 算法模板
 * 2. 返回计算得到的最长回文子串长度
 *
 * @param s 输入字符串
 * @return 最长回文子串的长度
 * @throws IllegalArgumentException 如果输入为 null
 */
public static int longestPalindromeLength(String s) {
 return manacher(s);
}

/**
 * 单元测试方法
 * 用于验证所有算法功能的正确性
 * 支持自动化测试和持续集成
 *
 * @return 测试是否全部通过
 */
public static boolean runUnitTests() {
 boolean allPassed = true;

 // 1. 测试最长回文子串
 System.out.println("\n===== 最长回文子串测试 =====");
 String[][] palindromeTests = {
 {"babad", "bab"}, // 奇数长度回文，有多个解
 {"cbbd", "bb"}, // 偶数长度回文
 };
}

```

```

 {"a", "a"}, // 单个字符
 {"", ""}, // 空字符串
 {"aaaaa", "aaaaa"} // 全相同字符
 };

for (int i = 0; i < palindromeTests.length; i++) {
 String input = palindromeTests[i][0];
 String expected = palindromeTests[i][1];
 String result = longestPalindrome(input);
 boolean passed = result.equals(expected) ||
 (expected.equals("bab") && result.equals("aba")); // 处理多个正确答案

 System.out.println("测试 " + (i + 1) + ":" + (passed ? "通过" : "失败"));
 if (!passed) allPassed = false;
}

// 2. 测试回文子串计数
System.out.println("\n===== 回文子串计数测试 =====");
Object[][] countTests = {
 {"abc", 3}, // 基本测试
 {"aaa", 6}, // 全相同字符
 {"a", 1}, // 单个字符
 {"", 0} // 空字符串
};

for (int i = 0; i < countTests.length; i++) {
 String input = (String) countTests[i][0];
 int expected = (int) countTests[i][1];
 int result = countSubstrings(input);
 boolean passed = result == expected;

 System.out.println("测试 " + (i + 1) + ":" + (passed ? "通过" : "失败"));
 if (!passed) allPassed = false;
}

// 3. 测试最短回文串
System.out.println("\n===== 最短回文串测试 =====");
String[][] shortestTests = {
 {"aacecaaa", "aaacecaaa"}, // 有部分前缀回文
 {"abcd", "dcbabcd"}, // 无前缀回文
 {"a", "a"}, // 单个字符
 {"", ""} // 空字符串
};

```

```

 for (int i = 0; i < shortestTests.length; i++) {
 String input = shortestTests[i][0];
 String expected = shortestTests[i][1];
 String result = shortestPalindrome(input);
 boolean passed = result.equals(expected);

 System.out.println("测试 " + (i + 1) + ": " + (passed ? "通过" : "失败"));
 if (!passed) allPassed = false;
 }

 System.out.println("\n单元测试结果: " + (allPassed ? "全部通过" : "存在失败"));
 return allPassed;
 }
}
=====
```

文件: Code02\_ExpandKMP.java

```
=====
package class103;

// 扩展 KMP 模版，又称 Z 算法或 Z 函数
// 给定两个字符串 a、b，求出两个数组
// b 与 b 每一个后缀串的最长公共前缀长度，z 数组
// b 与 a 每一个后缀串的最长公共前缀长度，e 数组
// 计算出要求的两个数组后，输出这两个数组的权值即可
// 对于一个数组 x，i 位置的权值定义为：(i * (x[i] + 1))
// 数组权值为所有位置权值的异或和
// 测试链接：https://www.luogu.com.cn/problem/P5410
// 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 code，提交时请把类名改成"Main"，可以直接通过

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Arrays;
import java.util.logging.Level;
```

```
import java.util.logging.Logger;

/**
 * 扩展 KMP 算法 (Z 函数) 工程化实现
 *
 * 该类提供了 Z 函数 (扩展 KMP) 的高效实现，并集成了以下工程化特性：
 * 1. 参数验证与异常处理
 * 2. 性能监控与日志记录
 * 3. 可配置的调试选项
 * 4. 单元测试支持
 * 5. 边界情况处理
 *
 * 算法应用场景：
 * 1. LeetCode 2223. 构造字符串的总得分和 - https://leetcode.com/problems/sum-of-scores-of-built-strings/
 * 2. LeetCode 3031. 将单词恢复初始状态所需的最短时间 II - https://leetcode.com/problems/minimum-time-to-revert-word-to-initial-state-ii/
 * 3. Codeforces 126B. Password - https://codeforces.com/problemset/problem/126/B
 * 4. 洛谷 P5410 【模板】扩展 KMP/exKMP (Z 函数) - https://www.luogu.com.cn/problem/P5410
 * 5. SPOJ - Pattern Find
 * 6. HackerEarth - String Similarity - https://www.hackerearth.com/practice/algorithms/string-algorithm/z-algorithm/tutorial/
 * 7. AtCoder ABC141E - Who Says a Pun? - https://atcoder.jp/contests/abc141/tasks/abc141_e
 * 8. USACO 2011 November Contest, Bronze - Cow Photographs
 * 9. 牛客网 NC15051 - 字符串的匹配
 */
public class Code02_ExpandKMP {
 // 日志记录器
 private static final Logger logger = Logger.getLogger(Code02_ExpandKMP.class.getName());

 // 性能监控开关
 private static boolean performanceMonitoring = false;

 // 调试信息开关
 private static boolean debugMode = false;

 // 预分配的最大数组大小
 public static int MAXN = 20000001;

 // Z 数组和 E 数组
 public static int[] z = new int[MAXN];
 public static int[] e = new int[MAXN];
```

```
/**
 * 设置性能监控状态
 */
public static void setPerformanceMonitoring(boolean enabled) {
 performanceMonitoring = enabled;
 logger.info("性能监控已" + (enabled ? "开启" : "关闭"));
}

/**
 * 设置调试模式状态
 */
public static void setDebugMode(boolean enabled) {
 debugMode = enabled;
 logger.info("调试模式已" + (enabled ? "开启" : "关闭"));
}

/**
 * 记录性能日志
 */
private static void logPerformance(String operation, long startTime, long endTime) {
 if (performanceMonitoring) {
 long duration = endTime - startTime;
 logger.info(String.format("操作 [%s] 耗时: %d ms", operation, duration));
 }
}

/**
 * 记录调试日志
 */
private static void logDebug(String message) {
 if (debugMode) {
 logger.info("[调试] " + message);
 }
}

public static void main(String[] args) {
 // 初始化日志记录器
 logger.info("Z 算法 (扩展 KMP) 程序启动 - " +
 LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")));

 // 解析命令行参数
 for (String arg : args) {
 if (arg.equals("--debug")) {
 logger.info("命令行参数 --debug 被检测到");
 }
 }
}
```

```
 setDebugMode(true);
 } else if (arg.equals("--performance")) {
 setPerformanceMonitoring(true);
 } else if (arg.equals("--test")) {
 runUnitTests();
 return;
 }
}

try {
 BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

 char[] a = null;
 char[] b = null;

 // 读取输入
 String line = in.readLine();
 if (line != null) {
 a = line.toCharArray();
 line = in.readLine();
 if (line != null) {
 b = line.toCharArray();
 }
 }
}

// 参数验证
if (a == null || b == null) {
 throw new IllegalArgumentException("输入数据不完整, 请提供两个字符串");
}

logDebug("输入字符串 a: " + new String(a));
logDebug("输入字符串 b: " + new String(b));

// 确保数组大小足够
ensureArrayCapacity(Math.max(a.length, b.length));

// 执行算法并记录性能
long startTime = System.currentTimeMillis();
zArray(b, b.length);
logPerformance("Z 数组计算", startTime, System.currentTimeMillis());

startTime = System.currentTimeMillis();
```

```

eArray(a, b, a.length, b.length);
logPerformance("E 数组计算", startTime, System.currentTimeMillis());

// 计算并输出结果
long resultZ = eor(z, b.length);
long resultE = eor(e, a.length);

out.println(resultZ);
out.println(resultE);
out.flush();

logDebug("Z 数组权值异或和: " + resultZ);
logDebug("E 数组权值异或和: " + resultE);

// 关闭资源
in.close();
out.close();

} catch (IOException e) {
 logger.log(Level.SEVERE, "IO 异常: " + e.getMessage(), e);
} catch (IllegalArgumentException e) {
 logger.log(Level.SEVERE, "参数错误: " + e.getMessage(), e);
 System.err.println("错误: " + e.getMessage());
} catch (Exception e) {
 logger.log(Level.SEVERE, "未预期的异常: " + e.getMessage(), e);
}

logger.info("程序执行完成 - " +
LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")));
}

/**
 * 确保预分配的数组容量足够大
 */
private static void ensureArrayCapacity(int requiredSize) {
 if (requiredSize > MAXN) {
 MAXN = requiredSize + 100; // 添加一些缓冲
 z = Arrays.copyOf(z, MAXN);
 e = Arrays.copyOf(e, MAXN);
 logDebug("已调整数组大小为: " + MAXN);
 }
}

```

```

/***
 * Z 函数计算
 * Z 函数 z[i] 表示字符串 s 从位置 i 开始与字符串 s 从位置 0 开始的最长公共前缀长度
 *
 * 算法原理：
 * 1. 维护一个匹配区间 [l, r]，表示当前已知的最右匹配区间
 * 2. 对于当前位置 i，如果 i <= r，可以利用已计算的信息优化
 * 3. 利用对称性，z[i] 至少为 min(r - i + 1, z[i - 1])
 * 4. 在此基础之上继续向右扩展匹配
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @param n 字符串长度
 * @throws IllegalArgumentException 当输入参数无效时抛出
 */
// 非常像 Manacher 算法
public static void zArray(char[] s, int n) {
 // 参数验证
 if (s == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }
 if (n < 0 || n > s.length) {
 throw new IllegalArgumentException("字符串长度无效：" + n);
 }

 // 处理边界情况
 if (n == 0) {
 return;
 }

 logDebug("开始计算 Z 数组，字符串长度：" + n);

 z[0] = n;
 // l: 当前最右匹配区间的左边界
 // r: 当前最右匹配区间的右边界
 // len: 当前位置的 Z 值（最长公共前缀长度）
 for (int i = 1, c = 1, r = 1, len; i < n; i++) {
 // 利用已计算的信息优化
 // 如果 i 在当前匹配区间内
 len = r > i ? Math.min(r - i, z[i - c]) : 0;
 ...
 }
}

```

```

 // 继续向右扩展匹配
 // 使用局部变量 s 来减少数组访问开销
 final char[] localS = s;
 while (i + len < n && localS[i + len] == localS[len]) {
 len++;
 }

 // 更新最右匹配区间
 if (i + len > r) {
 r = i + len;
 c = i;
 }

 z[i] = len;
 }

 if (debugMode && n <= 1000) { // 避免大数据量时日志过多
 logDebug("Z 数组计算完成, 前 10 个值: " +
 Arrays.toString(Arrays.copyOfRange(z, 0, Math.min(n, 10))));
 }
}

/**
 * 扩展 KMP 计算
 * 计算字符串 a 的每个后缀与字符串 b 的最长公共前缀长度
 *
 * 时间复杂度: O(n + m), 其中 n 是 a 的长度, m 是 b 的长度
 * 空间复杂度: O(n + m)
 *
 * @param a 字符串 a
 * @param b 字符串 b
 * @param n 字符串 a 的长度
 * @param m 字符串 b 的长度
 * @throws IllegalArgumentException 当输入参数无效时抛出
 */
// 非常像 Manacher 算法
public static void eArray(char[] a, char[] b, int n, int m) {
 // 参数验证
 if (a == null || b == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }
 if (n < 0 || n > a.length || m < 0 || m > b.length) {
 throw new IllegalArgumentException("字符串长度无效");
 }
}

```

