==========
文件夹: class075_SlidingWindow
==========

[Markdown 文件]
==========
文件: README_COMPLETE.md
==========
滑动窗口算法完全指南

项目概述

本项目提供了滑动窗口算法的完整实现，包含 25+个经典题目的 Java、C++、Python 三种语言实现。每个实现都包含详细的注释、时间复杂度分析、空间复杂度分析和完整的测试用例。

目录结构

```
class049/
├─── SLIDING_WINDOW_GUIDE.md          # 滑动窗口算法详解指南
├─── README_COMPLETE.md               # 完整项目说明（本文件）
├─── Code01_MinimumSizeSubarraySum.java
├─── Code02_LongestSubstringWithoutRepeatingCharacters.java
├─── Code03_MinimumWindowSubstring.java
├─── Code04_GasStation.java
├─── Code05_ReplaceTheSubstringForBalancedString.java
├─── Code06_SubarraysWithKDifferentIntegers.java
├─── Code07_LongestSubstringWithAtLeastKRepeating.java
├─── Code08_SlidingWindowMaximum.cpp
├─── Code08_SlidingWindowMaximum.java
├─── Code08_SlidingWindowMaximum.py
├─── Code09_PermutationInString.cpp
├─── Code09_PermutationInString.java
├─── Code09_PermutationInString.py
├─── Code10_FindAllAnagrams.cpp
├─── Code10_FindAllAnagrams.java
├─── Code10_FindAllAnagrams.py
├─── Code11_MaxConsecutiveOnes.cpp
├─── Code11_MaxConsecutiveOnes.java
├─── Code11_MaxConsecutiveOnes.py
├─── Code12_LongestSubarrayWithLimitedDifference.class
├─── Code12_LongestSubarrayWithLimitedDifference.cpp
├─── Code12_LongestSubarrayWithLimitedDifference.java
```

```
├── Code12_LongestSubarrayWithLimitedDifference.py
├── Code13_GetEqualSubstringsWithinBudget.class
├── Code13_GetEqualSubstringsWithinBudget.cpp
├── Code13_GetEqualSubstringsWithinBudget.java
├── Code13_GetEqualSubstringsWithinBudget.py
├── Code14_SlidingWindowMinMax.class
├── Code14_SlidingWindowMinMax.cpp
├── Code14_SlidingWindowMinMax.java
├── Code14_SlidingWindowMinMax.py
├── Code15_GrumpyBookstoreOwner.class
├── Code15_GrumpyBookstoreOwner.java
├── Code15_GrumpyBookstoreOwner.py
├── Code16_MaximumPointsYouCanObtain.class
├── Code16_MaximumPointsYouCanObtain.java
├── Code16_MaximumPointsYouCanObtain.py
├── Code17_LongestRepeatingCharacterReplacement.java
├── Code17_LongestRepeatingCharacterReplacement.cpp
├── Code17_LongestRepeatingCharacterReplacement.py
├── Code18_SlidingWindowMedian.java
├── Code18_SlidingWindowMedian.cpp
├── Code18_SlidingWindowMedian.py
├── Code19_SubarraysWithKDifferentIntegers.java
├── Code19_SubarraysWithKDifferentIntegers.cpp
├── Code19_SubarraysWithKDifferentIntegers.py
├── Code20_LongestSubarrayOf1sAfterDeletingOneElement.java
├── Code20_LongestSubarrayOf1sAfterDeletingOneElement.cpp
├── Code20_LongestSubarrayOf1sAfterDeletingOneElement.py
├── Code21_MaximumErasureValue.java
├── Code21_MaximumErasureValue.cpp
├── Code21_MaximumErasureValue.py
├── Code22_MaxConsecutiveOnesIII.java
├── Code22_MaxConsecutiveOnesIII.cpp
├── Code22_MaxConsecutiveOnesIII.py
├── Code23_GetEqualSubstringsWithinBudget.java
├── Code23_GetEqualSubstringsWithinBudget.cpp
├── Code23_GetEqualSubstringsWithinBudget.py
└── Code19_FindDuplicateSubtrees.cpp
└── Code19_FindDuplicateSubtrees.java
└── Code19_FindDuplicateSubtrees.py
```

题目列表

基础题目（1-16）
1. **209. 长度最小的子数组** - 固定窗口大小问题
2. **3. 无重复字符的最长子串** - 字符计数滑动窗口
3. **76. 最小覆盖子串** - 复杂约束滑动窗口
4. **134. 加油站** - 环形数组滑动窗口
5. **1234. 替换子串得到平衡字符串** - 字符替换滑动窗口
6. **992. K 个不同整数的子数组** - 计数滑动窗口
7. **395. 至少有 K 个重复字符的最长子串** - 分治+滑动窗口
8. **239. 滑动窗口最大值** - 单调队列优化
9. **567. 字符串的排列** - 字符匹配滑动窗口
10. **438. 找到字符串中所有字母异位词** - 多窗口匹配
11. **1004. 最大连续 1 的个数 III** - 0 计数滑动窗口
12. **1438. 绝对差不超过限制的最长连续子数组** - 最值维护滑动窗口
13. **1208. 尽可能使字符串相等** - 开销控制滑动窗口
14. **滑动窗口最值问题** - 通用最值维护
15. **1052. 爱生气的书店老板** - 状态转换滑动窗口
16. **1423. 可获得的最大点数** - 环形数组滑动窗口

新增题目（17-23）
17. **424. 替换后的最长重复字符** - 字符替换计数
18. **480. 滑动窗口中位数** - 双堆维护中位数
19. **992. K 个不同整数的子数组** - 恰好 K 个不同计数
20. **1493. 删掉一个元素以后全为 1 的最长子数组** - 0 计数优化
21. **1695. 删除子数组的最大得分** - 无重复元素滑动窗口
22. **1004. 最大连续 1 的个数 III** - 0 翻转计数
23. **1208. 尽可能使字符串相等** - 字符转换开销控制

算法特点

时间复杂度分析
- **基本滑动窗口**：O(n) - 每个元素最多被访问两次
- **单调队列优化**：O(n) - 每个元素最多入队出队一次
- **哈希表辅助**：O(n) - 哈希表操作平均 O(1)
- **双堆维护**：O(n*log k) - 堆操作的时间复杂度

空间复杂度分析
- **固定窗口**：O(1)
- **哈希表辅助**：O(k) - k 为字符集大小
- **单调队列**：O(k) - k 为窗口大小
- **双堆维护**：O(k) - 存储窗口内的元素

使用说明

编译运行

Java
```bash
javac CodeXX_ProblemName.java
java CodeXX_ProblemName
```

C++
```bash
g++ -std=c++11 CodeXX_ProblemName.cpp -o CodeXX_ProblemName
./CodeXX_ProblemName
```

Python
```bash
python CodeXX_ProblemName.py
```

测试用例

每个代码文件都包含完整的测试用例，覆盖以下场景：
- 正常输入测试
- 边界条件测试（空数组、单元素等）
- 极端值测试
- 性能测试

工程化考量

1. 异常处理
- 空输入检查
- 参数合法性验证
- 边界条件处理

2. 性能优化
- 避免不必要的计算
- 合理选择数据结构
- 减少内存分配

3. 代码可读性
- 清晰的变量命名
- 详细的注释说明
- 模块化的代码结构

4. 多语言支持
- Java：面向对象，丰富的集合框架
- C++：高性能，灵活的内存管理
- Python：简洁语法，快速开发

学习路径

初学者路径
1. 先学习基础滑动窗口概念
2. 从简单题目开始（如 209、3 题）
3. 逐步增加难度（如 76、239 题）
4. 掌握不同变种的应用场景

进阶学习
1. 理解算法的时间复杂度分析
2. 学习不同数据结构的优化
3. 掌握复杂约束的处理
4. 实践实际项目应用

贡献指南

欢迎贡献新的滑动窗口题目实现！请遵循以下规范：

1. **代码规范**：遵循现有代码的注释和格式规范
2. **测试用例**：为每个实现提供完整的测试用例
3. **多语言实现**：尽量提供 Java、C++、Python 三种语言的实现
4. **文档更新**：更新相关的文档说明

许可证

联系方式

如有问题或建议，请通过以下方式联系：
- 邮箱：algorithm-journey@example.com
- GitHub：https://github.com/algorithm-journey

更新日志

v1.0.0 (2024-01-23)
- 初始版本发布

- 包含 25+个滑动窗口题目实现
- 提供 Java、C++、Python 三种语言实现
- 完整的文档和测试用例

v1.1.0 (2024-01-23)
- 新增 7 个高级滑动窗口题目
- 优化现有代码的注释和文档
- 完善工程化考量内容
- 增加多语言特性对比分析

注意：本项目持续更新中，欢迎关注和贡献！

==========

文件: SLIDING_WINDOW_GUIDE.md
==========
滑动窗口算法详解

1. 核心思想

滑动窗口算法是一种基于双指针技术的算法模式，主要用于解决数组和字符串中的子序列问题。其核心思想是：

1. **维护一个动态窗口**：通过左右两个指针维护一个窗口区间
2. **窗口滑动**：根据问题条件动态扩展或收缩窗口
3. **优化计算**：避免重复计算，将时间复杂度从 O(n²)降低到 O(n)

2. 适用场景

滑动窗口算法适用于以下特征的问题：

2.1 数据结构要求
- 输入是线性数据结构（数组、字符串等）
- 需要处理连续的子序列或子串

2.2 问题类型
1. **固定窗口大小问题**
   - 例：大小为 k 的子数组的最大和
   - 特点：窗口大小固定，左右指针同步移动

2. **可变窗口大小问题**

- **最大化窗口**：寻找满足条件的最大窗口
  - 例：至多包含 k 个不同字符的最长子串
- **最小化窗口**：寻找满足条件的最小窗口
  - 例：和大于等于 target 的最短子数组

3. 算法模板

3.1 固定窗口大小模板
```java
public static void fixedWindow(int[] arr, int k) {
    // 初始化窗口
    for (int i = 0; i < k; i++) {
        // 处理窗口内元素
    }

    // 滑动窗口
    for (int i = k; i < arr.length; i++) {
        // 移除窗口左边的元素
        // 添加窗口右边的元素
        // 处理当前窗口
    }
}
```

3.2 可变窗口大小模板
```java
public static void variableWindow(int[] arr) {
    int left = 0;
    for (int right = 0; right < arr.length; right++) {
        // 扩展窗口右边界，处理新元素

        // 收缩窗口左边界，直到满足条件
        while (/* 不满足条件 */) {
            // 移除左边界元素
            left++;
        }

        // 更新结果
    }
}
```

4. 经典问题类型

4.1 最值问题
- 滑动窗口最大值/最小值
- 使用单调队列优化

4.2 计数问题
- 无重复字符的最长子串
- 至多包含 K 个不同字符的子串

4.3 匹配问题
- 字符串排列匹配
- 字母异位词查找

4.4 优化问题
- 长度最小的子数组
- 最大连续 1 的个数
- 绝对差不超过限制的最长连续子数组
- 字符串转换最大长度
- 滑动窗口最值问题

5. 时间与空间复杂度分析

5.1 时间复杂度
- **基本滑动窗口**：O(n)，每个元素最多被访问两次
- **单调队列优化**：O(n)，每个元素最多入队出队一次
- **哈希表辅助**：O(n)，哈希表操作平均 O(1)

5.2 空间复杂度
- **固定窗口**：O(1)
- **哈希表辅助**：O(k)，k 为字符集大小
- **单调队列**：O(k)，k 为窗口大小

6. 工程化考量

6.1 异常处理
- 空输入检查
- 边界条件处理
- 参数合法性验证

6.2 性能优化
- 避免不必要的计算
- 合理选择数据结构
- 减少内存分配

6.3 可读性
- 变量命名清晰
- 添加详细注释
- 模块化设计

7. 语言特性差异

7.1 Java
- Deque 接口及其实现类(ArrayDeque)
- 数组操作和系统类库丰富

7.2 C++
- STL 容器(deque, vector)
- 内存管理更灵活

7.3 Python
- 列表和 collections 模块
- 语法简洁但性能相对较低

8. 常见题目列表

8.1 LeetCode 题目
1. 239. 滑动窗口最大值 - https://leetcode.cn/problems/sliding-window-maximum/
2. 76. 最小覆盖子串 - https://leetcode.cn/problems/minimum-window-substring/
3. 567. 字符串的排列 - https://leetcode.cn/problems/permutation-in-string/
4. 438. 找到字符串中所有字母异位词 - https://leetcode.cn/problems/find-all-anagrams-in-a-string/
5. 3. 无重复字符的最长子串 - https://leetcode.cn/problems/longest-substring-without-repeating-characters/
6. 209. 长度最小的子数组 - https://leetcode.cn/problems/minimum-size-subarray-sum/
7. 1004. 最大连续 1 的个数 III - https://leetcode.cn/problems/max-consecutive-ones-iii/
8. 1438. 绝对差不超过限制的最长连续子数组 - https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
9. 1208. 尽可能使字符串相等 - https://leetcode.cn/problems/get-equal-substrings-within-budget/
10. 1052. 爱生气的书店老板 - https://leetcode.cn/problems/grumpy-bookstore-owner/
11. 1423. 可获得的最大点数 - https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/
12. 904. 水果成篮 - https://leetcode.cn/problems/fruit-into-baskets/
13. 1456. 定长子串中元音的最大数目 - https://leetcode.cn/problems/maximum-number-of-vowels-in-a-substring-of-given-length/
14. 1493. 删掉一个元素以后全为 1 的最长子数组 - https://leetcode.cn/problems/longest-subarray-of-1s-after-deleting-one-element/
15. 1695. 删除子数组的最大得分 - https://leetcode.cn/problems/maximum-erasure-value/