```

}

// 处理边界情况
if (n == 0 || m == 0) {
 return;
}

logDebug("开始计算 E 数组, a 长度: " + n + ", b 长度: " + m);

// 使用局部变量减少数组访问开销
final char[] localA = a;
final char[] localB = b;
final int[] localZ = z;
final int[] localE = e;

for (int i = 0, c = 0, r = 0, len; i < n; i++) {
 // 利用已计算的信息优化
 len = r > i ? Math.min(r - i, localZ[i - c]) : 0;

 // 继续向右扩展匹配
 while (i + len < n && len < m && localA[i + len] == localB[len]) {
 len++;
 }

 // 更新最右匹配区间
 if (i + len > r) {
 r = i + len;
 c = i;
 }
}

localE[i] = len;
}

if (debugMode && n <= 1000) { // 避免大数据量时日志过多
 logDebug("E 数组计算完成, 前 10 个值: " +
 Arrays.toString(Arrays.copyOfRange(e, 0, Math.min(n, 10))));
}
}

/***
 * 计算数组的权值异或和
 * 对于数组中的每个元素 arr[i], 权值为 (i+1) * (arr[i] + 1)
 */

```

```

* @param arr 输入数组
* @param n 数组长度
* @return 所有权值的异或和
* @throws IllegalArgumentException 当输入参数无效时抛出
*/
public static long eor(int[] arr, int n) {
 // 参数验证
 if (arr == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 if (n < 0 || n > arr.length) {
 throw new IllegalArgumentException("数组长度无效: " + n);
 }

 // 处理边界情况
 if (n == 0) {
 return 0;
 }

 long ans = 0;
 // 使用局部变量减少数组访问开销
 final int[] localArr = arr;

 for (int i = 0; i < n; i++) {
 ans ^= (long) (i + 1) * (localArr[i] + 1);
 }

 return ans;
}

/**
 * LeetCode 2223. 构造字符串的总得分和
 * 你需要从空字符串开始构造一个长度为 n 的字符串 s，构造过程为每次给当前字符串前面添加一个字符。
 * 构造过程中得到的所有字符串编号为 1 到 n，其中长度为 i 的字符串编号为 si。
 * si 的得分为 si 和 sn 的最长公共前缀的长度（注意 s == sn）。
 * 请你返回每一个 si 的得分之和。
 *
 * 示例：
 * 输入：s = "babab"
 * 输出：9
 * 解释：
 * s1 == "b"，得分 1

```

```

* s2 == "ab", 得分 0
* s3 == "bab", 得分 3
* s4 == "abab", 得分 0
* s5 == "babab", 得分 5
* 总和为 1+0+3+0+5=9
*
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*
* @param s 输入字符串
* @return 得分总和
* @throws IllegalArgumentException 当输入参数无效时抛出
*/
public static long sumScores(String s) {
 // 参数验证
 if (s == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }

 logDebug("计算构造字符串的总得分和, 输入: " + s);

 int n = s.length();
 char[] chars = s.toCharArray();

 // 计算 Z 函数
 int[] z = new int[n];
 z[0] = n;

 long sum = n; // s5 的得分就是整个字符串的长度

 for (int i = 1, l = 0, r = 0; i < n; i++) {
 // 利用之前计算的结果
 if (i <= r) {
 z[i] = Math.min(r - i + 1, z[i - 1]);
 }

 // 扩展匹配
 while (i + z[i] < n && chars[z[i]] == chars[i + z[i]]) {
 z[i]++;
 }

 // 更新匹配区间
 if (i + z[i] - 1 > r) {

```

```

 l = i;
 r = i + z[i] - 1;
}

// 累加得分
sum += z[i];
}

logDebug("总得分和计算结果: " + sum);
return sum;
}

/**
 * LeetCode 3031. 将单词恢复初始状态所需的最短时间 II
 * 给你一个下标从 0 开始的字符串 word 和一个整数 k。
 * 每一秒执行以下操作：
 * 1. 移除 word 的前 k 个字符
 * 2. 在 word 的末尾添加 k 个任意字符
 * 返回将 word 恢复到初始状态所需的最短时间（该时间必须大于零）。
 *
 * 示例：
 * 输入：word = "abacaba", k = 3
 * 输出：2
 * 解释：
 * 第 1 秒后，word 变成"acaba**"（用*表示添加的字符）
 * 第 2 秒后，word 变成"aba****"
 * 如果添加的字符分别为"cac"和"caba"，word 就恢复为"abacaba"
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param word 输入字符串
 * @param k 每次操作移除和添加的字符数
 * @return 恢复初始状态所需的最短时间
 * @throws IllegalArgumentException 当输入参数无效时抛出
 */
public static int minimumTimeToInitialState(String word, int k) {
 // 参数验证
 if (word == null) {
 throw new IllegalArgumentException("输入字符串不能为 null");
 }
 if (k <= 0) {
 throw new IllegalArgumentException("k 必须为正整数: " + k);
 }
}

```

```

}

logDebug("计算恢复初始状态所需时间, word: " + word + ", k: " + k);

int n = word.length();
char[] chars = word.toCharArray();

// 计算 Z 函数
int[] z = new int[n];
z[0] = n;

for (int i = 1, l = 0, r = 0; i < n; i++) {
 if (i <= r) {
 z[i] = Math.min(r - i + 1, z[i - 1]);
 }

 while (i + z[i] < n && chars[z[i]] == chars[i + z[i]]) {
 z[i]++;
 }

 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
}

// 查找满足条件的最短时间
for (int i = k; i < n; i += k) {
 // 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
 // 说明在第(i/k)步后可以恢复原字符串
 if (z[i] >= n - i) {
 int result = i / k;
 logDebug("找到最短时间: " + result);
 return result;
 }
}

// 最坏情况需要完全替换
int worstCase = (n + k - 1) / k;
logDebug("最坏情况时间: " + worstCase);
return worstCase;
}

```

```

/**
 * Codeforces 126B. Password
 *
 * 题目描述:
 * 给定一个字符串 s, 找出最长的子串 t, 它既是 s 的前缀, 也是 s 的后缀, 还在 s 的中间出现过。
 * 如果存在这样的子串, 输出最长的那个; 否则输出"Just a legend"。
 *
 * 解题思路:
 * 使用 Z 函数 (扩展 KMP) 算法解决此问题:
 * 1. 计算字符串 s 的 Z 函数数组 z, 其中 z[i] 表示以位置 i 开始的后缀与原字符串的最长公共前缀长度
 * 2. 遍历 z 数组, 找到既是前缀又是后缀的子串 (即 z[i] == n-i 的情况)
 * 3. 同时记录在中间出现过的前缀长度
 * 4. 找到满足所有条件的最长子串
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * @param s 输入字符串
 * @return 满足条件的最长子串, 如果不存在则返回"Just a legend"
 */
public static String solvePassword(String s) {
 int n = s.length();
 if (n <= 2) return "Just a legend";

 // 计算 Z 函数
 int[] z = zFunction(s);

 // 记录在中间出现过的前缀长度
 boolean[] hasPrefix = new boolean[n + 1];

 // 标记在中间出现过的前缀长度
 for (int i = 1; i < n; i++) {
 if (z[i] > 0) {
 hasPrefix[z[i]] = true;
 }
 }

 // 查找既是前缀又是后缀且在中间出现过的最长子串
 int maxLen = 0;
 for (int i = 1; i < n; i++) {
 // 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
 // 说明这个后缀与原字符串的前缀完全匹配
 if (z[i] == n - i && hasPrefix[z[i]]) {

```

```

 maxLen = Math.max(maxLen, z[i]);
 }
}

// 如果找到了满足条件的子串，返回它；否则返回"Just a legend"
return maxLen > 0 ? s.substring(0, maxLen) : "Just a legend";
}

/***
 * Z 函数计算
 * Z 函数 z[i] 表示字符串 s 从位置 i 开始与字符串 s 从位置 0 开始的最长公共前缀长度
 *
 * 算法原理：
 * 1. 维护一个匹配区间 [l, r]，表示当前已知的最右匹配区间
 * 2. 对于当前位置 i，如果 i <= r，可以利用已计算的信息优化
 * 3. 利用对称性，z[i] 至少为 min(r - i + 1, z[i - 1])
 * 4. 在此基础之上继续向右扩展匹配
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @return Z 函数数组
 */
public static int[] zFunction(String s) {
 int n = s.length();
 int[] z = new int[n];
 z[0] = n;

 // l: 当前最右匹配区间的左边界
 // r: 当前最右匹配区间的右边界
 for (int i = 1, l = 0, r = 0; i < n; i++) {
 // 利用已计算的信息优化
 // 如果 i 在当前匹配区间内
 if (i <= r) {
 z[i] = Math.min(r - i + 1, z[i - 1]);
 }

 // 继续向右扩展匹配
 while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
 z[i]++;
 }
 }
}

```

```
// 更新最右匹配区间
 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
}

return z;
}

/***
 * 运行单元测试
 */
public static void runUnitTests() {
 logger.info("开始运行单元测试...");

 // 测试 Z 函数
 testZArray();

 // 测试 LeetCode 2223
 testSumScores();

 // 测试 LeetCode 3031
 testMinimumTimeToInitialState();

 // 测试 Codeforces 126B
 testSolvePassword();

 logger.info("单元测试完成");
}

/***
 * 测试 Z 数组计算
 */
private static void testZArray() {
 String[] testCases = {
 "aaaaa",
 "ababc",
 "babab",
 "abcdef"
 };

 int[][] expectedResults = {
```

```
{5, 4, 3, 2, 1},
{5, 0, 2, 0, 1},
{5, 0, 3, 0, 1},
{6, 0, 0, 0, 0}
};

for (int i = 0; i < testCases.length; i++) {
 String s = testCases[i];
 char[] chars = s.toCharArray();
 int n = s.length();

 // 确保数组足够大
 if (n > MAXN) {
 MAXN = n;
 z = Arrays.copyOf(z, MAXN);
 }

 // 计算 Z 数组
 zArray(chars, n);

 // 验证结果
 boolean passed = true;
 for (int j = 0; j < n; j++) {
 if (z[j] != expectedResults[i][j]) {
 passed = false;
 logger.warning("Z 数组测试失败: s=" + s + ", 位置=" + j + ", 期望=" +
expectedResults[i][j] + ", 实际=" + z[j]);
 break;
 }
 }

 if (passed) {
 logger.info("Z 数组测试通过: s=" + s);
 }
}

/**
 * 测试 LeetCode 2223
 */
private static void testSumScores() {
 String[] testCases = {"babab", "azbazbzaz"};
 long[] expectedResults = {9, 14};
```

```

 for (int i = 0; i < testCases.length; i++) {
 long result = sumScores(testCases[i]);
 if (result == expectedResults[i]) {
 logger.info("sumScores 测试通过: s=" + testCases[i] + ", 结果=" + result);
 } else {
 logger.warning("sumScores 测试失败: s=" + testCases[i] + ", 期望=" +
expectedResults[i] + ", 实际=" + result);
 }
 }