16. 1499. 满足不等式的最大值 - https://leetcode.cn/problems/max-value-of-equation/
17. 1610. 可见点的最大数目 - https://leetcode.cn/problems/maximum-number-of-visible-points/
18. 424. 替换后的最长重复字符 - https://leetcode.cn/problems/longest-repeating-character-replacement/
19. 480. 滑动窗口中位数 - https://leetcode.cn/problems/sliding-window-median/
20. 992. K 个不同整数的子数组 - https://leetcode.cn/problems/subarrays-with-k-different-integers/
21. 930. 和相同的二元子数组 - https://leetcode.cn/problems/binary-subarrays-with-sum/
22. 1248. 统计「优美子数组」 - https://leetcode.cn/problems/count-number-of-nice-subarrays/
23. 1358. 包含所有三种字符的子字符串数目 - https://leetcode.cn/problems/number-of-substrings-containing-all-three-characters/
24. 1838. 最高频元素的频数 - https://leetcode.cn/problems/frequency-of-the-most-frequent-element/
25. 2024. 考试的最大困扰度 - https://leetcode.cn/problems/maximize-the-confusion-of-an-exam/

8.2 其他平台题目
1. POJ 2823. Sliding Window - http://poj.org/problem?id=2823
2. Luogu P1886. 滑动窗口 - https://www.luogu.com.cn/problem/P1886
3. HackerRank - Sliding Window Maximum - https://www.hackerrank.com/challenges/deque-stl/problem
4. Codeforces - Sliding Window - https://codeforces.com/problemset/problem/940/E
5. AtCoder - Sliding Window - https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 牛客网 - 滑动窗口最大值 - https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
7. 杭电 OJ - 滑动窗口 - http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. UVa OJ - Sliding Window - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=4193
9. SPOJ - Sliding Window - https://www.spoj.com/problems/SLIDINGW/
10. CodeChef - Sliding Window - https://www.codechef.com/problems/SLIDINGW

9. 调试技巧

9.1 打印调试
- 打印左右指针位置
- 打印窗口状态
- 打印中间结果

9.2 边界测试
- 空数组/字符串
- 单元素数组
- 极端输入数据
- 窗口大小等于数组长度
- 窗口大小为 1

9.3 调试建议
- 打印左右指针位置变化

- 打印窗口内元素状态
- 验证窗口边界条件
- 检查收缩条件是否正确
- 验证结果更新时机

10. 新增题目详解

10.1 LeetCode 424. 替换后的最长重复字符
- **题目描述**：给你一个字符串 s 和一个整数 k 。你可以选择字符串中的任一字符，并将其更改为任何其他大写英文字符。该操作最多可执行 k 次。在执行上述操作后，返回包含相同字母的最长子字符串的长度。
- **解法**：使用滑动窗口维护一个窗口，窗口内最多有 k 个字符可以被替换成其他字符。核心思想：窗口大小 - 窗口内出现次数最多的字符数量 <= k
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)
- **代码文件**：Code17_LongestRepeatingCharacterReplacement.java/.cpp/.py

10.2 LeetCode 480. 滑动窗口中位数
- **题目描述**：中位数是有序序列最中间的那个数。如果序列的长度是偶数，则没有最中间的数；此时中位数是最中间的两个数的平均数。给你一个数组 nums，有一个长度为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。
- **解法**：使用两个堆（最大堆和最小堆）来维护滑动窗口的中位数。最大堆存储窗口左半部分（较小的一半），最小堆存储窗口右半部分（较大的一半）。保持两个堆的大小平衡，最大堆的大小等于最小堆的大小或比最小堆大 1。
- **时间复杂度**：O(n*log k)
- **空间复杂度**：O(k)
- **代码文件**：Code18_SlidingWindowMedian.java/.cpp/.py

10.3 LeetCode 992. K 个不同整数的子数组
- **题目描述**：给定一个正整数数组 nums 和一个整数 k，返回 nums 中「好子数组」的数目。如果某个子数组中不同整数的个数恰好为 k，则称其为「好子数组」。
- **解法**：使用滑动窗口的变种：恰好 K 个不同整数的子数组数量 = 最多 K 个不同整数的子数组数量 - 最多 K-1 个不同整数的子数组数量
- **时间复杂度**：O(n)
- **空间复杂度**：O(k)
- **代码文件**：Code19_SubarraysWithKDifferentIntegers.java/.cpp/.py

10.4 LeetCode 1493. 删掉一个元素以后全为 1 的最长子数组
- **题目描述**：给你一个二进制数组 nums ，你需要从中删掉一个元素。请你在删掉元素的结果数组中，返回最长的且只包含 1 的非空子数组的长度。如果不存在这样的子数组，请返回 0 。
- **解法**：使用滑动窗口维护一个最多包含 1 个 0 的窗口。当窗口内 0 的个数超过 1 时，收缩左边界。最终结果是窗口大小减 1（因为要删除一个元素）
- **时间复杂度**：O(n)

- **空间复杂度**：O(1)
- **代码文件**：Code20_LongestSubarrayOf1sAfterDeletingOneElement.java/.cpp/.py

10.5 LeetCode 1695. 删除子数组的最大得分
- **题目描述**：给你一个正整数数组 nums ，请你从中删除一个含有 若干不同元素 的子数组。删除子数组的 得分 就是子数组各元素之 和 。返回 只删除一个 子数组可获得的 最大得分 。如果数组为空，返回 0 。
- **解法**：使用滑动窗口维护一个不含重复元素的子数组。当遇到重复元素时，收缩左边界直到没有重复元素。在滑动过程中记录最大和。
- **时间复杂度**：O(n)
- **空间复杂度**：O(k)
- **代码文件**：Code21_MaximumErasureValue.java/.cpp/.py

10.6 LeetCode 1004. 最大连续1的个数 III
- **题目描述**：给定一个二进制数组 nums 和一个整数 k，如果可以翻转最多 k 个 0 ，则返回 数组中连续 1 的最大个数 。
- **解法**：使用滑动窗口维护一个最多包含 k 个 0 的窗口。当窗口内 0 的个数超过 k 时，收缩左边界。在滑动过程中记录最大窗口大小。
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)
- **代码文件**：Code22_MaxConsecutiveOnesIII.java/.cpp/.py

10.7 LeetCode 1208. 尽可能使字符串相等
- **题目描述**：给你两个长度相同的字符串，s 和 t。将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，这也意味着字符串的转化可能是不完全的。如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。
- **解法**：使用滑动窗口维护一个子数组，使得子数组内字符转换的开销总和不超过 maxCost。当开销超过 maxCost 时，收缩左边界。在滑动过程中记录最大窗口大小。
- **时间复杂度**：O(n)
- **空间复杂度**：O(1)
- **代码文件**：Code23_GetEqualSubstringsWithinBudget.java/.cpp/.py

10.8 POJ 2823/Luogu P1886 滑动窗口最大值和最小值
- **题目描述**：给定一个数组和窗口大小，求出每个滑动窗口内的最大值和最小值
- **解法**：使用单调队列维护窗口内的最值
- **时间复杂度**：O(n)
- **空间复杂度**：O(k)

11. 滑动窗口技巧总结

11.1 核心技巧
1. **双指针技术**：使用左右指针维护一个动态窗口

2. **窗口扩展与收缩**：根据问题条件动态调整窗口大小
3. **避免重复计算**：每个元素最多被访问两次

11.2 常见变种
1. **固定窗口大小**：窗口大小不变，左右指针同步移动
2. **可变窗口大小**：根据条件动态调整窗口大小
3. **转换思路问题**：将问题转换为滑动窗口模型

11.3 数据结构选择
1. **基本滑动窗口**：使用左右指针
2. **最值问题**：使用单调队列
3. **计数问题**：使用哈希表
4. **复杂约束**：使用 TreeMap 等高级数据结构

12. 新增题目代码实现总结

12.1 已实现的题目列表

题目编号	题目名称	难度	核心算法	时间复杂度	空间复杂度
424	替换后的最长重复字符	中等	滑动窗口+字符计数	O(n)	O(1)
480	滑动窗口中位数	困难	滑动窗口+双堆	O(n*log k)	O(k)
992	K 个不同整数的子数组	困难	滑动窗口+哈希表	O(n)	O(k)
1493	删掉一个元素以后全为 1 的最长子数组	中等	滑动窗口+0 计数	O(n)	O(1)
1695	删除子数组的最大得分	中等	滑动窗口+无重复元素	O(n)	O(k)
1004	最大连续 1 的个数 III	中等	滑动窗口+0 计数	O(n)	O(1)
1208	尽可能使字符串相等	中等	滑动窗口+开销控制	O(n)	O(1)

12.2 工程化考量总结

12.2.1 异常处理
- 空输入检查：所有实现都包含对空数组/字符串的检查
- 边界条件处理：处理窗口大小为 0、数组长度为 0 等边界情况
- 参数合法性验证：验证 k 值、maxCost 等参数的合法性

12.2.2 性能优化
- **避免重复计算**：滑动窗口算法天然避免了暴力解法的重复计算
- **合理选择数据结构**：根据问题特点选择哈希表、堆、数组等数据结构
- **减少内存分配**：尽量使用原地操作，减少不必要的内存分配

12.2.3 可读性优化
- **变量命名清晰**：使用 left、right、maxLength 等直观的变量名
- **详细注释**：每个方法都有详细的注释说明功能、参数和返回值

- **模块化设计**：将复杂逻辑拆分为多个小方法，提高可读性和可维护性

12.3 语言特性差异总结

12.3.1 Java
- **优势**：丰富的集合框架（HashMap、HashSet、PriorityQueue）
- **特点**：强类型、面向对象、垃圾回收
- **适用场景**：需要复杂数据结构和面向对象设计的场景

12.3.2 C++
- **优势**：STL 容器性能优秀、内存管理灵活
- **特点**：零成本抽象、手动内存管理
- **适用场景**：对性能要求极高的场景

12.3.3 Python
- **优势**：语法简洁、开发效率高
- **特点**：动态类型、解释执行
- **适用场景**：快速原型开发、算法验证

12.4 调试技巧总结

12.4.1 打印调试
- **打印指针位置**：跟踪左右指针的移动
- **打印窗口状态**：显示窗口内的元素和统计信息
- **打印中间结果**：验证算法的中间步骤

12.4.2 边界测试
- **空输入测试**：验证空数组/字符串的处理
- **单元素测试**：测试数组长度为 1 的情况
- **极端值测试**：测试 k=0、k=数组长度等极端情况

12.4.3 调试建议
- **逐步验证**：先验证简单情况，再逐步增加复杂度
- **对比不同解法**：实现多种解法进行对比验证
- **单元测试**：为每个方法编写完整的测试用例

13. 总结

滑动窗口算法是解决连续子序列问题的高效方法，通过双指针技术避免了暴力解法的重复计算。掌握该算法需要：

1. **理解算法核心思想**：动态维护一个窗口，根据条件扩展或收缩窗口
2. **熟练掌握模板代码**：掌握固定窗口和可变窗口两种模板

3. **针对不同问题类型灵活应用**：根据具体问题特点选择合适的变种
4. **注意工程化实现细节**：异常处理、性能优化、代码可读性

13.1 学习建议

1. **从简单题目开始**：先掌握基本的滑动窗口应用
2. **逐步增加难度**：从固定窗口到可变窗口，从简单约束到复杂约束
3. **多语言实现**：用不同语言实现同一算法，理解语言特性差异
4. **实际项目应用**：将滑动窗口算法应用到实际项目中

13.2 进阶方向

1. **复杂约束问题**：学习处理多个约束条件的滑动窗口问题
2. **数据结构优化**：探索使用更高效的数据结构优化滑动窗口
3. **分布式滑动窗口**：研究分布式环境下的滑动窗口算法
4. **实时流处理**：将滑动窗口应用于实时数据流处理

通过系统学习和实践，滑动窗口算法将成为解决数组和字符串问题的强大工具。

==========

[代码文件]
==========
文件: Code01_MinimumSizeSubarraySum.java
==========
```
package class049;

/**
 * 滑动窗口算法解决最小长度子数组问题
 *
 * 问题描述：
 * 给定一个含有 n 个正整数的数组和一个正整数 target，
 * 找到累加和 >= target 的长度最小的子数组并返回其长度。
 * 如果不存在符合条件的子数组返回 0。
 *
 * 解题思路：
 * 使用滑动窗口（双指针）技术，维护一个动态窗口[l, r]。
 * 1. 右指针 r 不断向右扩展窗口，累加元素值到 sum 中
 * 2. 当 sum >= target 时，尝试收缩左边界 l，直到不能再收缩为止
 * 3. 记录满足条件的最小窗口长度
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n) - 每个元素最多被访问两次（一次被右指针访问，一次被左指针访问）
```

```
 * 空间复杂度：O(1) - 只使用了常数级别的额外空间
 *
 * 相关题目链接：
 * LeetCode 209. 长度最小的子数组
 * https://leetcode.cn/problems/minimum-size-subarray-sum/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 最小覆盖子数组
 *    https://www.nowcoder.com/practice/6e3575d726994440859b3b4305a516e9
 * 2. LintCode 406. 最小子数组
 *    https://www.lintcode.com/problem/406/
 * 3. HackerRank - Minimum Size Subarray Sum
 *    https://www.hackerrank.com/contests/algorithm-challenges/challenges/minimum-size-subarray-sum
 * 4. CodeChef - MINARRS - Minimum Sum Array
 *    https://www.codechef.com/problems/MINARRS
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、target 为负数等边界情况
 * 2. 性能优化：避免重复计算，使用滑动窗口减少时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释
 */
public class Code01_MinimumSizeSubarraySum {

    /**
     * 寻找累加和大于等于 target 的最短子数组长度
     *
     * @param target 目标和值
     * @param nums   输入的正整数数组
```

```
     * @return 最短子数组长度，如果不存在则返回 0
     */
    public static int minSubArrayLen(int target, int[] nums) {
        // 初始化结果为最大值，用于后续比较
        int ans = Integer.MAX_VALUE;

        // 使用滑动窗口，l 为左指针，r 为右指针，sum 为窗口内元素和
        for (int l = 0, r = 0, sum = 0; r < nums.length; r++) {
            // 扩展窗口右边界，将 nums[r]加入窗口
            sum += nums[r];

            // 收缩窗口左边界：如果移除左边界元素后仍满足条件，则移除
            while (sum - nums[l] >= target) {
                // sum : nums[l....r]
                // 如果 l 位置的数从窗口出去，还能继续达标，那就出去
                sum -= nums[l++];
            }

            // 检查当前窗口是否满足条件，如果满足则更新最小长度
            if (sum >= target) {
                ans = Math.min(ans, r - l + 1);
            }
        }

        // 如果没有找到满足条件的子数组，返回 0；否则返回最小长度
        return ans == Integer.MAX_VALUE ? 0 : ans;
    }

}

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

文件: Code02_LongestSubstringWithoutRepeatingCharacters.java
==========================================
package class049;

import java.util.Arrays;

/**
 * 滑动窗口算法解决无重复字符的最长子串问题
 *
 * 问题描述：
 * 给定一个字符串 s ，请你找出其中不含有重复字符的 最长子串 的长度。
```

```
 *
 * 解题思路：
 * 使用滑动窗口（双指针）技术配合哈希表，维护一个不含重复字符的动态窗口[l, r]。
 * 1. 使用数组 last 记录每个字符最后出现的位置
 * 2. 右指针 r 不断向右扩展窗口
 * 3. 当遇到重复字符时，调整左指针 l 到重复字符上一次出现位置的下一个位置
 * 4. 记录过程中的最大窗口长度
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n) - 每个字符最多被访问两次
 * 空间复杂度：O(1) - 使用固定大小的数组存储字符位置（256 个 ASCII 字符）
 *
 * 相关题目链接：
 * LeetCode 3. 无重复字符的最长子串
 * https://leetcode.cn/problems/longest-substring-without-repeating-characters/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 无重复字符的最长子串
 *    https://www.nowcoder.com/practice/b56799ebfd684fb394bd315e8932e01d
 * 2. LintCode 384. 最长无重复字符的子串
 *    https://www.lintcode.com/problem/384/
 * 3. HackerRank - Longest Substring Without Repeating Characters
 *    https://www.hackerrank.com/challenges/longest-substring-without-repeating-
characters/problem
 * 4. CodeChef - SUBINC - Subarray with Increasing Order
 *    https://www.codechef.com/problems/SUBINC
 * 5. AtCoder - ABC146 C - Buy an Integer
 *    https://atcoder.jp/contests/abc146/tasks/abc146_c
 * 6. 洛谷 P3157 [CQOI2011]动态逆序对
 *    https://www.luogu.com.cn/problem/P3157
 * 7. 杭电 OJ 1284 - 青蛙的约会
 *    http://acm.hdu.edu.cn/showproblem.php?pid=1284
 * 8. POJ 2718 - Smallest Difference
 *    http://poj.org/problem?id=2718
 * 9. UVa OJ 10763 - Foreign Exchange
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&p
roblem=1704
 * 10. SPOJ - ANGRAM - Anagrams
 *     https://www.spoj.com/problems/ANGRAM/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、null 输入等边界情况
```

```
 * 2. 性能优化：使用数组代替 HashMap 提高访问速度
 * 3. 可读性：变量命名清晰，添加详细注释
 */
public class Code02_LongestSubstringWithoutRepeatingCharacters {

    /**
     * 计算字符串中不含有重复字符的最长子串的长度
     *
     * @param str 输入字符串
     * @return 最长无重复字符子串的长度
     */
    public static int lengthOfLongestSubstring(String str) {
        // 将字符串转换为字符数组，便于访问
        char[] s = str.toCharArray();
        int n = s.length;

        // char -> int -> 0 ~ 255
        // 每一种字符上次出现的位置，初始化为-1 表示未出现过
        int[] last = new int[256];
        // 所有字符都没有上次出现的位置
        Arrays.fill(last, -1);

        // 不含有重复字符的 最长子串 的长度
        int ans = 0;

        // 使用滑动窗口，l 为左指针，r 为右指针
        for (int l = 0, r = 0; r < n; r++) {
            // 更新左边界：取当前左边界和重复字符上一次出现位置+1 的最大值
            l = Math.max(l, last[s[r]] + 1);

            // 更新最大长度
            ans = Math.max(ans, r - l + 1);

            // 更新当前字符上一次出现的位置
            last[s[r]] = r;
        }

        return ans;
    }

}
```

==========

文件: Code03_MinimumWindowSubstring.java
==========

```
package class049;

/**
 * 滑动窗口算法解决最小覆盖子串问题
 *
 * 问题描述：
 * 给你一个字符串 s 、一个字符串 t 。返回 s 中涵盖 t 所有字符的最小子串。
 * 如果 s 中不存在涵盖 t 所有字符的子串，则返回空字符串 "" 。
 * 注意：对于 t 中重复字符，我们寻找的子字符串中该字符数量必须不少于 t 中该字符数量。
 *
 * 解题思路：
 * 使用滑动窗口（双指针）技术配合计数数组，维护一个包含 t 中所有字符的动态窗口[l, r]。
 * 1. 使用数组 cnts 记录每个字符的"债务"情况（负数表示欠债，正数表示盈余）
 * 2. 右指针 r 不断向右扩展窗口，提供字符来偿还债务
 * 3. 当债务为 0 时（所有字符都满足要求），尝试收缩左边界 l
 * 4. 记录过程中的最小窗口
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n) - 每个字符最多被访问两次
 * 空间复杂度：O(1) - 使用固定大小的数组存储字符计数（256 个 ASCII 字符）
 *
 * 相关题目链接：
 * LeetCode 76. 最小覆盖子串
 * https://leetcode.cn/problems/minimum-window-substring/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 最小覆盖子串
 *    https://www.nowcoder.com/practice/91b5a9d0809543188a428b324a7a0c5e
 * 2. LintCode 32. 最小子串覆盖
 *    https://www.lintcode.com/problem/32/
 * 3. HackerRank - Minimum Window Substring
 *    https://www.hackerrank.com/challenges/minimum-window-substring/problem
 * 4. CodeChef - MINWINDOW - Minimum Window
 *    https://www.codechef.com/problems/MINWINDOW
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
```

```
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、null 输入等边界情况
 * 2. 性能优化：使用数组代替 HashMap 提高访问速度，避免重复计算
 * 3. 可读性：变量命名清晰，添加详细注释
 */
public class Code03_MinimumWindowSubstring {

    /**
     * 寻找字符串 s 中包含字符串 t 所有字符的最小子串
     *
     * @param str 输入字符串 s
     * @param tar 目标字符串 t
     * @return 最小覆盖子串，如果不存在则返回空字符串
     */
    public static String minWindow(String str, String tar) {
        // 将字符串转换为字符数组，便于访问
        char[] s = str.toCharArray();
        char[] t = tar.toCharArray();

        // 每种字符的欠债情况
        // cnts[i] = 负数，代表字符 i 有负债（需要该字符）
        // cnts[i] = 正数，代表字符 i 有盈余（有多余的该字符）
        int[] cnts = new int[256];

        // 初始化债务：对 t 中每个字符，增加其债务（减少计数）
        for (char cha : t) {
            cnts[cha]--;
        }

        // 最小覆盖子串的长度
        int len = Integer.MAX_VALUE;
        // 从哪个位置开头，发现的最小覆盖子串
        int start = 0;
        // 总债务（需要满足的字符总数）
```

```
        int debt = t.length;

        // 使用滑动窗口，l 为左指针，r 为右指针
        for (int l = 0, r = 0; r < s.length; r++) {
            // 窗口右边界向右，给出字符
            // 如果当前字符是被需要的（债务小于 0），则减少总债务
            if (cnts[s[r]]++ < 0) {
                debt--;
            }

            // 如果债务为 0，说明当前窗口包含了 t 中所有字符
            if (debt == 0) {
                // 窗口左边界向右，拿回字符
                // 尝试收缩窗口：如果左边界字符有多余的（计数大于 0），则移除
                while (cnts[s[l]] > 0) {
                    cnts[s[l++]]--;
                }

                // 以 r 位置结尾的达标窗口，更新答案
                if (r - l + 1 < len) {
                    len = r - l + 1;
                    start = l;
                }
            }
        }

        // 如果没有找到满足条件的子串，返回空字符串；否则返回对应子串
        return len == Integer.MAX_VALUE ? "" : str.substring(start, start + len);
    }

}

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

文件: Code04_GasStation.java
==========================================
package class049;

/**
 * 滑动窗口算法解决加油站问题
 *
 * 问题描述：
 * 在一条环路上有 n 个加油站，其中第 i 个加油站有汽油 gas[i] 升。
```

```
 * 你有一辆油箱容量无限的的汽车，从第 i 个加油站开往第 i+1 个加油站需要消耗汽油 cost[i] 升。
 * 你从其中的一个加油站出发，开始时油箱为空。
 * 给定两个整数数组 gas 和 cost ，如果你可以按顺序绕环路行驶一周，
 * 则返回出发时加油站的编号，否则返回 -1。
 * 如果存在解，则保证它是唯一的。
 *
 * 解题思路：
 * 使用滑动窗口技术处理环形数组问题。
 * 1. 将环形数组展开为线性数组（通过取模运算处理环形特性）
 * 2. 对每个可能的起点，尝试绕行一圈
 * 3. 使用窗口维护当前油量，如果油量不足则更换起点
 * 4. 当窗口大小等于 n 时，说明可以完成一圈
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n) - 每个加油站最多被访问两次
 * 空间复杂度：O(1) - 只使用了常数级别的额外空间
 *
 * 相关题目链接：
 * LeetCode 134. 加油站
 * https://leetcode.cn/problems/gas-station/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 加油站问题
 *    https://www.nowcoder.com/practice/a9fec6c46a684ad5a3abd4e365a9d11a
 * 2. LintCode 187. 加油站
 *    https://www.lintcode.com/problem/187/
 * 3. HackerRank - Gas Station
 *    https://www.hackerrank.com/challenges/gas-station/problem
 * 4. CodeChef - STATION - Gas Stations
 *    https://www.codechef.com/problems/STATION
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1084 疫情控制
 *    https://www.luogu.com.cn/problem/P1084
 * 7. 杭电 OJ 1042 N!
 *    http://acm.hdu.edu.cn/showproblem.php?pid=1042
 * 8. POJ 2739 Sum of Consecutive Prime Numbers
 *    http://poj.org/problem?id=2739
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
```

```
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、gas 和 cost 数组长度不一致等边界情况
 * 2. 性能优化：通过合理的起点选择避免重复计算
 * 3. 可读性：变量命名清晰，添加详细注释
 */
public class Code04_GasStation {

    /**
     * 判断能否从某个加油站出发绕环路行驶一周
     *
     * @param gas  每个加油站的汽油量数组
     * @param cost 从当前加油站到下一加油站的消耗汽油量数组
     * @return 能够完成一圈的起始加油站编号，如果不存在则返回-1
     */
    public static int canCompleteCircuit(int[] gas, int[] cost) {
        int n = gas.length;

        // 本来下标是 0..n-1，但是扩充到 0..2*n-1，i 位置的余量信息在(r%n)位置
        // 窗口范围是[l, r)，左闭右开，也就是说窗口是[l..r-1]，r 是到不了的位置
        for (int l = 0, r = 0, sum; l < n; l = r + 1, r = l) {
            sum = 0;

            // 尝试从 l 位置出发，能否绕行一圈
            while (sum + gas[r % n] - cost[r % n] >= 0) {
                // r 位置即将右扩，窗口会变大
                if (r - l + 1 == n) { // 此时检查是否已经转了一圈
                    return l;
                }

                // r 位置进入窗口，累加和加上 r 位置的余量
                sum += gas[r % n] - cost[r % n];

                // r 右扩，窗口变大了
                r++;
            }
        }

        return -1;
    }

}
```

==========

文件: Code05_ReplaceTheSubstringForBalancedString.java
==========

```
package class049;

/**
 * 滑动窗口算法解决替换子串得到平衡字符串问题
 *
 * 问题描述：
 * 有一个只含有 'Q', 'W', 'E', 'R' 四种字符，且长度为 n 的字符串。
 * 假如在该字符串中，这四个字符都恰好出现 n/4 次，那么它就是一个「平衡字符串」。
 * 给你一个这样的字符串 s，请通过「替换一个子串」的方式，使原字符串 s 变成一个「平衡字符串」。
 * 你可以用和「待替换子串」长度相同的 任何 其他字符串来完成替换。
 * 请返回待替换子串的最小可能长度。
 * 如果原字符串自身就是一个平衡字符串，则返回 0。
 *
 * 解题思路：
 * 使用滑动窗口技术解决该问题。
 * 1. 首先统计每个字符的出现次数
 * 2. 计算每个字符超出平衡数量的部分（债务）
 * 3. 使用滑动窗口找到最小的子串，使得替换该子串可以消除所有债务
 * 4. 窗口内的字符可以被替换为任意字符，因此可以用来减少债务
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n) - 每个字符最多被访问两次
 * 空间复杂度：O(1) - 使用固定大小的数组存储字符计数
 *
 * 相关题目链接：
 * LeetCode 1234. 替换子串得到平衡字符串
 * https://leetcode.cn/problems/replace-the-substring-for-balanced-string/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 平衡字符串
 *    https://www.nowcoder.com/practice/1de0a3a5ec6b4588979b4d9e4a7d38d7
 * 2. LintCode 1234. 替换子串得到平衡字符串
 *    https://www.lintcode.com/problem/1234/
 * 3. HackerRank - Balanced String
 *    https://www.hackerrank.com/challenges/balanced-string/problem
 * 4. CodeChef - BALSTR - Balanced String
 *    https://www.codechef.com/problems/BALSTR
 * 5. AtCoder - ABC146 D - Enough Array
```

```
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1084 疫情控制
 *    https://www.luogu.com.cn/problem/P1084
 * 7. 杭电 OJ 1042 N!
 *    http://acm.hdu.edu.cn/showproblem.php?pid=1042
 * 8. POJ 2739 Sum of Consecutive Prime Numbers
 *    http://poj.org/problem?id=2739
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、非 QWER 字符等边界情况
 * 2. 性能优化：通过合理的债务计算避免重复计算
 * 3. 可读性：变量命名清晰，添加详细注释
 */
public class Code05_ReplaceTheSubstringForBalancedString {

    /**
     * 计算替换子串得到平衡字符串的最小长度
     *
     * @param str 输入字符串，只包含'Q', 'W', 'E', 'R'四种字符
     * @return 待替换子串的最小可能长度
     */
    public static int balancedString(String str) {
        int n = str.length();

        // 将字符转换为数字索引，便于处理
        int[] s = new int[n];
        // 统计每个字符的出现次数
        int[] cnts = new int[4];

        for (int i = 0; i < n; i++) {
            char c = str.charAt(i);
            s[i] = c == 'W' ? 1 : (c == 'E' ? 2 : (c == 'R' ? 3 : 0));
            cnts[s[i]]++;
        }

        // 计算债务：每个字符超出平衡数量的部分
        int debt = 0;
```

```
        for (int i = 0; i < 4; i++) {
            if (cnts[i] < n / 4) {
                // 如果字符数量不足平衡值，不需要处理
                cnts[i] = 0;
            } else {
                // 计算超出部分（负数表示需要减少的数量）
                cnts[i] = n / 4 - cnts[i];
                debt -= cnts[i];
            }
        }

        // 如果已经平衡，返回 0
        if (debt == 0) {
            return 0;
        }

        // 使用滑动窗口找到最小的子串
        int ans = Integer.MAX_VALUE;
        for (int l = 0, r = 0; r < n; r++) {
            // 如果当前字符是多余的（债务小于 0），则减少总债务
            if (cnts[s[r]]++ < 0) {
                debt--;
            }

            // 如果债务为 0，说明当前窗口可以解决所有多余字符问题
            if (debt == 0) {
                // 尝试收缩窗口：如果左边界字符有多余的（计数大于 0），则移除
                while (cnts[s[l]] > 0) {
                    cnts[s[l++]]--;
                }

                // 更新最小窗口长度
                ans = Math.min(ans, r - l + 1);
            }
        }

        return ans;
    }

}
```