 }

/**
 * 测试 LeetCode 3031
 */
private static void testMinimumTimeToInitialState() {
 String[] words = {"abacaba", "abacaba", "abcdef"};
 int[] ks = {3, 4, 2};
 int[] expectedResults = {2, 1, 3};

 for (int i = 0; i < words.length; i++) {
 int result = minimumTimeToInitialState(words[i], ks[i]);
 if (result == expectedResults[i]) {
 logger.info("minimumTimeToInitialState 测试通过: word=" + words[i] + ", k=" + ks[i] +
", 结果=" + result);
 } else {
 logger.warning("minimumTimeToInitialState 测试失败: word=" + words[i] + ", k=" +
ks[i] + ", 期望=" + expectedResults[i] + ", 实际=" + result);
 }
 }
}

/**
 * 测试 Codeforces 126B
 */
private static void testSolvePassword() {
 String[] testCases = {"fixprefixsuffix", "abcdabc", "abcab"};
 String[] expectedResults = {"fix", "Just a legend", "ab"};

 for (int i = 0; i < testCases.length; i++) {
 String result = solvePassword(testCases[i]);
 if (result.equals(expectedResults[i])) {
 logger.info("solvePassword 测试通过: s=" + testCases[i] + ", 结果=" + result);
 }
 }
}

```

```
 } else {
 logger.warning("solvePassword 测试失败: s=" + testCases[i] + ", 期望=" +
expectedResults[i] + ", 实际=" + result);
 }
 }
}
```

---

文件: Code03\_MinimumTimeToInitialStateII.java

---

```
package class103;

// 将单词恢复初始状态所需的最短时间 II
// 给你一个下标从 0 开始的字符串 word 和一个整数 k
// 在每一秒，必须执行以下操作
// 移除 word 的前 k 个字符
// 在 word 的末尾添加 k 个任意字符
// 添加的字符不必和移除的字符相同
// 返回将 word 恢复到初始状态所需的最短时间
// 该时间必须大于零
// 测试链接 : https://leetcode.cn/problems/minimum-time-to-revert-word-to-initial-state-ii/
public class Code03_MinimumTimeToInitialStateII {

 /**
 * 使用 Z 函数解决将单词恢复初始状态所需的最短时间问题
 *
 * 算法思路:
 * 1. 每次操作移除前 k 个字符，相当于在字符串上以步长 k 向前移动
 * 2. 我们需要找到最小的移动次数，使得剩余的后缀能够通过添加字符恢复为原字符串
 * 3. 这等价于找到最小的 i (i 是 k 的倍数)，使得 word.substring(i) 与 word 的最长公共前缀
 * 等于 word.substring(i) 的长度
 * 4. 使用 Z 函数可以高效计算每个后缀与原字符串的最长公共前缀长度
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * @param word 输入字符串
 * @param k 每次操作移除/添加的字符数
 * @return 恢复初始状态所需的最短时间
 */
 public static int minimumTimeToInitialState(String word, int k) {
```

```

char[] s = word.toCharArray();
int n = s.length;
zArray(s, n);
for (int i = k; i < n; i += k) {
 // 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
 // 说明在第(i/k)步后可以恢复原字符串
 if (z[i] == n - i) {
 return i / k;
 }
}
// 最坏情况需要完全替换
return (n + k - 1) / k;
}

```

// leetcode 增加了数据量

// 所以把这个值改成  $10^6$  规模

```
public static int MAXN = 1000001;
```

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

```
/**
```

\* Z 函数计算

\* Z 函数  $z[i]$  表示字符串  $s$  从位置  $i$  开始与字符串  $s$  从位置 0 开始的最长公共前缀长度

\*

\* 算法原理:

- \* 1. 维护一个匹配区间  $[l, r]$ , 表示当前已知的最右匹配区间
- \* 2. 对于当前位置  $i$ , 如果  $i \leq r$ , 可以利用已计算的信息优化
- \* 3. 利用对称性,  $z[i]$  至少为  $\min(r - i + 1, z[i - 1])$
- \* 4. 在此基础之上继续向右扩展匹配

\*

\* 时间复杂度:  $O(n)$

\* 空间复杂度:  $O(n)$

\*/

```
public static void zArray(char[] s, int n) {
```

```
 z[0] = n;
```

// l: 当前最右匹配区间的左边界

// r: 当前最右匹配区间的右边界

// len: 当前位置的 Z 值 (最长公共前缀长度)

```
 for (int i = 1, c = 1, r = 1, len; i < n; i++) {
```

// 利用已计算的信息优化

// 如果 i 在当前匹配区间内

```
 len = r > i ? Math.min(r - i, z[i - c]) : 0;
```

// 继续向右扩展匹配

```

 while (i + len < n && s[i + len] == s[len]) {
 len++;
 }
 // 更新最右匹配区间
 if (i + len > r) {
 r = i + len;
 c = i;
 }
 z[i] = len;
 }
}

/***
 * LeetCode 2223. 构造字符串的总得分和（使用相同的 Z 函数）
 * 你需要从空字符串开始构造一个长度为 n 的字符串 s，构造过程为每次给当前字符串前面添加一个字符。
 * 构造过程中得到的所有字符串编号为 1 到 n，其中长度为 i 的字符串编号为 si。
 * si 的得分为 si 和 sn 的最长公共前缀的长度（注意 s == sn）。
 * 请你返回每一个 si 的得分之和。
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 */
public static long sumScores(String word) {
 char[] s = word.toCharArray();
 int n = s.length;

 // 复用 zArray 方法计算 Z 函数
 zArray(s, n);

 // 计算得分总和
 long sum = 0;
 for (int i = 0; i < n; i++) {
 // 每个 si 的得分就是 z[i]
 sum += z[i];
 }

 return sum;
}

/***
 * 洛谷 P5410 【模板】扩展 KMP（Z 函数）
 * 题目描述：给定两个字符串 a, b，求：
 */

```

```

* 1. b 与 b 每一个后缀串的最长公共前缀长度 (即 b 的 Z 函数)
* 2. a 与 b 每一个后缀串的最长公共前缀长度 (即扩展 KMP)
*
* 输入格式:
* 第一行输入一个字符串 a
* 第二行输入一个字符串 b
*
* 输出格式:
* 第一行输出 b 的 Z 函数的异或和
* 第二行输出 a 与 b 的扩展 KMP 的异或和
*
* 时间复杂度: O(n + m)
* 空间复杂度: O(n + m)
*/

```

```

public static long[] extendedKMP(String a, String b) {
 char[] aChars = a.toCharArray();
 char[] bChars = b.toCharArray();
 int n = a.length();
 int m = b.length();

 // 计算 b 的 Z 函数
 zArray(bChars, m);

 // 计算 a 与 b 的扩展 KMP
 int[] e = new int[n];
 for (int i = 0, c = 0, r = 0, len; i < n; i++) {
 // 利用已计算的信息优化
 len = r > i ? Math.min(r - i, z[i - c]) : 0;
 // 继续向右扩展匹配
 while (i + len < n && len < m && aChars[i + len] == bChars[len]) {
 len++;
 }
 // 更新最右匹配区间
 if (i + len > r) {
 r = i + len;
 c = i;
 }
 e[i] = len;
 }

 // 计算异或和
 long zXor = 0;
 for (int i = 0; i < m; i++) {

```

```

 zXor ^= (long) (i + 1) * (z[i] + 1);
}

long eXor = 0;
for (int i = 0; i < n; i++) {
 eXor ^= (long) (i + 1) * (e[i] + 1);
}

return new long[] {zXor, eXor};
}
}

```

---

文件: Codeforces126B\_Password.java

---

```

package class103;

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

/**
 * Codeforces 126B. Password
 *
 * 题目描述:
 * 给定一个字符串 s, 找出最长的子串 t, 它既是 s 的前缀, 也是 s 的后缀, 还在 s 的中间出现过。
 * 如果存在这样的子串, 输出最长的那个; 否则输出"Just a legend"。
 *
 * 解题思路:
 * 使用 Z 函数 (扩展 KMP) 算法解决此问题:
 * 1. 计算字符串 s 的 Z 函数数组 z, 其中 z[i] 表示以位置 i 开始的后缀与原字符串的最长公共前缀长度
 * 2. 遍历 z 数组, 找到既是前缀又是后缀的子串 (即 z[i] == n-i 的情况)
 * 3. 同时记录在中间出现过的前缀长度
 * 4. 找到满足所有条件的最长子串
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * 题目链接: https://codeforces.com/problemset/problem/126/B
 * 相关题目:
 * 1. LeetCode 28. 找出字符串中第一个匹配项的下标 - https://leetcode.com/problems/find-the-index-of-the-first-occurrence-in-a-string/
 * 2. LeetCode 214. 最短回文串 - https://leetcode.com/problems/shortest-palindrome/

```

```

* 3. LeetCode 459. 重复的子字符串 - https://leetcode.com/problems/repeated-substring-pattern/
* 4. SPOJ - Pattern Find
* 5. HackerEarth - String Similarity
* 6. AtCoder ABC141E - Who Says a Pun?
*/
public class Codeforces126B_Password {

 /**
 * 使用 Z 函数解决 Codeforces 126B Password 问题
 *
 * @param s 输入字符串
 * @return 满足条件的最长子串，如果不存在则返回"Just a legend"
 */
 public static String solve(String s) {
 int n = s.length();
 if (n <= 2) return "Just a legend";

 // 计算 Z 函数
 int[] z = zFunction(s);

 // 记录在中间出现过的前缀长度
 boolean[] hasPrefix = new boolean[n + 1];

 // 标记在中间出现过的前缀长度
 for (int i = 1; i < n; i++) {
 if (z[i] > 0) {
 hasPrefix[z[i]] = true;
 }
 }

 // 查找既是前缀又是后缀且在中间出现过的最长子串
 int maxLen = 0;
 for (int i = 1; i < n; i++) {
 // 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
 // 说明这个后缀与原字符串的前缀完全匹配
 if (z[i] == n - i && hasPrefix[z[i]]) {
 maxLen = Math.max(maxLen, z[i]);
 }
 }

 // 如果找到了满足条件的子串，返回它；否则返回"Just a legend"
 return maxLen > 0 ? s.substring(0, maxLen) : "Just a legend";
 }
}

```

```

/**
 * Z 函数计算
 * Z 函数 z[i] 表示字符串 s 从位置 i 开始与字符串 s 从位置 0 开始的最长公共前缀长度
 *
 * 算法原理：
 * 1. 维护一个匹配区间 [l, r]，表示当前已知的最右匹配区间
 * 2. 对于当前位置 i，如果 i <= r，可以利用已计算的信息优化
 * 3. 利用对称性，z[i] 至少为 min(r - i + 1, z[i - 1])
 * 4. 在此基础之上继续向右扩展匹配
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @return Z 函数数组
 */
public static int[] zFunction(String s) {
 int n = s.length();
 int[] z = new int[n];
 z[0] = n;

 // l: 当前最右匹配区间的左边界
 // r: 当前最右匹配区间的右边界
 for (int i = 1, l = 0, r = 0; i < n; i++) {
 // 利用已计算的信息优化
 // 如果 i 在当前匹配区间内
 if (i <= r) {
 z[i] = Math.min(r - i + 1, z[i - 1]);
 }

 // 继续向右扩展匹配
 while (i + z[i] < n && s.charAt(z[i]) == s.charAt(i + z[i])) {
 z[i]++;
 }

 // 更新最右匹配区间
 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
 }
}

```

```

 return z;
 }

// 测试方法
public static void main(String[] args) {
 // 示例测试
 System.out.println(solve("fixprefixsuffix")); // 输出: fix
 System.out.println(solve("abcdabc")); // 输出: Just a legend
 System.out.println(solve("abcab")); // 输出: ab
}
}

```

=====

文件: codeforces\_126b\_password.cpp

=====