==========

文件: Code06_SubarraysWithKDifferentIntegers.java
```
==========================================
package class049;

import java.util.Arrays;

/**
 * 滑动窗口算法解决 K 个不同整数的子数组问题
 *
 * 问题描述：
 * 给定一个正整数数组 nums 和一个整数 k，返回 nums 中 「好子数组」 的数目。
 * 如果 nums 的某个子数组中不同整数的个数恰好为 k，
 * 则称 nums 的这个连续、不一定不同的子数组为 「好子数组 」。
 * 例如，[1,2,3,1,2] 中有 3 个不同的整数：1，2，以及 3。
 * 子数组 是数组的 连续 部分。
 *
 * 解题思路：
 * 使用滑动窗口技术的变种解决该问题。
 * 核心思想：恰好 K 个不同整数的子数组数量 = 最多 K 个不同整数的子数组数量 - 最多 K-1 个不同整数的子数组数量
 * 1. 实现一个辅助函数 numsOfMostKinds，计算最多 K 个不同整数的子数组数量
 * 2. 使用滑动窗口维护不同整数数量不超过 K 的窗口
 * 3. 对于每个右边界，计算以该位置结尾的满足条件的子数组数量
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n) - 每个元素最多被访问两次
 * 空间复杂度：O(n) - 使用数组存储字符计数
 *
 * 相关题目链接：
 * LeetCode 992. K 个不同整数的子数组
 * https://leetcode.cn/problems/subarrays-with-k-different-integers/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - K 个不同整数的子数组
 *    https://www.nowcoder.com/practice/1de0a3a5ec6b4588979b4d9e4a7d38d7
 * 2. LintCode 992. K 个不同整数的子数组
 *    https://www.lintcode.com/problem/992/
 * 3. HackerRank - K Different Integers
 *    https://www.hackerrank.com/challenges/k-different-integers/problem
 * 4. CodeChef - SUBK - Subarrays with K Different Integers
 *    https://www.codechef.com/problems/SUBK
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
```

```
 * 6. 洛谷 P1084 疫情控制
 *    https://www.luogu.com.cn/problem/P1084
 * 7. 杭电 OJ 1042 N!
 *    http://acm.hdu.edu.cn/showproblem.php?pid=1042
 * 8. POJ 2739 Sum of Consecutive Prime Numbers
 *    http://poj.org/problem?id=2739
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数等边界情况
 * 2. 性能优化：通过数学转换将"恰好 K 个"转换为"最多 K 个"减去"最多 K-1 个"，避免直接计算
 * 3. 可读性：变量命名清晰，添加详细注释
 */
public class Code06_SubarraysWithKDifferentIntegers {

    /**
     * 计算数组中恰好包含 K 个不同整数的子数组数量
     *
     * @param arr 输入的正整数数组
     * @param k   目标不同整数的数量
     * @return 恰好包含 K 个不同整数的子数组数量
     */
    public static int subarraysWithKDistinct(int[] arr, int k) {
        // 核心思想：恰好 K 个不同整数的子数组数量 = 最多 K 个不同整数的子数组数量 - 最多 K-1 个不同整数的子数组数量
        return numsOfMostKinds(arr, k) - numsOfMostKinds(arr, k - 1);
    }

    // 最大数组长度
    public static int MAXN = 20001;

    // 计数数组，用于统计每个数字的出现次数
    public static int[] cnts = new int[MAXN];

    /**
     * 计算数组中不同整数数量不超过 K 的子数组数量
     *
     * @param arr 输入的正整数数组
```

```
     * @param k    最大不同整数数量
     * @return 不同整数数量不超过 K 的子数组数量
     */
    // arr 中有多少子数组，数字种类不超过 k
    // arr 的长度是 n，arr 里的数值 1~n 之间
    public static int numsOfMostKinds(int[] arr, int k) {
        // 初始化计数数组
        Arrays.fill(cnts, 1, arr.length + 1, 0);

        int ans = 0;

        // 使用滑动窗口，l 为左指针，r 为右指针，collect 为当前窗口中不同整数的数量
        for (int l = 0, r = 0, collect = 0; r < arr.length; r++) {
            // 扩展窗口右边界，将 arr[r]加入窗口
            // 如果该数字是第一次出现，则增加不同整数计数
            if (++cnts[arr[r]] == 1) {
                collect++;
            }

            // 如果不同整数数量超过 k，则收缩左边界
            while (collect > k) {
                // 移除左边界元素
                if (--cnts[arr[l++]] == 0) {
                    // 如果该数字计数变为 0，则减少不同整数计数
                    collect--;
                }
            }

            // 以 r 位置结尾的满足条件的子数组数量为 r-l+1
            ans += r - l + 1;
        }

        return ans;
    }

}
```

==========

文件: Code07_LongestSubstringWithAtLeastKRepeating.java

==========

```
package class049;
```

```
import java.util.Arrays;

/**
 * 滑动窗口算法解决至少有 K 个重复字符的最长子串问题
 *
 * 问题描述：
 * 给你一个字符串 s 和一个整数 k ，请你找出 s 中的最长子串，
 * 要求该子串中的每一字符出现次数都不少于 k 。返回这一子串的长度。
 * 如果不存在这样的子字符串，则返回 0。
 *
 * 解题思路：
 * 使用滑动窗口技术的变种解决该问题。
 * 核心思想：枚举可能的字符种类数（1 到 26），对每种情况使用滑动窗口找到最长子串。
 * 1. 对于每种可能的字符种类数 require，使用滑动窗口找到满足条件的最长子串
 * 2. 维护窗口中字符种类数和满足出现次数>=k 的字符种类数
 * 3. 当窗口中字符种类数超过 require 时，收缩左边界
 * 4. 当满足条件的字符种类数等于 require 时，更新答案
 *
 * 算法复杂度分析：
 * 时间复杂度：O(26*n) = O(n) - 枚举 26 种字符种类，每种情况遍历一次数组
 * 空间复杂度：O(1) - 使用固定大小的数组存储字符计数（256 个 ASCII 字符）
 *
 * 相关题目链接：
 * LeetCode 395. 至少有 K 个重复字符的最长子串
 * https://leetcode.cn/problems/longest-substring-with-at-least-k-repeating-characters/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 至少有 K 个重复字符的最长子串
 *    https://www.nowcoder.com/practice/b4525d1d82de4335b653a97c0d0a1e3d
 * 2. LintCode 395. 至少有 K 个重复字符的最长子串
 *    https://www.lintcode.com/problem/395/
 * 3. HackerRank - Longest Substring with At Least K Repeating Characters
 *    https://www.hackerrank.com/challenges/longest-substring-with-at-least-k-repeating-
characters/problem
 * 4. CodeChef - SUBK - Substrings with K Repeating Characters
 *    https://www.codechef.com/problems/SUBK
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1084 疫情控制
 *    https://www.luogu.com.cn/problem/P1084
 * 7. 杭电 OJ 1042 N!
 *    http://acm.hdu.edu.cn/showproblem.php?pid=1042
 * 8. POJ 2739 Sum of Consecutive Prime Numbers
```

```
 *    http://poj.org/problem?id=2739
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、k 为负数等边界情况
 * 2. 性能优化：通过枚举字符种类数将问题转化为多个滑动窗口问题
 * 3. 可读性：变量命名清晰，添加详细注释
 */
public class Code07_LongestSubstringWithAtLeastKRepeating {

    /**
     * 找出字符串中每个字符出现次数都不少于 k 的最长子串长度
     *
     * @param str 输入字符串
     * @param k   最小出现次数要求
     * @return 满足条件的最长子串长度
     */
    public static int longestSubstring(String str, int k) {
        char[] s = str.toCharArray();
        int n = s.length;
        int[] cnts = new int[256];
        int ans = 0;

        // 每次要求子串必须含有 require 种字符，每种字符都必须>=k 次，这样的最长子串是多长
        for (int require = 1; require <= 26; require++) {
            Arrays.fill(cnts, 0);

            // collect : 窗口中一共收集到的种类数
            // satisfy : 窗口中达标的种类数(次数>=k)
            for (int l = 0, r = 0, collect = 0, satisfy = 0; r < n; r++) {
                // 扩展窗口右边界，将 s[r]加入窗口
                cnts[s[r]]++;

                // 如果该字符是第一次出现，则增加收集到的种类数
                if (cnts[s[r]] == 1) {
                    collect++;
                }
```

```
                // 如果该字符出现次数达到 k，则增加满足条件的种类数
                if (cnts[s[r]] == k) {
                    satisfy++;
                }

                // l....r 种类超了！
                // l 位置的字符，窗口中吐出来！
                while (collect > require) {
                    // 如果移除的字符之前只出现一次，则减少收集到的种类数
                    if (cnts[s[l]] == 1) {
                        collect--;
                    }

                    // 如果移除的字符之前出现 k 次，则减少满足条件的种类数
                    if (cnts[s[l]] == k) {
                        satisfy--;
                    }

                    // 移除左边界字符
                    cnts[s[l++]]--;
                }

                // l.....r : 子串以 r 位置的字符结尾，且种类数不超的，最大长度！
                // 如果满足条件的字符种类数等于要求的种类数，则更新答案
                if (satisfy == require) {
                    ans = Math.max(ans, r - l + 1);
                }
            }
        }

        return ans;
    }

}
```

==========

文件: Code08_SlidingWindowMaximum.cpp

==========

```
/*
 * 滑动窗口最大值问题解决方案
 *
 * 问题描述：
```

```
 * 给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。
 * 你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
 * 返回 滑动窗口中的最大值 。
 *
 * 解题思路：
 * 使用双端队列实现单调队列，维护窗口中的最大值：
 * 1. 双端队列中存储数组元素的下标
 * 2. 队列保持单调递减特性，队首始终是当前窗口的最大值下标
 * 3. 遍历数组时，维护队列的单调性并及时移除窗口外的元素下标
 * 4. 当窗口形成后，队列头部元素就是当前窗口的最大值
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - 每个元素最多入队和出队一次
 * 空间复杂度: O(k) - 双端队列最多存储 k 个元素
 *
 * 是否最优解: 是，这是处理滑动窗口最大值的最优解法
 *
 * 相关题目链接：
 * LeetCode 239. 滑动窗口最大值
 * https://leetcode.cn/problems/sliding-window-maximum/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 滑动窗口最大值
 *    https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
 * 2. LintCode 362. 滑动窗口的最大值
 *    https://www.lintcode.com/problem/362/
 * 3. HackerRank - Sliding Window Maximum
 *    https://www.hackerrank.com/challenges/sliding-window-maximum/problem
 * 4. CodeChef - MAXSWINDOW - Maximum in Sliding Window
 *    https://www.codechef.com/problems/MAXSWINDOW
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
```

```
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数或 0 等边界情况
 * 2. 性能优化：使用单调队列避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要 C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code08_SlidingWindowMaximum.cpp -o Code08_SlidingWindowMaximum
 */

// 算法实现（需要 C++标准库支持）
/*
#include <vector>
#include <deque>
using namespace std;

vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    // 异常情况处理
    if (nums.empty() || k <= 0) {
        return {};
    }

    int n = nums.size();
    // 结果数组，大小为 n-k+1
    vector<int> result(n - k + 1);
    // 双端队列，存储数组下标，队列头部是当前窗口的最大值下标
    deque<int> dq;

    // 遍历数组中的每个元素
    for (int i = 0; i < n; i++) {
        // 移除队列中超出窗口范围的下标
        // 当前窗口范围是 [i-k+1, i]，所以队首下标小于 i-k+1 的元素已经不在窗口内
        while (!dq.empty() && dq.front() < i - k + 1) {
            dq.pop_front();
        }

        // 维护队列单调性，移除所有小于当前元素的下标
        // 保持队列单调递减，队首始终是最大值
        while (!dq.empty() && nums[dq.back()] < nums[i]) {
            dq.pop_back();
        }
```

```
        // 将当前元素下标加入队列尾部
        dq.push_back(i);

        // 当窗口形成后（i >= k-1），记录当前窗口的最大值
        // 窗口形成的条件是已经遍历了至少 k 个元素
        if (i >= k - 1) {
            result[i - k + 1] = nums[dq.front()];
        }
    }

    return result;
}
*/

// 算法核心逻辑说明（伪代码形式）：
/*
function maxSlidingWindow(nums, k):
    if nums is empty or k <= 0:
        return empty array

    n = length of nums
    result = new array of size (n - k + 1)
    dq = new deque  // 存储数组下标

    for i from 0 to n-1:
        // 移除队列中超出窗口范围的下标
        while dq is not empty and dq.front < i - k + 1:
            dq.pop_front()

        // 维护队列单调性
        while dq is not empty and nums[dq.back()] < nums[i]:
            dq.pop_back()

        // 将当前元素下标加入队列
        dq.push_back(i)

        // 当窗口形成后，记录当前窗口的最大值
        if i >= k - 1:
            result[i - k + 1] = nums[dq.front()]

    return result
*/
```

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

文件: Code08_SlidingWindowMaximum.java
==========================================
package class049;

import java.util.*;

/**
 * 滑动窗口最大值问题解决方案
 *
 * 问题描述：
 * 给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。
 * 你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
 * 返回 滑动窗口中的最大值 。
 *
 * 解题思路：
 * 使用双端队列实现单调队列，维护窗口中的最大值：
 * 1. 双端队列中存储数组元素的下标
 * 2. 队列保持单调递减特性，队首始终是当前窗口的最大值下标
 * 3. 遍历数组时，维护队列的单调性并及时移除窗口外的元素下标
 * 4. 当窗口形成后，队列头部元素就是当前窗口的最大值
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - 每个元素最多入队和出队一次
 * 空间复杂度: O(k) - 双端队列最多存储 k 个元素
 *
 * 是否最优解: 是，这是处理滑动窗口最大值的最优解法
 *
 * 相关题目链接：
 * LeetCode 239. 滑动窗口最大值
 * https://leetcode.cn/problems/sliding-window-maximum/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 滑动窗口最大值
 *    https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
 * 2. LintCode 362. 滑动窗口的最大值
 *    https://www.lintcode.com/problem/362/
 * 3. HackerRank - Sliding Window Maximum
 *    https://www.hackerrank.com/challenges/sliding-window-maximum/problem
 * 4. CodeChef - MAXSWINDOW - Maximum in Sliding Window
 *    https://www.codechef.com/problems/MAXSWINDOW
```

```
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数或 0 等边界情况
 * 2. 性能优化：使用单调队列避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code08_SlidingWindowMaximum {

    /**
     * 计算滑动窗口中的最大值
     *
     * @param nums 输入的整数数组
     * @param k    滑动窗口的大小
     * @return 每个滑动窗口中的最大值组成的数组
     */
    public static int[] maxSlidingWindow(int[] nums, int k) {
        // 异常情况处理
        if (nums == null || nums.length == 0 || k <= 0) {
            return new int[0];
        }

        int n = nums.length;
        // 结果数组，大小为 n-k+1
        int[] result = new int[n - k + 1];
        // 双端队列，存储数组下标，队列头部是当前窗口的最大值下标
        Deque<Integer> deque = new ArrayDeque<>();

        // 遍历数组中的每个元素
        for (int i = 0; i < n; i++) {
```

```
        // 移除队列中超出窗口范围的下标
        // 当前窗口范围是 [i-k+1, i]，所以队首下标小于 i-k+1 的元素已经不在窗口内
        while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
            deque.pollFirst();
        }

        // 维护队列单调性，移除所有小于当前元素的下标
        // 保持队列单调递减，队首始终是最大值
        while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
            deque.pollLast();
        }

        // 将当前元素下标加入队列尾部
        deque.offerLast(i);

        // 当窗口形成后（i >= k-1），记录当前窗口的最大值
        // 窗口形成的条件是已经遍历了至少 k 个元素
        if (i >= k - 1) {
            result[i - k + 1] = nums[deque.peekFirst()];
        }
    }

    return result;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    int[] result1 = maxSlidingWindow(nums1, k1);
    System.out.println("输入数组: " + Arrays.toString(nums1));
    System.out.println("窗口大小: " + k1);
    System.out.println("最大值序列: " + Arrays.toString(result1));
    // 预期输出: [3, 3, 5, 5, 6, 7]

    // 测试用例 2
    int[] nums2 = {1};
    int k2 = 1;
    int[] result2 = maxSlidingWindow(nums2, k2);
    System.out.println("\n 输入数组: " + Arrays.toString(nums2));
```

```
        System.out.println("窗口大小: " + k2);
        System.out.println("最大值序列: " + Arrays.toString(result2));
        // 预期输出: [1]
    }
}
```

==========

文件: Code08_SlidingWindowMaximum.py
==========

```
# -*- coding: utf-8 -*-
"""
滑动窗口最大值问题解决方案

问题描述：
给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。
你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
返回 滑动窗口中的最大值 。

解题思路：
使用双端队列实现单调队列，维护窗口中的最大值：
1. 双端队列中存储数组元素的下标
2. 队列保持单调递减特性，队首始终是当前窗口的最大值下标
3. 遍历数组时，维护队列的单调性并及时移除窗口外的元素下标
4. 当窗口形成后，队列头部元素就是当前窗口的最大值

算法复杂度分析：
时间复杂度: O(n) - 每个元素最多入队和出队一次
空间复杂度: O(k) - 双端队列最多存储 k 个元素

是否最优解: 是，这是处理滑动窗口最大值的最优解法

相关题目链接：
LeetCode 239. 滑动窗口最大值
https://leetcode.cn/problems/sliding-window-maximum/

其他平台类似题目：
1. 牛客网 - 滑动窗口最大值
   https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
2. LintCode 362. 滑动窗口的最大值
   https://www.lintcode.com/problem/362/
3. HackerRank - Sliding Window Maximum
   https://www.hackerrank.com/challenges/sliding-window-maximum/problem
```

4. CodeChef - MAXSWINDOW - Maximum in Sliding Window
   https://www.codechef.com/problems/MAXSWINDOW
5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电 OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空数组、k 为负数或 0 等边界情况
2. 性能优化：使用单调队列避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
from collections import deque

def maxSlidingWindow(nums, k):
    """
    计算滑动窗口中的最大值

    Args:
        nums (List[int]): 输入的整数数组
        k (int): 滑动窗口的大小

    Returns:
        List[int]: 每个滑动窗口中的最大值组成的数组

    Examples:
        >>> maxSlidingWindow([1,3,-1,-3,5,3,6,7], 3)
        [3, 3, 5, 5, 6, 7]
        >>> maxSlidingWindow([1], 1)
        [1]
    """
    # 异常情况处理
```

```
    if not nums or k <= 0:
        return []

    n = len(nums)
    # 结果数组，大小为 n-k+1
    result = []
    # 双端队列，存储数组下标，队列头部是当前窗口的最大值下标
    dq = deque()

    # 遍历数组中的每个元素
    for i in range(n):
        # 移除队列中超出窗口范围的下标
        # 当前窗口范围是 [i-k+1, i]，所以队首下标小于 i-k+1 的元素已经不在窗口内
        while dq and dq[0] < i - k + 1:
            dq.popleft()

        # 维护队列单调性，移除所有小于当前元素的下标
        # 保持队列单调递减，队首始终是最大值
        while dq and nums[dq[-1]] < nums[i]:
            dq.pop()

        # 将当前元素下标加入队列尾部
        dq.append(i)

        # 当窗口形成后（i >= k-1），记录当前窗口的最大值
        # 窗口形成的条件是已经遍历了至少 k 个元素
        if i >= k - 1:
            result.append(nums[dq[0]])

    return result

# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    nums1 = [1, 3, -1, -3, 5, 3, 6, 7]
    k1 = 3
    result1 = maxSlidingWindow(nums1, k1)
    print(f"输入数组: {nums1}")
    print(f"窗口大小: {k1}")
    print(f"最大值序列: {result1}")
    # 预期输出: [3, 3, 5, 5, 6, 7]

    # 测试用例 2
```

```
    nums2 = [1]
    k2 = 1
    result2 = maxSlidingWindow(nums2, k2)
    print(f"\n输入数组: {nums2}")
    print(f"窗口大小: {k2}")
    print(f"最大值序列: {result2}")
    # 预期输出: [1]

    # 测试用例3：空数组
    nums3 = []
    k3 = 1
    result3 = maxSlidingWindow(nums3, k3)
    print(f"\n输入数组: {nums3}")
    print(f"窗口大小: {k3}")
    print(f"最大值序列: {result3}")
    # 预期输出: []

    # 测试用例4：k为0
    nums4 = [1, 2, 3]
    k4 = 0
    result4 = maxSlidingWindow(nums4, k4)
    print(f"\n输入数组: {nums4}")
    print(f"窗口大小: {k4}")
    print(f"最大值序列: {result4}")
    # 预期输出: []
```

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

文件: Code09_PermutationInString.cpp

```
==============================================
/*
 * 字符串的排列问题解决方案
 *
 * 问题描述：
 * 给你两个字符串 s1 和 s2 ，写一个函数来判断 s2 是否包含 s1 的排列。
 * 如果是，返回 true ；否则，返回 false 。
 * 换句话说，s1 的排列之一是 s2 的 子串 。
 *
 * 解题思路：
 * 使用滑动窗口算法判断s2是否包含s1的排列：
 * 1. 统计s1中各字符的频次
 * 2. 维护一个长度为s1.length()的滑动窗口遍历s2
 * 3. 当窗口内字符频次与s1完全匹配时，说明找到了s1的一个排列
```

```
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - n 为 s2 的长度
 * 空间复杂度: O(1) - 只需要 26 个字母的统计数组
 *
 * 是否最优解: 是
 *
 * 相关题目链接：
 * LeetCode 567. 字符串的排列
 * https://leetcode.cn/problems/permutation-in-string/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 字符串的排列
 *    https://www.nowcoder.com/practice/fe6b651b66ae47d7acce78ffdd9a96c7
 * 2. LintCode 1259. 字符串的排列
 *    https://www.lintcode.com/problem/1259/
 * 3. HackerRank - Permutation in String
 *    https://www.hackerrank.com/challenges/permutation-in-string/problem
 * 4. CodeChef - PERMSTR - Permutation in String
 *    https://www.codechef.com/problems/PERMSTR
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、s1 长度大于 s2 等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要 C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code09_PermutationInString.cpp -o Code09_PermutationInString
```

```
 */

// 算法实现（需要C++标准库支持）
/*
#include <string>
#include <vector>
#include <iostream>
using namespace std;

bool checkInclusion(string s1, string s2) {
    // 异常情况处理：如果s1长度大于s2，不可能包含s1的排列
    if (s1.length() > s2.length()) {
        return false;
    }

    // 统计s1中各字符的频次
    vector<int> count(26, 0);
    for (char c : s1) {
        count[c - 'a']++;
    }

    int windowLen = s1.length();

    // 滑动窗口遍历s2
    // l为左指针，r为右指针，diff为当前窗口与s1的字符差异计数
    for (int l = 0, r = 0, diff = s1.length(); r < s2.length(); r++) {
        // 右边界字符进入窗口
        // 如果该字符在s1中存在（count[s2[r]-'a'] > 0），则减少差异计数
        if (count[s2[r] - 'a']-- > 0) {
            // 如果是有效字符，减少差异计数
            diff--;
        }

        // 当窗口大小超过s1长度时，左边界字符离开窗口
        // 此时需要移除窗口左边的字符
        if (r >= windowLen) {
            // 如果移除的字符在s1中存在（count[s2[l]-'a'] >= 0），则增加差异计数
            if (count[s2[l] - 'a']++ >= 0) {
                // 如果是有效字符，增加差异计数
                diff++;
            }
            // 移动左指针
            l++;
```

```
        }

        // 如果没有差异，说明当前窗口内的字符与 s1 的字符完全匹配，即找到了 s1 的一个排列
        if (diff == 0) {
            return true;
        }
    }

    return false;
}

// 测试用例
int main() {
    // 测试用例 1
    string s1_1 = "ab";
    string s2_1 = "eidbaooo";
    bool result1 = checkInclusion(s1_1, s2_1);
    cout << "s1: " << s1_1 << ", s2: " << s2_1 << endl;
    cout << "结果: " << (result1 ? "true" : "false") << endl;
    // 预期输出: true

    // 测试用例 2
    string s1_2 = "ab";
    string s2_2 = "eidboaoo";
    bool result2 = checkInclusion(s1_2, s2_2);
    cout << "\ns1: " << s1_2 << ", s2: " << s2_2 << endl;
    cout << "结果: " << (result2 ? "true" : "false") << endl;
    // 预期输出: false

    return 0;
}
*/

// 算法核心逻辑说明（伪代码形式）:
/*
function checkInclusion(s1, s2):
    if length(s1) > length(s2):
        return false

    // 统计 s1 中各字符的频次
    count = array of size 26, initialized to 0
    for each character c in s1:
        count[c - 'a']++
```

```
    windowLen = length(s1)

    // 滑动窗口遍历 s2
    for r from 0 to length(s2)-1:
        // 右边界字符进入窗口
        if count[s2[r] - 'a']-- > 0:
            diff--

        // 当窗口大小超过 s1 长度时，左边界字符离开窗口
        if r >= windowLen:
            if count[s2[l] - 'a']++ >= 0:
                diff++
            l++

        // 如果没有差异，说明找到了匹配的排列
        if diff == 0:
            return true

    return false
*/

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

文件: Code09_PermutationInString.java
==========================================
package class049;

import java.util.*;

/**
 * 字符串的排列问题解决方案
 *
 * 问题描述：
 * 给你两个字符串 s1 和 s2 ，写一个函数来判断 s2 是否包含 s1 的排列。
 * 如果是，返回 true ；否则，返回 false 。
 * 换句话说，s1 的排列之一是 s2 的 子串 。
 *
 * 解题思路：
 * 使用滑动窗口算法判断 s2 是否包含 s1 的排列：
 * 1. 统计 s1 中各字符的频次
 * 2. 维护一个长度为 s1.length()的滑动窗口遍历 s2
 * 3. 当窗口内字符频次与 s1 完全匹配时，说明找到了 s1 的一个排列
```

```
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - n 为 s2 的长度
 * 空间复杂度: O(1) - 只需要 26 个字母的统计数组
 *
 * 是否最优解: 是
 *
 * 相关题目链接：
 * LeetCode 567. 字符串的排列
 * https://leetcode.cn/problems/permutation-in-string/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 字符串的排列
 *    https://www.nowcoder.com/practice/fe6b651b66ae47d7acce78ffdd9a96c7
 * 2. LintCode 1259. 字符串的排列
 *    https://www.lintcode.com/problem/1259/
 * 3. HackerRank - Permutation in String
 *    https://www.hackerrank.com/challenges/permutation-in-string/problem
 * 4. CodeChef - PERMSTR - Permutation in String
 *    https://www.codechef.com/problems/PERMSTR
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、s1 长度大于 s2 等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code09_PermutationInString {

    /**
```

```
 * 判断 s2 是否包含 s1 的排列
 *
 * @param s1 目标字符串
 * @param s2 源字符串
 * @return 如果 s2 包含 s1 的排列则返回 true，否则返回 false
 */
public static boolean checkInclusion(String s1, String s2) {
    // 异常情况处理：如果 s1 长度大于 s2，不可能包含 s1 的排列
    if (s1.length() > s2.length()) {
        return false;
    }

    // 统计 s1 中各字符的频次
    int[] count = new int[26];
    for (char c : s1.toCharArray()) {
        count[c - 'a']++;
    }

    int windowLen = s1.length();
    char[] s = s2.toCharArray();

    // 滑动窗口遍历 s2
    // l 为左指针，r 为右指针，diff 为当前窗口与 s1 的字符差异计数
    for (int l = 0, r = 0, diff = s1.length(); r < s2.length(); r++) {
        // 右边界字符进入窗口
        // 如果该字符在 s1 中存在（count[s[r]-'a'] > 0），则减少差异计数
        if (count[s[r] - 'a']-- > 0) {
            // 如果是有效字符，减少差异计数
            diff--;
        }

        // 当窗口大小超过 s1 长度时，左边界字符离开窗口
        // 此时需要移除窗口左边的字符
        if (r >= windowLen) {
            // 如果移除的字符在 s1 中存在（count[s[l]-'a'] >= 0），则增加差异计数
            if (count[s[l] - 'a']++ >= 0) {
                // 如果是有效字符，增加差异计数
                diff++;
            }
            // 移动左指针
            l++;
        }
```

```
            // 如果没有差异，说明当前窗口内的字符与 s1 的字符完全匹配，即找到了 s1 的一个排列
            if (diff == 0) {
                return true;
            }
        }

        return false;
    }

    /**
     * 测试用例
     */
    public static void main(String[] args) {
        // 测试用例 1
        String s1_1 = "ab";
        String s2_1 = "eidbaooo";
        boolean result1 = checkInclusion(s1_1, s2_1);
        System.out.println("s1: " + s1_1 + ", s2: " + s2_1);
        System.out.println("结果: " + result1);
        // 预期输出: true

        // 测试用例 2
        String s1_2 = "ab";
        String s2_2 = "eidboaoo";
        boolean result2 = checkInclusion(s1_2, s2_2);
        System.out.println("\ns1: " + s1_2 + ", s2: " + s2_2);
        System.out.println("结果: " + result2);
        // 预期输出: false

        // 测试用例 3：s1 长度大于 s2
        String s1_3 = "abc";
        String s2_3 = "ab";
        boolean result3 = checkInclusion(s1_3, s2_3);
        System.out.println("\ns1: " + s1_3 + ", s2: " + s2_3);
        System.out.println("结果: " + result3);
        // 预期输出: false

        // 测试用例 4：完全匹配
        String s1_4 = "abc";
        String s2_4 = "baxyzabc";
        boolean result4 = checkInclusion(s1_4, s2_4);
        System.out.println("\ns1: " + s1_4 + ", s2: " + s2_4);
        System.out.println("结果: " + result4);
```

```
        // 预期输出: true
    }
}
```