```

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

/***
 * Z 函数计算
 * Z 函数 $z[i]$ 表示字符串 s 从位置 i 开始与字符串 s 从位置 0 开始的最长公共前缀长度
 *
 * 算法原理:
 * 1. 维护一个匹配区间 $[l, r]$, 表示当前已知的最右匹配区间
 * 2. 对于当前位置 i , 如果 $i \leq r$, 可以利用已计算的信息优化
 * 3. 利用对称性, $z[i]$ 至少为 $\min(r - i + 1, z[i - 1])$
 * 4. 在此基础之上继续向右扩展匹配
 *
 * 时间复杂度: $O(n)$
 * 空间复杂度: $O(n)$
 *
 * @param s 输入字符串
 * @return Z 函数数组
 */
vector<int> zFunction(string s) {
 int n = s.length();
 vector<int> z(n, 0);
 z[0] = n;
}

```

```

// l: 当前最右匹配区间的左边界
// r: 当前最右匹配区间的右边界
int l = 0, r = 0;

for (int i = 1; i < n; i++) {
 // 利用已计算的信息优化
 // 如果 i 在当前匹配区间内
 if (i <= r) {
 z[i] = min(r - i + 1, z[i - 1]);
 }

 // 继续向右扩展匹配
 while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
 z[i]++;
 }

 // 更新最右匹配区间
 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
}

return z;
}

/**
 * Codeforces 126B. Password
 *
 * 题目描述:
 * 给定一个字符串 s, 找出最长的子串 t, 它既是 s 的前缀, 也是 s 的后缀, 还在 s 的中间出现过。
 * 如果存在这样的子串, 输出最长的那个; 否则输出"Just a legend"。
 *
 * 解题思路:
 * 使用 Z 函数 (扩展 KMP) 算法解决此问题:
 * 1. 计算字符串 s 的 Z 函数数组 z, 其中 z[i] 表示以位置 i 开始的后缀与原字符串的最长公共前缀长度
 * 2. 遍历 z 数组, 找到既是前缀又是后缀的子串 (即 z[i] == n-i 的情况)
 * 3. 同时记录在中间出现过的前缀长度
 * 4. 找到满足所有条件的最长子串
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */

```

```

* @param s 输入字符串
* @return 满足条件的最长子串，如果不存在则返回"Just a legend"
*/
string solve(string s) {
 int n = s.length();
 if (n <= 2) return "Just a legend";

 // 计算 Z 函数
 vector<int> z = zFunction(s);

 // 记录在中间出现过的前缀长度
 vector<bool> hasPrefix(n + 1, false);

 // 标记在中间出现过的前缀长度
 for (int i = 1; i < n; i++) {
 if (z[i] > 0) {
 hasPrefix[z[i]] = true;
 }
 }

 // 查找既是前缀又是后缀且在中间出现过的最长子串
 int maxLen = 0;
 for (int i = 1; i < n; i++) {
 // 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
 // 说明这个后缀与原字符串的前缀完全匹配
 if (z[i] == n - i && hasPrefix[z[i]]) {
 maxLen = max(maxLen, z[i]);
 }
 }

 // 如果找到了满足条件的子串，返回它；否则返回"Just a legend"
 return maxLen > 0 ? s.substr(0, maxLen) : "Just a legend";
}

// 测试方法
int main() {
 // 示例测试
 cout << solve("fixprefixsuffix") << endl; // 输出: fix
 cout << solve("abcdabc") << endl; // 输出: Just a legend
 cout << solve("abcab") << endl; // 输出: ab

 return 0;
}

```

文件: codeforces\_126b\_password.py

```
=====
```

```
def z_function(s):
```

```
 """
```

```
 Z 函数计算
```

```
 Z 函数 $z[i]$ 表示字符串 s 从位置 i 开始与字符串 s 从位置 0 开始的最长公共前缀长度
```

算法原理:

1. 维护一个匹配区间  $[l, r]$ , 表示当前已知的最右匹配区间
2. 对于当前位置  $i$ , 如果  $i \leq r$ , 可以利用已计算的信息优化
3. 利用对称性,  $z[i]$  至少为  $\min(r - i + 1, z[i - 1])$
4. 在此基础之上继续向右扩展匹配

时间复杂度:  $O(n)$

空间复杂度:  $O(n)$

Args:

```
 s: 输入字符串
```

Returns:

```
 Z 函数数组
```

```
 """
```

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

```
z = [0] * n
```

```
z[0] = n
```

```
l: 当前最右匹配区间的左边界
```

```
r: 当前最右匹配区间的右边界
```

```
l = r = 0
```

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

```
 # 利用已计算的信息优化
```

```
 # 如果 i 在当前匹配区间内
```

```
 if i <= r:
```

```
 z[i] = min(r - i + 1, z[i - 1])
```

```
 # 继续向右扩展匹配
```

```
 while i + z[i] < n and s[z[i]] == s[i + z[i]]:
```

```
 z[i] += 1
```

```
更新最右匹配区间
if i + z[i] - 1 > r:
 l = i
 r = i + z[i] - 1

return z
```

```
def solve(s):
"""
Codeforces 126B. Password
```

### 题目描述：

给定一个字符串 s，找出最长的子串 t，它既是 s 的前缀，也是 s 的后缀，还在 s 的中间出现过。如果存在这样的子串，输出最长的那个；否则输出"Just a legend"。

### 解题思路：

使用 Z 函数（扩展 KMP）算法解决此问题：

1. 计算字符串 s 的 Z 函数数组 z，其中  $z[i]$  表示以位置  $i$  开始的后缀与原字符串的最长公共前缀长度
2. 遍历 z 数组，找到既是前缀又是后缀的子串（即  $z[i] == n-i$  的情况）
3. 同时记录在中间出现过的前缀长度
4. 找到满足所有条件的最长子串

时间复杂度： $O(n)$

空间复杂度： $O(n)$

### Args:

s: 输入字符串

### Returns:

满足条件的最长子串，如果不存在则返回"Just a legend"

"""

```
n = len(s)
if n <= 2:
 return "Just a legend"
```

### # 计算 Z 函数

```
z = z_function(s)
```

### # 记录在中间出现过的前缀长度

```
has_prefix = [False] * (n + 1)
```

### # 标记在中间出现过的前缀长度

```

for i in range(1, n):
 if z[i] > 0:
 has_prefix[z[i]] = True

查找既是前缀又是后缀且在中间出现过的最长子串
max_len = 0
for i in range(1, n):
 # 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
 # 说明这个后缀与原字符串的前缀完全匹配
 if z[i] == n - i and has_prefix[z[i]]:
 max_len = max(max_len, z[i])

如果找到了满足条件的子串，返回它；否则返回"Just a legend"
return s[:max_len] if max_len > 0 else "Just a legend"

```

```

测试方法
if __name__ == "__main__":
 # 示例测试
 print(solve("fixprefixsuffix")) # 输出: fix
 print(solve("abcdabc")) # 输出: Just a legend
 print(solve("abca")) # 输出: ab

```

---

文件: final\_verification.py

---

```

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

```

"""

最终验证脚本

用于验证[class103] (file:///d:/Upan/src/algorithm-journey/src/algorithm-journey/src/class103) 目录下所有算法实现的正确性

包括 Java、Python、C++三种语言的实现

"""

```

import os
import subprocess
import sys
import time

```

# 测试用例

```
TEST_CASES = [
 "abc12321cba",
 "a",
 "aaaaa",
 "abcdefg",
 "babad",
 "cbbd",
 "aacecaaa",
 "abcd",
 "ababbb",
 "zaaaxbbby"
]

def run_java_tests():
 """运行 Java 代码测试"""
 print("开始 Java 代码测试...")

 # 编译 Java 代码
 try:
 subprocess.run(["javac", "Code01_Manacher.java"], check=True, cwd=". ")
 subprocess.run(["javac", "Code02_ExpandKMP.java"], check=True, cwd=". ")
 subprocess.run(["javac", "Codeforces126B_Password.java"], check=True, cwd=". ")
 subprocess.run(["javac", "LeetCode1960_MaxProduct.java"], check=True, cwd=". ")
 print("Java 代码编译成功")
 except subprocess.CalledProcessError:
 print("Java 代码编译失败")
 return False

 # 运行测试
 try:
 # 测试 Manacher 算法
 result = subprocess.run(["java", "Code01_Manacher"],
 input="abc12321cba\n",
 text=True,
 capture_output=True,
 cwd=". ")
 print("Manacher 算法测试结果:", result.stdout.strip())

 # 测试 Z 函数
 result = subprocess.run(["java", "Code02_ExpandKMP"],
 input="abc\nabcd\n",
 text=True,
 capture_output=True,
```

```

 cwd=". ")
print("Z 函数测试结果:", result.stdout.strip())

print("Java 代码测试完成")
return True
except Exception as e:
 print(f"Java 代码运行失败: {e}")
 return False

def run_python_tests():
 """运行 Python 代码测试"""
 print("开始 Python 代码测试...")

 try:
 # 测试 Manacher 算法
 from manacher_python import manacher, longest_palindrome, count_substrings,
shortest_palindrome

 for test_case in TEST_CASES[:3]:
 result = manacher(test_case)
 print(f"manacher(' {test_case}') = {result}")

 # 测试 Z 函数
 from z_function_python import z_function, sum_scores, minimum_time_to_initial_state

 z_result = z_function("abc")
 print(f"z_function(' abc') = {z_result}")

 score = sum_scores("babab")
 print(f"sum_scores(' babab') = {score}")

 time_result = minimum_time_to_initial_state("abacaba", 3)
 print(f"minimum_time_to_initial_state(' abacaba', 3) = {time_result}")

 print("Python 代码测试完成")
 return True
 except Exception as e:
 print(f"Python 代码运行失败: {e}")
 return False

def run_cpp_tests():
 """运行 C++代码测试"""
 print("开始 C++代码测试...")

```

```
编译 C++代码
try:
 subprocess.run(["g++", "-std=c++11", "manacher_cpp.cpp", "-o", "manacher_cpp"],
check=True, cwd=". ")
 subprocess.run(["g++", "-std=c++11", "z_function_cpp.cpp", "-o", "z_function_cpp"],
check=True, cwd=". ")
 print("C++代码编译成功")
except subprocess.CalledProcessError:
 print("C++代码编译失败")
 return False

运行测试
try:
 # 测试 Manacher 算法
 result = subprocess.run("./manacher_cpp",
 input="abc12321cba\n",
 text=True,
 capture_output=True,
 cwd=". ")
 print("Manacher C++测试结果:", result.stdout.strip())

 # 测试 Z 函数
 result = subprocess.run("./z_function_cpp",
 input="abc\nabcd\n",
 text=True,
 capture_output=True,
 cwd=". ")
 print("Z 函数 C++测试结果:", result.stdout.strip())

 print("C++代码测试完成")
 return True
except Exception as e:
 print(f"C++代码运行失败: {e}")
 return False

def main():
 """主函数"""
 print("开始[class103](file:///d:/Upan/src/algorithm-journey/src/algorithm-
journey/src/class103)算法实现验证")
 print("=" * 50)

 # 获取当前目录
```

```
current_dir = os.getcwd()
print(f"当前目录: {current_dir}")

验证文件存在性
required_files = [
 "Code01_Manacher.java",
 "Code02_ExpandKMP.java",
 "Codeforces126B_Password.java",
 "LeetCode1960_MaxProduct.java",
 "manacher_python.py",
 "z_function_python.py",
 "manacher_cpp.cpp",
 "z_function_cpp.cpp"
]

missing_files = []
for file in required_files:
 if not os.path.exists(file):
 missing_files.append(file)

if missing_files:
 print(f"缺少以下文件: {missing_files}")
 return False

print("所有必需文件都存在")

运行各语言测试
tests_passed = 0
total_tests = 3

if run_java_tests():
 tests_passed += 1

if run_python_tests():
 tests_passed += 1

if run_cpp_tests():
 tests_passed += 1

print("=" * 50)
print(f"测试完成: {tests_passed}/{total_tests} 个测试通过")

if tests_passed == total_tests:
```

```
 print("所有测试都通过了！")
 return True
else:
 print("部分测试失败，请检查代码实现")
 return False

if __name__ == "__main__":
 success = main()
 sys.exit(0 if success else 1)
```

---

文件: LeetCode1960\_MaxProduct. java

---

```
package class103;

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

/**
 * LeetCode 1960. 两个回文子字符串长度的最大乘积
 *
 * 题目描述:
 * 给你一个下标从 0 开始的字符串 s，你需要找到两个不重叠的回文子字符串，它们的长度都必须为奇数，使得它们长度的乘积最大。
 *
 * 解题思路:
 * 使用 Manacher 算法计算所有奇回文信息:
 * 1. 使用 Manacher 算法计算每个位置为中心的最长奇回文半径
 * 2. 预处理前缀和后缀数组，分别记录到每个位置为止的最长回文长度
 * 3. 枚举每个分割点，通过前后缀获取左右两个子串中的最长回文大小，相乘即可
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * 题目链接: https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/
 *
 * 相关题目:
 * 1. LeetCode 5. 最长回文子串 - https://leetcode.com/problems/longest-palindromic-substring/
 * 2. LeetCode 647. 回文子串 - https://leetcode.com/problems/palindromic-substrings/
 * 3. LeetCode 336. 回文对 - https://leetcode.com/problems/palindrome-pairs/
 * 4. LeetCode 131. 分割回文串 - https://leetcode.com/problems/palindrome-partitioning/
 * 5. LeetCode 132. 分割回文串 II - https://leetcode.com/problems/palindrome-partitioning-ii/
```

```

* 6. 洛谷 P3805 【模板】manacher - https://www.luogu.com.cn/problem/P3805
* 7. UVa 11475 - Extend to Palindrome -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2470
* 8. Codeforces 1326D2 - Prefix-Suffix Palindrome -
https://codeforces.com/problemset/problem/1326/D2
* 9. HackerRank - Palindromic Substrings
* 10. AcWing 141. 周期 - https://www.acwing.com/problem/content/143/
* 11. POJ 3240 - 回文串
*/
public class LeetCode1960_MaxProduct {

 /**
 * 计算两个不重叠奇回文子字符串长度的最大乘积
 *
 * @param s 输入字符串
 * @return 最大乘积
 */
 public static long maxProduct(String s) {
 int n = s.length();

 // 使用 Manacher 算法计算每个位置为中心的最长奇回文半径
 int[] radius = manacherOdd(s);

 // prefix[i] 表示在 [0, i] 范围内能找到的最长奇回文子串长度
 long[] prefix = new long[n];
 // suffix[i] 表示在 [i, n-1] 范围内能找到的最长奇回文子串长度
 long[] suffix = new long[n];

 // 初始化
 prefix[0] = 1;
 suffix[n - 1] = 1;

 // 计算前缀数组
 for (int i = 1; i < n; i++) {
 // 检查以位置 i 结尾的回文串
 for (int j = 0; j <= i; j++) {
 // 回文串的右边界是 i, 中心是 j, 半径是 radius[j]
 if (j + radius[j] - 1 >= i) {
 prefix[i] = Math.max(prefix[i], 2 * (i - j) + 1);
 }
 }
 prefix[i] = Math.max(prefix[i], prefix[i - 1]);
 }
 }
}

```

```

// 计算后缀数组
for (int i = n - 2; i >= 0; i--) {
 // 检查以位置 i 开头的回文串
 for (int j = i; j < n; j++) {
 // 回文串的左边界是 i, 中心是 j, 半径是 radius[j]
 if (j - radius[j] + 1 <= i) {
 suffix[i] = Math.max(suffix[i], 2 * (j - i) + 1);
 }
 }
 suffix[i] = Math.max(suffix[i], suffix[i + 1]);
}

// 枚举分割点, 计算最大乘积
long maxProduct = 0;
for (int i = 0; i < n - 1; i++) {
 maxProduct = Math.max(maxProduct, prefix[i] * suffix[i + 1]);
}

return maxProduct;
}

/***
 * Manacher 算法计算奇回文串
 *
 * @param s 输入字符串
 * @return 每个位置为中心的最长奇回文半径数组
 */
public static int[] manacherOdd(String s) {
 int n = s.length();
 int[] radius = new int[n];

 for (int i = 0, l = 0, r = -1; i < n; i++) {
 // 利用回文对称性
 int k = (i > r) ? 1 : Math.min(radius[l + r - i], r - i + 1);

 // 尝试扩展回文串
 while (0 <= i - k && i + k < n && s.charAt(i - k) == s.charAt(i + k)) {
 k++;
 }

 radius[i] = k--;
 }
}

```

```

 // 更新最右回文边界
 if (i + k > r) {
 l = i - k;
 r = i + k;
 }
}

return radius;
}

// 测试方法
public static void main(String[] args) {
 // 示例测试
 System.out.println(maxProduct("ababbb")); // 输出: 9
 System.out.println(maxProduct("zaaaxbbby")); // 输出: 9
}
}

```

=====

文件: leetcode\_1960\_max\_product.cpp

=====

```

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

/**
 * Manacher 算法计算奇回文串
 *
 * @param s 输入字符串
 * @return 每个位置为中心的最长奇回文半径数组
 */
vector<int> manacherOdd(string s) {
 int n = s.length();
 vector<int> radius(n, 0);

 int l = 0, r = -1;
 for (int i = 0; i < n; i++) {
 // 利用回文对称性
 int k = (i > r) ? 1 : min(radius[l + r - i], r - i + 1);

```

```

// 尝试扩展回文串
while (0 <= i - k && i + k < n && s[i - k] == s[i + k]) {
 k++;
}

radius[i] = k - 1;

// 更新最右回文边界
if (i + radius[i] > r) {
 l = i - radius[i];
 r = i + radius[i];
}
}

return radius;
}

/***
 * LeetCode 1960. 两个回文子字符串长度的最大乘积
 *
 * 题目描述:
 * 给你一个下标从 0 开始的字符串 s，你需要找到两个不重叠的回文子字符串，
 * 它们的长度都必须为奇数，使得它们长度的乘积最大。
 *
 * 解题思路:
 * 使用 Manacher 算法计算所有奇回文信息:
 * 1. 使用 Manacher 算法计算每个位置为中心的最长奇回文半径
 * 2. 预处理前缀和后缀数组，分别记录到每个位置为止的最长回文长度
 * 3. 枚举每个分割点，通过前后缀获取左右两个子串中的最长回文大小，相乘即可
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 *
 * @param s 输入字符串
 * @return 最大乘积
 */
long long maxProduct(string s) {
 int n = s.length();

 // 使用 Manacher 算法计算每个位置为中心的最长奇回文半径
 vector<int> radius = manacherOdd(s);

 // prefix[i] 表示在 [0, i] 范围内能找到的最长奇回文子串长度

```

```

vector<long long> prefix(n, 0);
// suffix[i] 表示在 [i, n-1] 范围内能找到的最长奇回文子串长度
vector<long long> suffix(n, 0);

// 初始化
prefix[0] = 1;
suffix[n - 1] = 1;

// 计算前缀数组
for (int i = 1; i < n; i++) {
 // 检查以位置 i 结尾的回文串
 for (int j = 0; j <= i; j++) {
 // 回文串的右边界是 i, 中心是 j, 半径是 radius[j]
 if (j + radius[j] >= i) {
 prefix[i] = max(prefix[i], (long long)(2 * (i - j) + 1));
 }
 }
 prefix[i] = max(prefix[i], prefix[i - 1]);
}

// 计算后缀数组
for (int i = n - 2; i >= 0; i--) {
 // 检查以位置 i 开头的回文串
 for (int j = i; j < n; j++) {
 // 回文串的左边界是 i, 中心是 j, 半径是 radius[j]
 if (j - radius[j] <= i) {
 suffix[i] = max(suffix[i], (long long)(2 * (j - i) + 1));
 }
 }
 suffix[i] = max(suffix[i], suffix[i + 1]);
}

// 枚举分割点, 计算最大乘积
long long maxProduct = 0;
for (int i = 0; i < n - 1; i++) {
 maxProduct = max(maxProduct, prefix[i] * suffix[i + 1]);
}

return maxProduct;
}

// 测试方法
int main() {

```

```
// 示例测试
cout << maxProduct("ababbb") << endl; // 输出: 9
cout << maxProduct("zaaaxbbby") << endl; // 输出: 9

return 0;
}
```

=====

文件: leetcode\_1960\_max\_product.py

=====

```
def manacher_odd(s):
 """
 Manacher 算法计算奇回文串
```

Args:

s: 输入字符串

Returns:

每个位置为中心的最长奇回文半径数组

"""

```
n = len(s)
radius = [0] * n
```

```
l, r = 0, -1
for i in range(n):
 # 利用回文对称性
 k = l if i > r else min(radius[l + r - i], r - i + 1)
```

# 尝试扩展回文串

```
while 0 <= i - k and i + k < n and s[i - k] == s[i + k]:
 k += 1
```

```
radius[i] = k - 1
```

# 更新最右回文边界

```
if i + radius[i] > r:
 l = i - radius[i]
 r = i + radius[i]
```

```
return radius
```

```
def max_product(s):
"""
LeetCode 1960. 两个回文子字符串长度的最大乘积
```

题目描述：

给你一个下标从 0 开始的字符串 s，你需要找到两个不重叠的回文子字符串，它们的长度都必须为奇数，使得它们长度的乘积最大。

解题思路：

使用 Manacher 算法计算所有奇回文信息：

1. 使用 Manacher 算法计算每个位置为中心的最长奇回文半径
2. 预处理前缀和后缀数组，分别记录到每个位置为止的最长回文长度
3. 枚举每个分割点，通过前后缀获取左右两个子串中的最长回文大小，相乘即可

时间复杂度：O(n)

空间复杂度：O(n)

Args:

s: 输入字符串

Returns:

最大乘积

"""

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

```
使用 Manacher 算法计算每个位置为中心的最长奇回文半径
```

```
radius = manacher_odd(s)
```

```
prefix[i] 表示在 [0, i] 范围内能找到的最长奇回文子串长度
```

```
prefix = [0] * n
```

```
suffix[i] 表示在 [i, n-1] 范围内能找到的最长奇回文子串长度
```

```
suffix = [0] * n
```

```
初始化
```

```
prefix[0] = 1
```

```
suffix[n - 1] = 1
```

```
计算前缀数组
```

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

```
检查以位置 i 结尾的回文串
```

```
for j in range(i + 1):
```

```
回文串的右边界是 i, 中心是 j, 半径是 radius[j]
```

```
if j + radius[j] >= i:
```

```

 prefix[i] = max(prefix[i], 2 * (i - j) + 1)
prefix[i] = max(prefix[i], prefix[i - 1])

计算后缀数组
for i in range(n - 2, -1, -1):
 # 检查以位置 i 开头的回文串
 for j in range(i, n):
 # 回文串的左边界是 i, 中心是 j, 半径是 radius[j]
 if j - radius[j] <= i:
 suffix[i] = max(suffix[i], 2 * (j - i) + 1)
suffix[i] = max(suffix[i], suffix[i + 1])