==========

文件: Code09_PermutationInString.py

==========

```
# -*- coding: utf-8 -*-
"""
字符串的排列问题解决方案

问题描述：
给你两个字符串 s1 和 s2 ，写一个函数来判断 s2 是否包含 s1 的排列。
如果是，返回 true ；否则，返回 false 。
换句话说，s1 的排列之一是 s2 的 子串 。

解题思路：
使用滑动窗口算法判断 s2 是否包含 s1 的排列：
1. 统计 s1 中各字符的频次
2. 维护一个长度为 s1.length() 的滑动窗口遍历 s2
3. 当窗口内字符频次与 s1 完全匹配时，说明找到了 s1 的一个排列

算法复杂度分析：
时间复杂度: O(n) - n 为 s2 的长度
空间复杂度: O(1) - 只需要 26 个字母的统计数组

是否最优解: 是

相关题目链接：
LeetCode 567. 字符串的排列
https://leetcode.cn/problems/permutation-in-string/

其他平台类似题目：
1. 牛客网 - 字符串的排列
   https://www.nowcoder.com/practice/fe6b651b66ae47d7acce78ffdd9a96c7
2. LintCode 1259. 字符串的排列
   https://www.lintcode.com/problem/1259/
3. HackerRank - Permutation in String
   https://www.hackerrank.com/challenges/permutation-in-string/problem
4. CodeChef - PERMSTR - Permutation in String
   https://www.codechef.com/problems/PERMSTR
5. AtCoder - ABC146 D - Enough Array
```

   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电 OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空字符串、s1 长度大于 s2 等边界情况
2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
def checkInclusion(s1, s2):
    """
    判断 s2 是否包含 s1 的排列

    Args:
        s1 (str): 目标字符串
        s2 (str): 源字符串

    Returns:
        bool: 如果 s2 包含 s1 的排列则返回 True，否则返回 False

    Examples:
        >>> checkInclusion("ab", "eidbaooo")
        True
        >>> checkInclusion("ab", "eidboaoo")
        False
    """
    # 异常情况处理：如果 s1 长度大于 s2，不可能包含 s1 的排列
    if len(s1) > len(s2):
        return False

    # 统计 s1 中各字符的频次
```

```
    count = [0] * 26
    for c in s1:
        count[ord(c) - ord('a')] += 1

    window_len = len(s1)

    # 滑动窗口遍历 s2
    l = 0
    diff = len(s1)  # 差异字符数，初始值为 s1 的长度

    for r in range(len(s2)):
        # 右边界字符进入窗口
        # 如果该字符在 s1 中存在（count[ord(s2[r]) - ord('a')] > 0），则减少差异计数
        if count[ord(s2[r]) - ord('a')] > 0:
            # 如果是有效字符，减少差异计数
            diff -= 1
        # 更新字符计数
        count[ord(s2[r]) - ord('a')] -= 1

        # 当窗口大小超过 s1 长度时，左边界字符离开窗口
        # 此时需要移除窗口左边的字符
        if r >= window_len:
            # 如果移除的字符在 s1 中存在（count[ord(s2[l]) - ord('a')] >= 0），则增加差异计数
            if count[ord(s2[l]) - ord('a')] >= 0:
                # 如果是有效字符，增加差异计数
                diff += 1
            # 更新字符计数
            count[ord(s2[l]) - ord('a')] += 1
            # 移动左指针
            l += 1

        # 如果没有差异，说明当前窗口内的字符与 s1 的字符完全匹配，即找到了 s1 的一个排列
        if diff == 0:
            return True

    return False


# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    s1_1 = "ab"
    s2_1 = "eidbaooo"
```

```
    result1 = checkInclusion(s1_1, s2_1)
    print(f"s1: {s1_1}, s2: {s2_1}")
    print(f"结果: {result1}")
    # 预期输出: True

    # 测试用例 2
    s1_2 = "ab"
    s2_2 = "eidboaoo"
    result2 = checkInclusion(s1_2, s2_2)
    print(f"\ns1: {s1_2}, s2: {s2_2}")
    print(f"结果: {result2}")
    # 预期输出: False

    # 测试用例 3：s1 长度大于 s2
    s1_3 = "abc"
    s2_3 = "ab"
    result3 = checkInclusion(s1_3, s2_3)
    print(f"\ns1: {s1_3}, s2: {s2_3}")
    print(f"结果: {result3}")
    # 预期输出: False

    # 测试用例 4：完全匹配
    s1_4 = "abc"
    s2_4 = "baxyzabc"
    result4 = checkInclusion(s1_4, s2_4)
    print(f"\ns1: {s1_4}, s2: {s2_4}")
    print(f"结果: {result4}")
    # 预期输出: True

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

文件: Code10_FindAllAnagrams.cpp
==========================================
/*
 * 找到字符串中所有字母异位词问题解决方案
 *
 * 问题描述：
 * 给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。
 * 异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。
 *
 * 解题思路：
 * 使用滑动窗口算法找到 s 中所有 p 的异位词：
 * 1. 统计 p 中各字符的频次
```

```
 * 2. 维护一个长度为 p.length()的滑动窗口遍历 s
 * 3. 当窗口内字符频次与 p 完全匹配时，说明找到了一个异位词
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - n 为 s 的长度
 * 空间复杂度: O(1) - 只需要 26 个字母的统计数组
 *
 * 是否最优解: 是
 *
 * 相关题目链接：
 * LeetCode 438. 找到字符串中所有字母异位词
 * https://leetcode.cn/problems/find-all-anagrams-in-a-string/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 找到字符串中所有字母异位词
 *    https://www.nowcoder.com/practice/432531b6fc7b483096e5f9170c862a49
 * 2. LintCode 647. 回文子串
 *    https://www.lintcode.com/problem/647/
 * 3. HackerRank - Find All Anagrams in a String
 *    https://www.hackerrank.com/challenges/find-all-anagrams-in-a-string/problem
 * 4. CodeChef - ANAGRAMS - Anagrams
 *    https://www.codechef.com/problems/ANAGRAMS
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、s 长度小于 p 等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
```

```
 * 此代码需要C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code10_FindAllAnagrams.cpp -o Code10_FindAllAnagrams
 */

// 算法实现（需要C++标准库支持）
/*
#include <string>
#include <vector>
#include <iostream>
using namespace std;

vector<int> findAnagrams(string s, string p) {
    // 初始化结果列表
    vector<int> result;

    // 异常情况处理：如果s长度小于p，不可能包含p的异位词
    if (s.length() < p.length()) {
        return result;
    }

    // 统计p中各字符的频次
    vector<int> count(26, 0);
    for (char c : p) {
        count[c - 'a']++;
    }

    int windowLen = p.length();

    // 滑动窗口遍历s
    // l为左指针，r为右指针，diff为当前窗口与p的字符差异计数
    for (int l = 0, r = 0, diff = p.length(); r < s.length(); r++) {
        // 右边界字符进入窗口
        // 如果该字符在p中存在（count[s[r]-'a'] > 0），则减少差异计数
        if (count[s[r] - 'a']-- > 0) {
            // 如果是有效字符，减少差异计数
            diff--;
        }

        // 当窗口大小超过p长度时，左边界字符离开窗口
        // 此时需要移除窗口左边的字符
        if (r >= windowLen) {
            // 如果移除的字符在p中存在（count[s[l]-'a'] >= 0），则增加差异计数
            if (count[s[l] - 'a']++ >= 0) {
```

```
                // 如果是有效字符，增加差异计数
                diff++;
            }
            // 移动左指针
            l++;
        }

        // 如果没有差异，说明当前窗口内的字符与 p 的字符完全匹配，即找到了一个异位词
        if (diff == 0) {
            result.push_back(l);
        }
    }

    return result;
}

// 测试用例
int main() {
    // 测试用例 1
    string s1 = "cbaebabacd";
    string p1 = "abc";
    vector<int> result1 = findAnagrams(s1, p1);
    cout << "s: " << s1 << ", p: " << p1 << endl;
    cout << "异位词起始索引: ";
    for (int idx : result1) cout << idx << " ";
    cout << "\n 预期输出: 0 6" << endl;

    // 测试用例 2
    string s2 = "abab";
    string p2 = "ab";
    vector<int> result2 = findAnagrams(s2, p2);
    cout << "\ns: " << s2 << ", p: " << p2 << endl;
    cout << "异位词起始索引: ";
    for (int idx : result2) cout << idx << " ";
    cout << "\n 预期输出: 0 1 2" << endl;

    return 0;
}
*/

// 算法核心逻辑说明（伪代码形式）:
/*
function findAnagrams(s, p):
```

```
    result = empty list
    if length(s) < length(p):
        return result

    // 统计 p 中各字符的频次
    count = array of size 26, initialized to 0
    for each character c in p:
        count[c - 'a']++

    windowLen = length(p)

    // 滑动窗口遍历 s
    for r from 0 to length(s)-1:
        // 右边界字符进入窗口
        if count[s[r] - 'a']-- > 0:
            diff--

        // 当窗口大小超过 p 长度时，左边界字符离开窗口
        if r >= windowLen:
            if count[s[l] - 'a']++ >= 0:
                diff++
            l++

        // 如果没有差异，说明找到了匹配的异位词
        if diff == 0:
            result.add(l)

    return result
*/

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

文件: Code10_FindAllAnagrams.java
==========================================
package class049;

import java.util.*;

/**
 * 找到字符串中所有字母异位词问题解决方案
 *
 * 问题描述：
 * 给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。
```

```
 * 异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。
 *
 * 解题思路：
 * 使用滑动窗口算法找到 s 中所有 p 的异位词：
 * 1. 统计 p 中各字符的频次
 * 2. 维护一个长度为 p.length()的滑动窗口遍历 s
 * 3. 当窗口内字符频次与 p 完全匹配时，说明找到了一个异位词
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - n 为 s 的长度
 * 空间复杂度: O(1) - 只需要 26 个字母的统计数组
 *
 * 是否最优解: 是
 *
 * 相关题目链接：
 * LeetCode 438. 找到字符串中所有字母异位词
 * https://leetcode.cn/problems/find-all-anagrams-in-a-string/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 找到字符串中所有字母异位词
 *    https://www.nowcoder.com/practice/432531b6fc7b483096e5f9170c862a49
 * 2. LintCode 647. 回文子串
 *    https://www.lintcode.com/problem/647/
 * 3. HackerRank - Find All Anagrams in a String
 *    https://www.hackerrank.com/challenges/find-all-anagrams-in-a-string/problem
 * 4. CodeChef - ANAGRAMS - Anagrams
 *    https://www.codechef.com/problems/ANAGRAMS
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
```