枚举分割点, 计算最大乘积
max_prod = 0
for i in range(n - 1):
 max_prod = max(max_prod, prefix[i] * suffix[i + 1])

return max_prod

```

```

测试方法
if __name__ == "__main__":
 # 示例测试
 print(max_product("ababbb")) # 输出: 9
 print(max_product("zaaaxbbby")) # 输出: 9
=====

文件: manacher_cpp.cpp
=====

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

const int MAXN = 11000001;
char ss[MAXN << 1];
int p[MAXN << 1];
int n;

/***
 * Manacher 算法主函数, 用于计算字符串中最长回文子串的长度

```

```

*
* 算法原理:
* 1. 预处理: 在原字符串的每个字符之间插入特殊字符'#'，并在首尾也添加'#'
* 这样可以将奇数长度和偶数长度的回文串统一处理为奇数长度的回文串
* 2. 利用回文串的对称性，避免重复计算
* 3. 维护当前最右回文边界 r 和对应的中心 c，通过已计算的信息加速新位置的计算
*
* 时间复杂度: O(n)，其中 n 为字符串长度
* 空间复杂度: O(n)
*
* @param str 输入字符串
* @return 最长回文子串的长度
*
* 应用场景:
* 1. LeetCode 5. 最长回文子串 - https://leetcode.com/problems/longest-palindromic-substring/
* 2. LeetCode 647. 回文子串 - https://leetcode.com/problems/palindromic-substrings/
* 3. LeetCode 214. 最短回文串 - https://leetcode.com/problems/shortest-palindrome/
* 4. 洛谷 P3805 【模板】manacher - https://www.luogu.com.cn/problem/P3805
* 5. UVa 11475 - Extend to Palindrome -
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2470
* 6. Codeforces 1326D2 - Prefix-Suffix Palindrome -
https://codeforces.com/problemset/problem/1326/D2
* 7. HackerRank - Palindromic Substrings
* 8. AcWing 141. 周期 - https://www.acwing.com/problem/content/143/
* 9. POJ 3240 - 回文串
* 10. LeetCode 336. 回文对 - https://leetcode.com/problems/palindrome-pairs/
* 11. LeetCode 131. 分割回文串 - https://leetcode.com/problems/palindrome-partitioning/
* 12. LeetCode 132. 分割回文串 II - https://leetcode.com/problems/palindrome-partitioning-ii/
*/
int manacher(string str) {
 // 预处理字符串
 n = str.length() * 2 + 1;
 for (int i = 0, j = 0; i < n; i++) {
 ss[i] = (i & 1) == 0 ? '#' : str[j++];
 }
}

int maxLen = 0;
// c: 当前最右回文子串的中心
// r: 当前最右回文子串的右边界
for (int i = 0, c = 0, r = 0; i < n; i++) {
 // 利用回文对称性优化
 int len = r > i ? min(p[2 * c - i], r - i) : 1;
 ...
}

```

```

// 尝试扩展回文串
while (i + len < n && i - len >= 0 && ss[i + len] == ss[i - len]) {
 len++;
}

// 更新最右回文边界和中心
if (i + len > r) {
 r = i + len;
 c = i;
}

// 更新最大回文半径
maxLen = max(maxLen, len);
p[i] = len;
}

```

// 由于我们插入了'#'字符，实际回文长度是半径减1

```

return maxLen - 1;
}

```

```

/**
 * LeetCode 5. 最长回文子串
 * 给你一个字符串 s，找到 s 中最长的回文子串
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */

```

```

string longestPalindrome(string s) {
 if (s.empty()) return "";

 // 预处理字符串
 string processed = "^";
 for (int i = 0; i < s.length(); i++) {
 processed += "#" + string(1, s[i]);
 }
 processed += "#$";

 int len = processed.length();
 vector<int> radius(len, 0);

 int center = 0, right = 0;
 int maxLen = 0, centerIndex = 0;

```

```

for (int i = 1; i < len - 1; i++) {
 // 利用回文对称性
 int mirror = 2 * center - i;

 if (i < right) {
 radius[i] = min(right - i, radius[mirror]);
 }

 // 尝试扩展回文
 while (processed[i + radius[i] + 1] == processed[i - radius[i] - 1]) {
 radius[i]++;
 }

 // 更新最右边界
 if (i + radius[i] > right) {
 center = i;
 right = i + radius[i];
 }

 // 更新最长回文
 if (radius[i] > maxLen) {
 maxLen = radius[i];
 centerIndex = i;
 }
}

// 从处理后的字符串中提取原始回文子串
int start = (centerIndex - maxLen) / 2;
return s.substr(start, maxLen);
}

/***
 * LeetCode 647. 回文子串
 * 给定一个字符串，计算其中回文子串的数目
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
int countSubstrings(string s) {
 if (s.empty()) return 0;

 // 预处理字符串
 string processed = "^";

```

```

for (int i = 0; i < s.length(); i++) {
 processed += "#" + string(1, s[i]);
}
processed += "#$";

int len = processed.length();
vector<int> radius(len, 0);

int center = 0, right = 0;
int count = 0;

for (int i = 1; i < len - 1; i++) {
 // 利用回文对称性
 int mirror = 2 * center - i;

 if (i < right) {
 radius[i] = min(right - i, radius[mirror]);
 }

 // 尝试扩展回文
 while (processed[i + radius[i] + 1] == processed[i - radius[i] - 1]) {
 radius[i]++;
 }

 // 更新最右边界
 if (i + radius[i] > right) {
 center = i;
 right = i + radius[i];
 }

 // 每个回文半径可以贡献 radius[i]/2 个回文子串
 count += (radius[i] + 1) / 2;
}

return count;
}

/**
 * LeetCode 214. 最短回文串
 * 给定一个字符串 s，可以通过在字符串前面添加字符将其转换为回文串。
 * 找到并返回可以用这种方式转换的最短回文串。
 *
 * 时间复杂度：O(n)

```

```

* 空间复杂度: O(n)
*/
string shortestPalindrome(string s) {
 if (s.length() <= 1) return s;

 // 将字符串与其反转拼接，中间用特殊字符分隔
 string combined = s + "#" + string(s.rbegin(), s.rend());

 // 计算 KMP 的 LPS 数组
 vector<int> lps(combined.length(), 0);
 for (int i = 1, j = 0; i < combined.length(); i++) {
 if (combined[i] == combined[j]) {
 lps[i] = ++j;
 } else {
 if (j != 0) {
 j = lps[j - 1];
 }
 i--;
 }
 }

 // 找到原字符串的最长回文前缀
 int overlap = lps[combined.length() - 1];

 // 在原字符串前添加反转的部分
 string prefix = string(s.begin() + overlap, s.end());
 reverse(prefix.begin(), prefix.end());
 return prefix + s;
}

/***
 * 洛谷 P3805 【模板】manacher
 * 题目描述: 给出一个只由小写英文字母 a, b, c... y, z 组成的字符串 S ,
 * 求 S 中最长回文串的长度 。
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(n)
 */
int longestPalindromeLength(string s) {
 return manacher(s);
}

int main() {

```

```
// 测试洛谷 P3805 Manacher 模板题
string input;
getline(cin, input);
cout << manacher(input) << endl;

return 0;
}
```

---

文件: manacher\_python.py

```
def manacher(s):
 """
 Manacher 算法主函数，用于计算字符串中最长回文子串的长度

```

算法原理:

1. 预处理: 在原字符串的每个字符之间插入特殊字符'#'，并在首尾也添加'#'  
这样可以将奇数长度和偶数长度的回文串统一处理为奇数长度的回文串
2. 利用回文串的对称性，避免重复计算
3. 维护当前最右回文边界 r 和对应的中心 c，通过已计算的信息加速新位置的计算

时间复杂度:  $O(n)$ ，其中 n 为字符串长度

空间复杂度:  $O(n)$

Args:

s: 输入字符串

Returns:

最长回文子串的长度

应用场景:

1. LeetCode 5. 最长回文子串 - <https://leetcode.com/problems/longest-palindromic-substring/>
2. LeetCode 647. 回文子串 - <https://leetcode.com/problems/palindromic-substrings/>
3. LeetCode 214. 最短回文串 - <https://leetcode.com/problems/shortest-palindrome/>
4. 洛谷 P3805 【模板】manacher - <https://www.luogu.com.cn/problem/P3805>
5. UVa 11475 - Extend to Palindrome -

[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&page=show\\_problem&problem=2470](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2470)

6. Codeforces 1326D2 - Prefix-Suffix Palindrome -

<https://codeforces.com/problemset/problem/1326/D2>

7. HackerRank - Palindromic Substrings
8. AcWing 141. 周期 - <https://www.acwing.com/problem/content/143/>
9. POJ 3240 - 回文串

10. LeetCode 336. 回文对 - <https://leetcode.com/problems/palindrome-pairs/>
  11. LeetCode 131. 分割回文串 - <https://leetcode.com/problems/palindrome-partitioning/>
  12. LeetCode 132. 分割回文串 II - <https://leetcode.com/problems/palindrome-partitioning-ii/>
- """

```
预处理字符串
```

```
processed = '#' + join(' ^{} $'.format(s))
```

```
n = len(processed)
```

```
p = [0] * n
```

```
max_len = 0
```

```
c: 当前最右回文子串的中心
```

```
r: 当前最右回文子串的右边界
```

```
c = r = 0
```

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

```
 # 利用回文对称性优化
```

```
 # 如果 i 在当前右边界内，则可以利用对称点 2*c-i 的信息
```

```
 if i < r:
```

```
 p[i] = min(r - i, p[2 * c - i])
```

```
尝试扩展回文串
```

```
从当前半径开始，尝试向两边扩展
```

```
try:
```

```
 while processed[i + p[i] + 1] == processed[i - p[i] - 1]:
```

```
 p[i] += 1
```

```
except IndexError:
```

```
 # 边界情况处理
```

```
 pass
```

```
更新最右回文边界和中心
```

```
if i + p[i] > r:
```

```
 c, r = i, i + p[i]
```

```
更新最大回文半径
```

```
max_len = max(max_len, p[i])
```

```
由于我们插入了 '#' 字符，实际回文长度就是半径
```

```
return max_len
```

```
def longest_palindrome(s):
```

```
"""
```

```
LeetCode 5. 最长回文子串
```

给你一个字符串 s，找到 s 中最长的回文子串

时间复杂度: O(n)

空间复杂度: O(n)

Args:

s: 输入字符串

Returns:

最长回文子串

"""

if not s:

    return ""

# 预处理字符串

processed = '#'.join('^{}\$'.format(s))

n = len(processed)

p = [0] \* n

c = r = 0

max\_len = 0

center\_index = 0

for i in range(1, n - 1):

# 利用回文对称性

if i < r:

    p[i] = min(r - i, p[2 \* c - i])

# 尝试扩展回文

try:

    while processed[i + p[i] + 1] == processed[i - p[i] - 1]:

        p[i] += 1

except IndexError:

    pass

# 更新最右边界

if i + p[i] > r:

    c, r = i, i + p[i]

# 更新最长回文

if p[i] > max\_len:

    max\_len = p[i]

    center\_index = i

```
从处理后的字符串中提取原始回文子串
start = (center_index - max_len) // 2
return s[start:start + max_len]
```

```
def count_substrings(s):
 """
 LeetCode 647. 回文子串
 给定一个字符串，计算其中回文子串的数目
```

时间复杂度: O(n)

空间复杂度: O(n)

Args:

s: 输入字符串

Returns:

回文子串的数目

"""
if not s:
 return 0

# 预处理字符串

processed = '#'.join('^{}\$'.format(s))

n = len(processed)

p = [0] \* n

c = r = 0

count = 0

for i in range(1, n - 1):

# 利用回文对称性

if i < r:

p[i] = min(r - i, p[2 \* c - i])

# 尝试扩展回文

try:

while processed[i + p[i] + 1] == processed[i - p[i] - 1]:

p[i] += 1

except IndexError:

pass

```
更新最右边界
if i + p[i] > r:
 c, r = i, i + p[i]

每个回文半径可以贡献 (p[i]+1)//2 个回文子串
count += (p[i] + 1) // 2

return count
```

```
def shortest_palindrome(s):
```

```
"""
```

```
LeetCode 214. 最短回文串
```

```
给定一个字符串 s，可以通过在字符串前面添加字符将其转换为回文串。
找到并返回可以用这种方式转换的最短回文串。
```

时间复杂度: O(n)

空间复杂度: O(n)

Args:

s: 输入字符串

Returns:

最短回文串

```
"""
```

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

```
将字符串与其反转拼接，中间用特殊字符分隔
```

```
combined = s + "#" + s[::-1]
```

```
计算 KMP 的 LPS 数组
```

```
lps = [0] * len(combined)
for i in range(1, len(combined)):
 j = lps[i - 1]
 while j > 0 and combined[i] != combined[j]:
 j = lps[j - 1]
 if combined[i] == combined[j]:
 j += 1
 lps[i] = j
```

```
找到原字符串的最长回文前缀
```

```
overlap = lps[-1]
```

```
在原字符串前添加反转的部分
return s[overlap:][::-1] + s
```

```
def longest_palindrome_length(s):
```

```
 """

```

洛谷 P3805 【模板】manacher

题目描述：给出一个只由小写英文字母 a, b, c... y, z 组成的字符串 S，  
求 S 中最长回文串的长度。

时间复杂度：O(n)

空间复杂度：O(n)

Args:

s: 输入字符串

Returns:

最长回文子串的长度

```
 """

```

```
return manacher(s)
```

```
测试洛谷 P3805 Manacher 模板题
```

```
if __name__ == "__main__":
 input_str = input().strip()
 print(manacher(input_str))
```

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

文件：test\_all.py

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

```

```
"""

```

测试所有算法实现的脚本  
包括单元测试和集成测试

```
"""

```

```
import unittest
import sys
import os
```

```
添加当前目录到 Python 路径
sys.path.append('.')

class TestManacherAlgorithms(unittest.TestCase):
 """Manacher 算法测试类"""

 def setUp(self):
 """测试前准备"""
 # 导入 Manacher 算法实现
 try:
 from manacher_python import manacher, longest_palindrome, count_substrings,
shortest_palindrome
 self.manacher = manacher
 self.longest_palindrome = longest_palindrome
 self.count_substrings = count_substrings
 self.shortest_palindrome = shortest_palindrome
 except ImportError:
 self.skipTest("Python Manacher 实现未找到")

 def test_manacher_basic(self):
 """测试基本 Manacher 算法"""
 self.assertEqual(self.manacher("abc12321cba"), 7)
 self.assertEqual(self.manacher("a"), 1)
 self.assertEqual(self.manacher("aaaaa"), 5)
 self.assertEqual(self.manacher(""), 0)

 def test_longest_palindrome(self):
 """测试最长回文子串"""
 result = self.longest_palindrome("babad")
 self.assertIn(result, ["bab", "aba"])

 result = self.longest_palindrome("cbbd")
 self.assertEqual(result, "bb")

 result = self.longest_palindrome("a")
 self.assertEqual(result, "a")

 result = self.longest_palindrome("")
 self.assertEqual(result, "")

 def test_count_substrings(self):
 """测试回文子串计数"""
```

```
self.assertEqual(self.count_substrings("abc"), 3)
self.assertEqual(self.count_substrings("aaa"), 6)
self.assertEqual(self.count_substrings("a"), 1)
self.assertEqual(self.count_substrings(""), 0)

def test_shortest_palindrome(self):
 """测试最短回文串"""
 self.assertEqual(self.shortest_palindrome("aacecaaa"), "aaacecaaa")
 self.assertEqual(self.shortest_palindrome("abcd"), "dcbabcd")
 self.assertEqual(self.shortest_palindrome("a"), "a")
 self.assertEqual(self.shortest_palindrome(""), "")

class TestZFunctionAlgorithms(unittest.TestCase):
 """Z 函数算法测试类"""

 def setUp(self):
 """测试前准备"""
 # 导入 Z 函数实现
 try:
 from z_function_python import z_function, sum_scores, minimum_time_to_initial_state
 self.z_function = z_function
 self.sum_scores = sum_scores
 self.minimum_time_to_initial_state = minimum_time_to_initial_state
 except ImportError:
 self.skipTest("Python Z 函数实现未找到")

 def test_z_function_basic(self):
 """测试基本 Z 函数"""
 result = self.z_function("aaaaa")
 expected = [5, 4, 3, 2, 1]
 self.assertEqual(result, expected)

 result = self.z_function("ababc")
 expected = [5, 0, 2, 0, 1]
 self.assertEqual(result, expected)

 def test_sum_scores(self):
 """测试构造字符串的总得分和"""
 self.assertEqual(self.sum_scores("babab"), 9)
 self.assertEqual(self.sum_scores("azbazbzaz"), 14)

 def test_minimum_time_to_initial_state(self):
 """测试恢复初始状态所需的最短时间"""
```

```
 self.assertEqual(self.minimum_time_to_initial_state("abacaba", 3), 2)
 self.assertEqual(self.minimum_time_to_initial_state("abacaba", 4), 1)
 self.assertEqual(self.minimum_time_to_initial_state("abcdef", 2), 3)

class TestCodeforcesProblems(unittest.TestCase):
 """Codeforces 题目测试类"""

 def setUp(self):
 """测试前准备"""
 # 导入 Codeforces 问题实现
 try:
 from z_function_python import z_function
 self.z_function = z_function
 except ImportError:
 self.skipTest("Python Z 函数实现未找到")

 def test_password_problem(self):
 """测试 Codeforces 126B Password 问题"""
 # 这里我们测试 Z 函数的正确性，因为 Password 问题的完整实现可能在 Java 中
 result = self.z_function("abcabc")
 expected = [6, 0, 0, 3, 0, 0]
 self.assertEqual(result, expected)

class TestLeetCodeProblems(unittest.TestCase):
 """LeetCode 题目测试类"""

 def setUp(self):
 """测试前准备"""
 # 导入 LeetCode 问题实现
 try:
 from manacher_python import manacher
 from z_function_python import sum_scores, minimum_time_to_initial_state
 self.manacher = manacher
 self.sum_scores = sum_scores
 self.minimum_time_to_initial_state = minimum_time_to_initial_state
 except ImportError:
 self.skipTest("Python 算法实现未找到")

 def test_leetcode_5(self):
 """测试 LeetCode 5. 最长回文子串"""
 # 这个测试已经在 TestManacherAlgorithms 中完成

 def test_leetcode_2223(self):
```

```

"""测试 LeetCode 2223. 构造字符串的总得分和"""
self.assertEqual(self.sum_scores("babab"), 9)

def test_leetcode_3031(self):
 """测试 LeetCode 3031. 将单词恢复初始状态所需的最短时间 II"""
 self.assertEqual(self.minimum_time_to_initial_state("abacaba", 3), 2)

def run_all_tests():
 """运行所有测试"""
 # 创建测试套件
 test_suite = unittest.TestSuite()

 # 添加所有测试类
 test_suite.addTest(unittest.makeSuite(TestManacherAlgorithms))
 test_suite.addTest(unittest.makeSuite(TestZFunctionAlgorithms))
 test_suite.addTest(unittest.makeSuite(TestCodeforcesProblems))
 test_suite.addTest(unittest.makeSuite(TestLeetCodeProblems))

 # 运行测试
 runner = unittest.TextTestRunner(verbosity=2)
 result = runner.run(test_suite)

 return result.wasSuccessful()

if __name__ == "__main__":
 print("开始运行所有算法测试...")
 print("=" * 50)

 success = run_all_tests()

 print("=" * 50)
 if success:
 print("所有测试通过!")
 sys.exit(0)
 else:
 print("部分测试失败!")
 sys.exit(1)

```

=====

文件: z\_function\_cpp.cpp

=====

```
#include <iostream>
```

```
#include <string>
#include <vector>
#include <algorithm>
#include <utility>
using namespace std;

const int MAXN = 20000001;
int z[MAXN];
int e[MAXN];

/***
 * Z 函数计算
 * Z 函数 $z[i]$ 表示字符串 s 从位置 i 开始与字符串 s 从位置 0 开始的最长公共前缀长度
 *
 * 算法原理：
 * 1. 维护一个匹配区间 $[l, r]$, 表示当前已知的最右匹配区间
 * 2. 对于当前位置 i , 如果 $i \leq r$, 可以利用已计算的信息优化
 * 3. 利用对称性, $z[i]$ 至少为 $\min(r - i + 1, z[i - 1])$
 * 4. 在此基础之上继续向右扩展匹配
 *
 * 时间复杂度: $O(n)$
 * 空间复杂度: $O(n)$
 *
 * 应用场景:
 * 1. LeetCode 2223. 构造字符串的总得分和 - https://leetcode.com/problems/sum-of-scores-of-built-strings/
 * 2. LeetCode 3031. 将单词恢复初始状态所需的最短时间 II - https://leetcode.com/problems/minimum-time-to-revert-word-to-initial-state-ii/
 * 3. Codeforces 126B. Password - https://codeforces.com/problemset/problem/126/B
 * 4. 洛谷 P5410 【模板】扩展 KMP/exKMP (Z 函数) - https://www.luogu.com.cn/problem/P5410
 * 5. SPOJ - Pattern Find
 * 6. HackerEarth - String Similarity - https://www.hackerearth.com/practice/algorithms/string-algorithm/z-algorithm/tutorial/
 * 7. AtCoder ABC141E - Who Says a Pun? - https://atcoder.jp/contests/abc141/tasks/abc141_e
 * 8. USACO 2011 November Contest, Bronze - Cow Photographs
 * 9. 牛客网 NC15051 - 字符串的匹配
 */
void zArray(string s, int n) {
 z[0] = n;
 // l: 当前最右匹配区间的左边界
 // r: 当前最右匹配区间的右边界
 for (int i = 1, l = 0, r = 0; i < n; i++) {
 // 利用已计算的信息优化
 }
}
```

```

// 如果 i 在当前匹配区间内
int len = (r > i) ? min(r - i + 1, z[i - 1]) : 0;
// 继续向右扩展匹配
while (i + len < n && s[i + len] == s[len]) {
 len++;
}
// 更新最右匹配区间
if (i + len > r) {
 r = i + len;
 l = i;
}
z[i] = len;
}

/**
 * 扩展 KMP 计算
 * 计算字符串 a 的每个后缀与字符串 b 的最长公共前缀长度
 *
 * 时间复杂度: O(n + m)，其中 n 是 a 的长度，m 是 b 的长度
 * 空间复杂度: O(n + m)
 *
 * @param a 主字符串
 * @param b 模式字符串
 * @param n 主字符串长度
 * @param m 模式字符串长度
 */
void eArray(string a, string b, int n, int m) {
 for (int i = 0, l = 0, r = 0; i < n; i++) {
 // 利用已计算的信息优化
 int len = (r > i) ? min(r - i + 1, z[i - 1]) : 0;
 // 继续向右扩展匹配
 while (i + len < n && len < m && a[i + len] == b[len]) {
 len++;
 }
 // 更新最右匹配区间
 if (i + len > r) {
 r = i + len;
 l = i;
 }
 e[i] = len;
 }
}