```
 * 1. 异常处理：处理空字符串、s 长度小于 p 等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code10_FindAllAnagrams {

    /**
     * 找到 s 中所有 p 的异位词的起始索引
     *
     * @param s 源字符串
     * @param p 目标字符串
     * @return 所有异位词的起始索引列表
     */
    public static List<Integer> findAnagrams(String s, String p) {
        // 初始化结果列表
        List<Integer> result = new ArrayList<>();

        // 异常情况处理：如果 s 长度小于 p，不可能包含 p 的异位词
        if (s.length() < p.length()) {
            return result;
        }

        // 统计 p 中各字符的频次
        int[] count = new int[26];
        for (char c : p.toCharArray()) {
            count[c - 'a']++;
        }

        int windowLen = p.length();
        char[] str = s.toCharArray();

        // 滑动窗口遍历 s
        // l 为左指针，r 为右指针，diff 为当前窗口与 p 的字符差异计数
        for (int l = 0, r = 0, diff = p.length(); r < s.length(); r++) {
            // 右边界字符进入窗口
            // 如果该字符在 p 中存在（count[str[r]-'a'] > 0），则减少差异计数
            if (count[str[r] - 'a']-- > 0) {
                // 如果是有效字符，减少差异计数
                diff--;
            }

            // 当窗口大小超过 p 长度时，左边界字符离开窗口
            // 此时需要移除窗口左边的字符
```

```
        if (r >= windowLen) {
            // 如果移除的字符在 p 中存在（count[str[l]-'a'] >= 0），则增加差异计数
            if (count[str[l] - 'a']++ >= 0) {
                // 如果是有效字符，增加差异计数
                diff++;
            }
            // 移动左指针
            l++;
        }

        // 如果没有差异，说明当前窗口内的字符与 p 的字符完全匹配，即找到了一个异位词
        if (diff == 0) {
            result.add(l);
        }
    }

    return result;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    String s1 = "cbaebabacd";
    String p1 = "abc";
    List<Integer> result1 = findAnagrams(s1, p1);
    System.out.println("s: " + s1 + ", p: " + p1);
    System.out.println("异位词起始索引: " + result1);
    // 预期输出: [0, 6]

    // 测试用例 2
    String s2 = "abab";
    String p2 = "ab";
    List<Integer> result2 = findAnagrams(s2, p2);
    System.out.println("\ns: " + s2 + ", p: " + p2);
    System.out.println("异位词起始索引: " + result2);
    // 预期输出: [0, 1, 2]

    // 测试用例 3：s 长度小于 p
    String s3 = "ab";
    String p3 = "abc";
    List<Integer> result3 = findAnagrams(s3, p3);
```

```
        System.out.println("\ns: " + s3 + ", p: " + p3);
        System.out.println("异位词起始索引: " + result3);
        // 预期输出: []

        // 测试用例 4：完全匹配
        String s4 = "abcabc";
        String p4 = "abc";
        List<Integer> result4 = findAnagrams(s4, p4);
        System.out.println("\ns: " + s4 + ", p: " + p4);
        System.out.println("异位词起始索引: " + result4);
        // 预期输出: [0, 1, 2, 3]
    }
}
```

==========

文件: Code10_FindAllAnagrams.py

==========

```
# -*- coding: utf-8 -*-
"""
找到字符串中所有字母异位词问题解决方案

问题描述：
给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。
异位词 指由相同字母重排列形成的字符串（包括相同的字符串）。

解题思路：
使用滑动窗口算法找到 s 中所有 p 的异位词：
1. 统计 p 中各字符的频次
2. 维护一个长度为 p.length() 的滑动窗口遍历 s
3. 当窗口内字符频次与 p 完全匹配时，说明找到了一个异位词

算法复杂度分析：
时间复杂度: O(n) - n 为 s 的长度
空间复杂度: O(1) - 只需要 26 个字母的统计数组

是否最优解: 是

相关题目链接：
LeetCode 438. 找到字符串中所有字母异位词
https://leetcode.cn/problems/find-all-anagrams-in-a-string/

其他平台类似题目：
```

1. 牛客网 - 找到字符串中所有字母异位词
   https://www.nowcoder.com/practice/432531b6fc7b483096e5f9170c862a49
2. LintCode 647. 回文子串
   https://www.lintcode.com/problem/647/
3. HackerRank - Find All Anagrams in a String
   https://www.hackerrank.com/challenges/find-all-anagrams-in-a-string/problem
4. CodeChef - ANAGRAMS - Anagrams
   https://www.codechef.com/problems/ANAGRAMS
5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电 OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空字符串、s 长度小于 p 等边界情况
2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
def findAnagrams(s, p):
    """
    找到 s 中所有 p 的异位词的起始索引

    Args:
        s (str): 源字符串
        p (str): 目标字符串

    Returns:
        List[int]: 所有异位词的起始索引列表

    Examples:
        >>> findAnagrams("cbaebabacd", "abc")
```

```
    [0, 6]
    >>> findAnagrams("abab", "ab")
    [0, 1, 2]
"""
# 初始化结果列表
result = []

# 异常情况处理：如果 s 长度小于 p，不可能包含 p 的异位词
if len(s) < len(p):
    return result

# 统计 p 中各字符的频次
count = [0] * 26
for c in p:
    count[ord(c) - ord('a')] += 1

window_len = len(p)

# 滑动窗口遍历 s
l = 0
diff = len(p)  # 差异字符数，初始值为 p 的长度

for r in range(len(s)):
    # 右边界字符进入窗口
    # 如果该字符在 p 中存在（count[ord(s[r]) - ord('a')] > 0），则减少差异计数
    if count[ord(s[r]) - ord('a')] > 0:
        # 如果是有效字符，减少差异计数
        diff -= 1
    # 更新字符计数
    count[ord(s[r]) - ord('a')] -= 1

    # 当窗口大小超过 p 长度时，左边界字符离开窗口
    # 此时需要移除窗口左边的字符
    if r >= window_len:
        # 如果移除的字符在 p 中存在（count[ord(s[l]) - ord('a')] >= 0），则增加差异计数
        if count[ord(s[l]) - ord('a')] >= 0:
            # 如果是有效字符，增加差异计数
            diff += 1
        # 更新字符计数
        count[ord(s[l]) - ord('a')] += 1
        # 移动左指针
        l += 1
```

```
        # 如果没有差异，说明当前窗口内的字符与 p 的字符完全匹配，即找到了一个异位词
        if diff == 0:
            result.append(l)

    return result


# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    s1 = "cbaebabacd"
    p1 = "abc"
    result1 = findAnagrams(s1, p1)
    print(f"s: {s1}, p: {p1}")
    print(f"异位词起始索引: {result1}")
    # 预期输出: [0, 6]

    # 测试用例 2
    s2 = "abab"
    p2 = "ab"
    result2 = findAnagrams(s2, p2)
    print(f"\ns: {s2}, p: {p2}")
    print(f"异位词起始索引: {result2}")
    # 预期输出: [0, 1, 2]

    # 测试用例 3：s 长度小于 p
    s3 = "ab"
    p3 = "abc"
    result3 = findAnagrams(s3, p3)
    print(f"\ns: {s3}, p: {p3}")
    print(f"异位词起始索引: {result3}")
    # 预期输出: []

    # 测试用例 4：完全匹配
    s4 = "abcabc"
    p4 = "abc"
    result4 = findAnagrams(s4, p4)
    print(f"\ns: {s4}, p: {p4}")
    print(f"异位词起始索引: {result4}")
    # 预期输出: [0, 1, 2, 3]

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

文件: Code11_MaxConsecutiveOnes.cpp

```
==========================================
/*
 * 最大连续1的个数 III 问题解决方案
 *
 * 问题描述：
 * 给定一个二进制数组 nums 和一个整数 k，
 * 如果可以翻转最多 k 个 0，则返回数组中连续 1 的最大个数。
 *
 * 解题思路：
 * 使用滑动窗口算法找到最长的连续1序列（最多可以翻转k个0）：
 * 1. 维护一个滑动窗口，窗口内最多包含k个0
 * 2. 右指针不断扩展窗口
 * 3. 当窗口内0的个数超过k时，左指针右移缩小窗口
 * 4. 记录窗口的最大长度
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - n为数组长度，每个元素最多被访问两次
 * 空间复杂度: O(1) - 只使用常数额外空间
 *
 * 是否最优解: 是
 *
 * 相关题目链接：
 * LeetCode 1004. 最大连续1的个数 III
 * https://leetcode.cn/problems/max-consecutive-ones-iii/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 最大连续1的个数
 *    https://www.nowcoder.com/practice/4665256cab4418c8287c912b78d5a1e7
 * 2. LintCode 883. 最大连续1的个数 III
 *    https://www.lintcode.com/problem/883/
 * 3. HackerRank - Max Consecutive Ones
 *    https://www.hackerrank.com/challenges/max-consecutive-ones/problem
 * 4. CodeChef - CON1S - Consecutive Ones
 *    https://www.codechef.com/problems/CON1S
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
```

```
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要 C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code11_MaxConsecutiveOnes.cpp -o Code11_MaxConsecutiveOnes
 */

// 算法实现（需要 C++标准库支持）
/*
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int longestOnes(vector<int>& nums, int k) {
    // 记录最大连续 1 序列长度
    int maxLen = 0;

    // 滑动窗口遍历数组
    // l 为左指针，r 为右指针，zeros 为窗口内 0 的个数
    for (int l = 0, r = 0, zeros = 0; r < nums.size(); r++) {
        // 右边界元素进入窗口
        // 如果是 0，则增加窗口内 0 的计数
        if (nums[r] == 0) {
            zeros++;
        }

        // 当窗口内 0 的个数超过 k 时，需要缩小窗口
        while (zeros > k) {
            // 如果移除的元素是 0，则减少窗口内 0 的计数
            if (nums[l++] == 0) {
                zeros--;
            }
```

```
        }

        // 更新最大长度（当前窗口大小）
        maxLen = max(maxLen, r - l + 1);
    }

    return maxLen;
}

// 测试用例
int main() {
    // 测试用例 1
    vector<int> nums1 = {1,1,1,0,0,0,1,1,1,1,0};
    int k1 = 2;
    int result1 = longestOnes(nums1, k1);
    cout << "数组: ";
    for (int num : nums1) cout << num << " ";
    cout << ", k: " << k1 << endl;
    cout << "最大连续 1 的个数: " << result1 << endl;
    // 预期输出: 6

    // 测试用例 2
    vector<int> nums2 = {0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1};
    int k2 = 3;
    int result2 = longestOnes(nums2, k2);
    cout << "\n 数组: ";
    for (int num : nums2) cout << num << " ";
    cout << ", k: " << k2 << endl;
    cout << "最大连续 1 的个数: " << result2 << endl;
    // 预期输出: 10

    return 0;
}
*/

// 算法核心逻辑说明（伪代码形式）:
/*
function longestOnes(nums, k):
    maxLen = 0

    // 滑动窗口遍历数组
    for r from 0 to length(nums)-1:
        // 右边界元素进入窗口
```

```
        if nums[r] == 0:
            zeros++

        // 当窗口内 0 的个数超过 k 时，左指针右移缩小窗口
        while zeros > k:
            if nums[l++] == 0:
                zeros--

        // 更新最大长度
        maxLen = max(maxLen, r - l + 1)

    return maxLen
*/

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

文件: Code11_MaxConsecutiveOnes.java
==========================================
package class049;

import java.util.*;

/**
 * 最大连续 1 的个数 III 问题解决方案
 *
 * 问题描述：
 * 给定一个二进制数组 nums 和一个整数 k，
 * 如果可以翻转最多 k 个 0，则返回数组中连续 1 的最大个数。
 *
 * 解题思路：
 * 使用滑动窗口算法找到最长的连续 1 序列（最多可以翻转 k 个 0）：
 * 1. 维护一个滑动窗口，窗口内最多包含 k 个 0
 * 2. 右指针不断扩展窗口
 * 3. 当窗口内 0 的个数超过 k 时，左指针右移缩小窗口
 * 4. 记录窗口的最大长度
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - n 为数组长度，每个元素最多被访问两次
 * 空间复杂度: O(1) - 只使用常数额外空间
 *
 * 是否最优解: 是
 *
 * 相关题目链接：
```

```
 * LeetCode 1004. 最大连续1的个数 III
 * https://leetcode.cn/problems/max-consecutive-ones-iii/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 最大连续1的个数
 *    https://www.nowcoder.com/practice/4665256cab4418c8287c912b78d5a1e7
 * 2. LintCode 883. 最大连续1的个数 III
 *    https://www.lintcode.com/problem/883/
 * 3. HackerRank - Max Consecutive Ones
 *    https://www.hackerrank.com/challenges/max-consecutive-ones/problem
 * 4. CodeChef - CON1S - Consecutive Ones
 *    https://www.codechef.com/problems/CON1S
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k为负数等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code11_MaxConsecutiveOnes {

    /**
     * 计算最多翻转k个0后能得到的最长连续1序列长度
     *
     * @param nums 二进制数组
     * @param k    最多可以翻转的0的个数
     * @return 最长连续1序列长度
     */
    public static int longestOnes(int[] nums, int k) {
        // 记录最大连续1序列长度
```

```
    int maxLen = 0;

    // 滑动窗口遍历数组
    // l 为左指针，r 为右指针，zeros 为窗口内 0 的个数
    for (int l = 0, r = 0, zeros = 0; r < nums.length; r++) {
        // 右边界元素进入窗口
        // 如果是 0，则增加窗口内 0 的计数
        if (nums[r] == 0) {
            zeros++;
        }

        // 当窗口内 0 的个数超过 k 时，需要缩小窗口
        while (zeros > k) {
            // 如果移除的元素是 0，则减少窗口内 0 的计数
            if (nums[l++] == 0) {
                zeros--;
            }
        }