```

```

/***
 * 计算数组的权值异或和
 * 对于数组中的每个元素 arr[i]，权值为 (i+1) * (arr[i] + 1)
 *
 * @param arr 输入数组
 * @param n 数组长度
 * @return 所有权值的异或和
 */
long long eor(int arr[], int n) {
 long long ans = 0;
 for (int i = 0; i < n; i++) {
 ans ^= (long long)(i + 1) * (arr[i] + 1);
 }
 return ans;
}

/***
 * LeetCode 2223. 构造字符串的总得分和
 * 你需要从空字符串开始构造一个长度为 n 的字符串 s，构造过程为每次给当前字符串前面添加一个字符。
 * 构造过程中得到的所有字符串编号为 1 到 n，其中长度为 i 的字符串编号为 si。
 * si 的得分为 si 和 sn 的最长公共前缀的长度（注意 s == sn）。
 * 请你返回每一个 si 的得分之和。
 *
 * 示例：
 * 输入：s = "babab"
 * 输出：9
 * 解释：
 * s1 == "b"，得分 1
 * s2 == "ab"，得分 0
 * s3 == "bab"，得分 3
 * s4 == "abab"，得分 0
 * s5 == "babab"，得分 5
 * 总和为 1+0+3+0+5=9
 *
 * 时间复杂度：O(n)
 * 空间复杂度：O(n)
 *
 * @param s 输入字符串
 * @return 得分总和
 */
long long sumScores(string s) {
 int n = s.length();

```

```

// 计算 Z 函数
vector<int> z(n, 0);
z[0] = n;

long long sum = n; // sn 的得分就是整个字符串的长度

for (int i = 1, l = 0, r = 0; i < n; i++) {
 // 利用之前计算的结果
 if (i <= r) {
 z[i] = min(r - i + 1, z[i - 1]);
 }

 // 扩展匹配
 while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
 z[i]++;
 }

 // 更新匹配区间
 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
}

// 累加得分
sum += z[i];
}

return sum;
}

/***
 * LeetCode 3031. 将单词恢复初始状态所需的最短时间 II
 * 给你一个下标从 0 开始的字符串 word 和一个整数 k。
 * 每一秒执行以下操作：
 * 1. 移除 word 的前 k 个字符
 * 2. 在 word 的末尾添加 k 个任意字符
 * 返回将 word 恢复到初始状态所需的最短时间（该时间必须大于零）。
 *
 * 示例：
 * 输入：word = "abacaba", k = 3
 * 输出：2
 * 解释：
 */

```

```

* 第 1 秒后, word 变成"acaba**" (用*表示添加的字符)
* 第 2 秒后, word 变成"aba****"
* 如果添加的字符分别为"cac"和"caba", word 就恢复为"abacaba"
*
* 时间复杂度: O(n)
* 空间复杂度: O(n)
*
* @param word 输入字符串
* @param k 每次操作移除和添加的字符数
* @return 恢复初始状态所需的最短时间
*/
int minimumTimeToInitialState(string word, int k) {
 int n = word.length();

 // 计算 Z 函数
 vector<int> z(n, 0);
 z[0] = n;

 for (int i = 1, l = 0, r = 0; i < n; i++) {
 if (i <= r) {
 z[i] = min(r - i + 1, z[i - 1]);
 }

 while (i + z[i] < n && word[z[i]] == word[i + z[i]]) {
 z[i]++;
 }

 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
 }

 // 查找满足条件的最长时间
 for (int i = k; i < n; i += k) {
 // 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
 // 说明在第(i/k)步后可以恢复原字符串
 if (z[i] >= n - i) {
 return i / k;
 }
 }

 // 最坏情况需要完全替换
}

```

```

 return (n + k - 1) / k;
}

/***
 * 洛谷 P5410 【模板】扩展 KMP (Z 函数)
 * 题目描述: 给定两个字符串 a, b, 求:
 * 1. b 与 b 每一个后缀串的最长公共前缀长度 (即 b 的 Z 函数)
 * 2. a 与 b 每一个后缀串的最长公共前缀长度 (即扩展 KMP)
 *
 * 时间复杂度: O(n + m)
 * 空间复杂度: O(n + m)
 *
 * @param a 主字符串
 * @param b 模式字符串
 * @return pair<Z 数组异或和, E 数组异或和>
*/
pair<long long, long long> extendedKMP(string a, string b) {
 int n = a.length();
 int m = b.length();

 // 计算 b 的 Z 函数
 vector<int> z(m, 0);
 z[0] = m;

 for (int i = 1, l = 0, r = 0; i < m; i++) {
 if (i <= r) {
 z[i] = min(r - i + 1, z[i - 1]);
 }
 }

 while (i + z[i] < m && b[z[i]] == b[i + z[i]]) {
 z[i]++;
 }

 if (i + z[i] - 1 > r) {
 l = i;
 r = i + z[i] - 1;
 }
}

// 计算 a 与 b 的扩展 KMP
vector<int> e(n, 0);
for (int i = 0, l = 0, r = 0; i < n; i++) {
 int len = (r > i) ? min(r - i + 1, z[i - 1]) : 0;
}

```

```

while (i + len < n && len < m && a[i + len] == b[len]) {
 len++;
}

if (i + len > r) {
 r = i + len;
 l = i;
}
e[i] = len;
}

// 计算异或和
long long zXor = 0;
for (int i = 0; i < m; i++) {
 zXor ^= (long long)(i + 1) * (z[i] + 1);
}

long long eXor = 0;
for (int i = 0; i < n; i++) {
 eXor ^= (long long)(i + 1) * (e[i] + 1);
}

return make_pair(zXor, eXor);
}

int main() {
// 测试洛谷 P5410 扩展 KMP 模板题
string a, b;
getline(cin, a);
getline(cin, b);

int n = a.length();
int m = b.length();

// 计算 b 的 Z 函数
zArray(b, m);
// 计算 a 与 b 的扩展 KMP
eArray(a, b, n, m);

cout << eor(z, m) << endl;
cout << eor(e, n) << endl;
}

```

```
 return 0;
}
```

=====

文件: z\_function\_python.py

=====

```
def z_function(s):
 """
 Z 函数计算
 Z 函数 $z[i]$ 表示字符串 s 从位置 i 开始与字符串 s 从位置 0 开始的最长公共前缀长度
```

算法原理:

1. 维护一个匹配区间  $[l, r]$ , 表示当前已知的最右匹配区间
2. 对于当前位置  $i$ , 如果  $i \leq r$ , 可以利用已计算的信息优化
3. 利用对称性,  $z[i]$  至少为  $\min(r - i + 1, z[i - 1])$
4. 在此基础之上继续向右扩展匹配

时间复杂度:  $O(n)$

空间复杂度:  $O(n)$

Args:

s: 输入字符串

Returns:

Z 函数数组

"""

```
n = len(s)
z = [0] * n
z[0] = n

l: 当前最右匹配区间的左边界
r: 当前最右匹配区间的右边界
l = r = 0
```

```
for i in range(1, n):
 # 利用已计算的信息优化
 # 如果 i 在当前匹配区间内
 if i <= r:
 z[i] = min(r - i + 1, z[i - 1])

 # 继续向右扩展匹配
 while i + z[i] < n and s[z[i]] == s[i + z[i]]:
```

```

z[i] += 1

更新最右匹配区间
if i + z[i] - 1 > r:
 l = i
 r = i + z[i] - 1

return z

def extended_kmp(a, b):
 """
 扩展 KMP 计算
 计算字符串 a 的每个后缀与字符串 b 的最长公共前缀长度
 """

 time complexity: O(n + m), 其中 n 是 a 的长度, m 是 b 的长度
 space complexity: O(n + m)

```

Args:

- a: 主字符串
- b: 模式字符串

Returns:

E 数组, 其中 e[i] 表示 a[i:] 与 b 的最长公共前缀长度

"""

n, m = len(a), len(b)

# 先计算 b 的 Z 函数

z = z\_function(b)

# 计算扩展 KMP

e = [0] \* n

l = r = 0

for i in range(n):

# 利用已计算的信息优化

if i <= r:

e[i] = min(r - i + 1, z[i - 1])

# 继续向右扩展匹配

while i + e[i] < n and e[i] < m and a[i + e[i]] == b[e[i]]:

e[i] += 1

```

更新最右匹配区间
if i + e[i] - 1 > r:
 l = i
 r = i + e[i] - 1

return e

def xor_sum(arr):
 """计算数组的权值: xor(i * (arr[i] + 1))"""
 result = 0
 for i in range(len(arr)):
 result ^= (i + 1) * (arr[i] + 1)
 return result

```

```

def sum_scores(s):
 """
 LeetCode 2223. 构造字符串的总得分和
 你需要从空字符串开始构造一个长度为 n 的字符串 s，构造过程为每次给当前字符串前面添加一个字符。
 构造过程中得到的所有字符串编号为 1 到 n，其中长度为 i 的字符串编号为 si。
 si 的得分为 si 和 sn 的最长公共前缀的长度（注意 s == sn）。
 请你返回每一个 si 的得分之和。
 """

 time complexity: O(n)
 space complexity: O(n)
 """

```

```

n = len(s)

计算 Z 函数
z = z_function(s)

计算得分总和
每个 si 的得分就是 z[i]
return sum(z)

```

```

def minimum_time_to_initial_state(word, k):
 """
 LeetCode 3031. 将单词恢复初始状态所需的最短时间 II
 给你一个下标从 0 开始的字符串 word 和一个整数 k。
 每一秒执行以下操作：
 1. 移除 word 的前 k 个字符
 """

 time complexity: O(n)
 space complexity: O(1)

```

2. 在 word 的末尾添加 k 个任意字符

返回将 word 恢复到初始状态所需的最短时间（该时间必须大于零）。

时间复杂度：O(n)

空间复杂度：O(n)

"""

```
n = len(word)
```

```
计算 Z 函数
```

```
z = z_function(word)
```

```
查找满足条件的最短时间
```

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

```
 # 如果从位置 i 开始的后缀与原字符串的最长公共前缀长度等于后缀长度
```

```
 # 说明在第(i//k)步后可以恢复原字符串
```

```
 if z[i] >= n - i:
```

```
 return i // k
```

```
最坏情况需要完全替换
```

```
return (n + k - 1) // k
```

```
def extended_kmp_template(a, b):
```

"""

洛谷 P5410 【模板】扩展 KMP (Z 函数)

题目描述：给定两个字符串 a, b，求：

1. b 与 b 每一个后缀串的最长公共前缀长度（即 b 的 Z 函数）

2. a 与 b 每一个后缀串的最长公共前缀长度（即扩展 KMP）

时间复杂度：O(n + m)

空间复杂度：O(n + m)

"""

```
n, m = len(a), len(b)
```

```
计算 b 的 Z 函数
```

```
z = z_function(b)
```

```
计算 a 与 b 的扩展 KMP
```

```
e = extended_kmp(a, b)
```

```
计算异或和
```

```
z_xor = xor_sum(z)
```

```
e_xor = xor_sum(e)
```

```
return z_xor, e_xor

测试洛谷 P5410 扩展 KMP 模板题
if __name__ == "__main__":
 a = input().strip()
 b = input().strip()

 # 计算 b 的 Z 函数
 z = z_function(b)
 # 计算 a 与 b 的扩展 KMP
 e = extended_kmp(a, b)

 print(xor_sum(z))
 print(xor_sum(e))

=====
```