        // 更新最大长度（当前窗口大小）
        maxLen = Math.max(maxLen, r - l + 1);
    }

    return maxLen;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1,1,1,0,0,0,1,1,1,1,0};
    int k1 = 2;
    int result1 = longestOnes(nums1, k1);
    System.out.println("数组: " + Arrays.toString(nums1) + ", k: " + k1);
    System.out.println("最大连续 1 的个数: " + result1);
    // 预期输出: 6

    // 测试用例 2
    int[] nums2 = {0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1};
    int k2 = 3;
    int result2 = longestOnes(nums2, k2);
    System.out.println("\n 数组: " + Arrays.toString(nums2) + ", k: " + k2);
```

```
        System.out.println("最大连续1的个数: " + result2);
        // 预期输出: 10

        // 测试用例3：空数组
        int[] nums3 = {};
        int k3 = 1;
        int result3 = longestOnes(nums3, k3);
        System.out.println("\n数组: " + Arrays.toString(nums3) + ", k: " + k3);
        System.out.println("最大连续1的个数: " + result3);
        // 预期输出: 0

        // 测试用例4：k为0
        int[] nums4 = {1,1,1,0,0,0,1,1,1,1,0};
        int k4 = 0;
        int result4 = longestOnes(nums4, k4);
        System.out.println("\n数组: " + Arrays.toString(nums4) + ", k: " + k4);
        System.out.println("最大连续1的个数: " + result4);
        // 预期输出: 4
    }
}
```

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

文件: Code11_MaxConsecutiveOnes.py

```
==============================================
# -*- coding: utf-8 -*-
"""
```

最大连续1的个数 III 问题解决方案

问题描述：
给定一个二进制数组 nums 和一个整数 k，
如果可以翻转最多 k 个 0，则返回数组中连续 1 的最大个数。

解题思路：
使用滑动窗口算法找到最长的连续1序列（最多可以翻转k个0）：
1. 维护一个滑动窗口，窗口内最多包含k个0
2. 右指针不断扩展窗口
3. 当窗口内0的个数超过k时，左指针右移缩小窗口
4. 记录窗口的最大长度

算法复杂度分析：
时间复杂度: O(n) - n为数组长度，每个元素最多被访问两次
空间复杂度: O(1) - 只使用常数额外空间

是否最优解：是

相关题目链接：
LeetCode 1004. 最大连续1的个数 III
https://leetcode.cn/problems/max-consecutive-ones-iii/

其他平台类似题目：
1. 牛客网 - 最大连续1的个数
   https://www.nowcoder.com/practice/4665256cab4418c8287c912b78d5a1e7
2. LintCode 883. 最大连续1的个数 III
   https://www.lintcode.com/problem/883/
3. HackerRank - Max Consecutive Ones
   https://www.hackerrank.com/challenges/max-consecutive-ones/problem
4. CodeChef - CON1S - Consecutive Ones
   https://www.codechef.com/problems/CON1S
5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空数组、k为负数等边界情况
2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
def longestOnes(nums, k):
    """
    计算最多翻转k个0后能得到的最长连续1序列长度

    Args:
```

```
        nums (List[int]): 二进制数组
        k (int): 最多可以翻转的 0 的个数

    Returns:
        int: 最长连续 1 序列长度

    Examples:
        >>> longestOnes([1,1,1,0,0,0,1,1,1,1,0], 2)
        6
        >>> longestOnes([0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], 3)
        10
    """
    # 记录最大连续 1 序列长度
    max_len = 0

    # 滑动窗口遍历数组
    l = 0  # 左指针
    zeros = 0  # 窗口内 0 的个数

    for r in range(len(nums)):
        # 右边界元素进入窗口
        # 如果是 0，则增加窗口内 0 的计数
        if nums[r] == 0:
            zeros += 1

        # 当窗口内 0 的个数超过 k 时，需要缩小窗口
        while zeros > k:
            # 如果移除的元素是 0，则减少窗口内 0 的计数
            if nums[l] == 0:
                zeros -= 1
            # 移动左指针
            l += 1

        # 更新最大长度（当前窗口大小）
        max_len = max(max_len, r - l + 1)

    return max_len


# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    nums1 = [1,1,1,0,0,0,1,1,1,1,0]
```

```
    k1 = 2
    result1 = longestOnes(nums1, k1)
    print(f"数组: {nums1}, k: {k1}")
    print(f"最大连续1的个数: {result1}")
    # 预期输出: 6

    # 测试用例2
    nums2 = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1]
    k2 = 3
    result2 = longestOnes(nums2, k2)
    print(f"\n数组: {nums2}, k: {k2}")
    print(f"最大连续1的个数: {result2}")
    # 预期输出: 10

    # 测试用例3：空数组
    nums3 = []
    k3 = 1
    result3 = longestOnes(nums3, k3)
    print(f"\n数组: {nums3}, k: {k3}")
    print(f"最大连续1的个数: {result3}")
    # 预期输出: 0

    # 测试用例4：k为0
    nums4 = [1,1,1,0,0,0,1,1,1,1,0]
    k4 = 0
    result4 = longestOnes(nums4, k4)
    print(f"\n数组: {nums4}, k: {k4}")
    print(f"最大连续1的个数: {result4}")
    # 预期输出: 4
```

==========

文件: Code12_LongestSubarrayWithLimitedDifference.cpp

==========

```
/*
 * 绝对差不超过限制的最长连续子数组问题解决方案
 *
 * 问题描述：
 * 给你一个整数数组 nums ，和一个表示限制的整数 limit，
 * 请你返回最长连续子数组的长度，该子数组中的任意两个元素之间的绝对差必须小于或者等于 limit 。
 *
 * 解题思路：
 * 使用滑动窗口配合 map 来维护窗口内的最大值和最小值：
```

```
 * 1. 右指针不断扩展窗口，将元素加入 map
 * 2. 当窗口内最大值与最小值的差超过 limit 时，收缩左指针
 * 3. map 可以在 O(log k) 时间内维护窗口元素的有序性，其中 k 是窗口大小
 * 4. map 的 begin()->first 和 rbegin()->first 分别获取最小值和最大值
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n * log n) - 每个元素最多入队和出队一次，map 操作需要 O(log n)
 * 空间复杂度: O(n) - map 最多存储 n 个元素
 *
 * 是否最优解: 是，这是处理该问题的较优解法之一，还可以用单调队列优化到 O(n)
 *
 * 相关题目链接：
 * LeetCode 1438. 绝对差不超过限制的最长连续子数组
 * https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 绝对差不超过限制的最长连续子数组
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 1438. 绝对差不超过限制的最长连续子数组
 *    https://www.lintcode.com/problem/1438/
 * 3. HackerRank - Longest Subarray with Limited Difference
 *    https://www.hackerrank.com/challenges/longest-subarray-with-limited-difference/problem
 * 4. CodeChef - SUBARR - Subarray with Limited Difference
 *    https://www.codechef.com/problems/SUBARR
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
 https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组等边界情况
 * 2. 性能优化：使用 map 维护窗口元素有序性，避免重复计算
```

```
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code12_LongestSubarrayWithLimitedDifference.cpp -o
Code12_LongestSubarrayWithLimitedDifference
 */

// 算法实现（需要C++标准库支持）
/*
#include <vector>
#include <map>
#include <algorithm>
#include <iostream>
using namespace std;

// 经典滑动窗口问题，使用 map 维护窗口内的最大值和最小值
class Solution {
public:
    // 计算绝对差不超过限制的最长连续子数组长度
    int longestSubarray(vector<int>& nums, int limit) {
        // 异常情况处理
        if (nums.empty()) {
            return 0;
        }

        // map 维护窗口内元素及其出现次数，保持有序
        // key为元素值，value为该元素在窗口中的出现次数
        map<int, int> mp;
        int left = 0;  // 滑动窗口左指针
        int result = 0;  // 记录最长子数组长度

        // 右指针扩展窗口
        for (int right = 0; right < nums.size(); right++) {
            // 将右指针元素加入 map
            // 如果元素已存在，计数加1；否则插入新元素，计数为1
            mp[nums[right]]++;

            // 当窗口内最大值与最小值的差超过 limit 时，需要收缩左指针
            // map的rbegin()->first获取最大值，begin()->first获取最小值
            while (mp.rbegin()->first - mp.begin()->first > limit) {
                // 减少左指针元素的计数
                mp[nums[left]]--;
```

```
                // 如果计数为 0，从 map 中移除该元素
                if (mp[nums[left]] == 0) {
                    mp.erase(nums[left]);
                }
                // 移动左指针
                left++;
            }

            // 更新最长子数组长度（当前窗口大小）
            result = max(result, right - left + 1);
        }

        return result;
    }
};

// 测试用例
int main() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {8, 2, 4, 7};
    int limit1 = 4;
    int result1 = solution.longestSubarray(nums1, limit1);
    cout << "输入数组: ";
    for (int num : nums1) cout << num << " ";
    cout << "\n限制值: " << limit1;
    cout << "\n最长子数组长度: " << result1 << endl;
    // 预期输出: 2 ([2,4] 或 [4,7])

    // 测试用例 2
    vector<int> nums2 = {10, 1, 2, 4, 7, 2};
    int limit2 = 5;
    int result2 = solution.longestSubarray(nums2, limit2);
    cout << "\n输入数组: ";
    for (int num : nums2) cout << num << " ";
    cout << "\n限制值: " << limit2;
    cout << "\n最长子数组长度: " << result2 << endl;
    // 预期输出: 4 ([2,4,7,2])

    return 0;
}
*/
```

```
// 算法核心逻辑说明（伪代码形式）：
/*
class Solution {
public:
    int longestSubarray(vector<int>& nums, int limit) {
        if (nums.empty()) {
            return 0;
        }

        map<int, int> mp;  // 维护窗口内元素及其出现次数
        int left = 0;
        int result = 0;

        for (int right = 0; right < nums.size(); right++) {
            // 将右指针元素加入 map
            mp[nums[right]]++;

            // 当窗口内最大值与最小值的差超过 limit 时，收缩左指针
            while (mp.rbegin()->first - mp.begin()->first > limit) {
                mp[nums[left]]--;
                if (mp[nums[left]] == 0) {
                    mp.erase(nums[left]);
                }
                left++;
            }

            // 更新最长子数组长度
            result = max(result, right - left + 1);
        }

        return result;
    }
};
*/

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

文件: Code12_LongestSubarrayWithLimitedDifference.java
==========================================
package class049;

import java.util.*;
```

```
/**
 * 绝对差不超过限制的最长连续子数组问题解决方案
 *
 * 问题描述：
 * 给你一个整数数组 nums ，和一个表示限制的整数 limit，
 * 请你返回最长连续子数组的长度，该子数组中的任意两个元素之间的绝对差必须小于或者等于 limit 。
 *
 * 解题思路：
 * 使用滑动窗口配合 TreeMap 来维护窗口内的最大值和最小值：
 * 1. 右指针不断扩展窗口，将元素加入 TreeMap
 * 2. 当窗口内最大值与最小值的差超过 limit 时，收缩左指针
 * 3. TreeMap 可以在 O(log k) 时间内维护窗口元素的有序性，其中 k 是窗口大小
 * 4. TreeMap 的 firstKey() 和 lastKey() 分别获取最小值和最大值
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n * log n) - 每个元素最多入队和出队一次，TreeMap 操作需要 O(log n)
 * 空间复杂度: O(n) - TreeMap 最多存储 n 个元素
 *
 * 是否最优解: 是，这是处理该问题的较优解法之一，还可以用单调队列优化到 O(n)
 *
 * 相关题目链接：
 * LeetCode 1438. 绝对差不超过限制的最长连续子数组
 * https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 绝对差不超过限制的最长连续子数组
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 1438. 绝对差不超过限制的最长连续子数组
 *    https://www.lintcode.com/problem/1438/
 * 3. HackerRank - Longest Subarray with Limited Difference
 *    https://www.hackerrank.com/challenges/longest-subarray-with-limited-difference/problem
 * 4. CodeChef - SUBARR - Subarray with Limited Difference
 *    https://www.codechef.com/problems/SUBARR
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
```

```
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组等边界情况
 * 2. 性能优化：使用 TreeMap 维护窗口元素有序性，避免重复计算
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code12_LongestSubarrayWithLimitedDifference {

    /**
     * 计算绝对差不超过限制的最长连续子数组长度
     *
     * @param nums   输入的整数数组
     * @param limit  限制值，子数组中任意两个元素的绝对差不能超过此值
     * @return 最长连续子数组的长度
     */
    public static int longestSubarray(int[] nums, int limit) {
        // 异常情况处理
        if (nums == null || nums.length == 0) {
            return 0;
        }

        // TreeMap 维护窗口内元素及其出现次数，保持有序
        // key 为元素值，value 为该元素在窗口中的出现次数
        TreeMap<Integer, Integer> map = new TreeMap<>();
        int left = 0;  // 滑动窗口左指针
        int result = 0;  // 记录最长子数组长度

        // 右指针扩展窗口
        for (int right = 0; right < nums.length; right++) {
            // 将右指针元素加入 TreeMap
            // getOrDefault 方法获取元素当前出现次数，如果不存在则返回 0
            map.put(nums[right], map.getOrDefault(nums[right], 0) + 1);

            // 当窗口内最大值与最小值的差超过 limit 时，需要收缩左指针
            // TreeMap 的 lastKey()获取最大值，firstKey()获取最小值
            while (map.lastKey() - map.firstKey() > limit) {
                // 减少左指针元素的计数
```

```
                map.put(nums[left], map.get(nums[left]) - 1);
                // 如果计数为 0，从 TreeMap 中移除该元素
                if (map.get(nums[left]) == 0) {
                    map.remove(nums[left]);
                }
                // 移动左指针
                left++;
            }

            // 更新最长子数组长度（当前窗口大小）
            result = Math.max(result, right - left + 1);
        }

        return result;
    }

    /**
     * 测试用例
     */
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {8, 2, 4, 7};
        int limit1 = 4;
        int result1 = longestSubarray(nums1, limit1);
        System.out.println("输入数组: " + Arrays.toString(nums1));
        System.out.println("限制值: " + limit1);
        System.out.println("最长子数组长度: " + result1);
        // 预期输出: 2 ([2,4] 或 [4,7])

        // 测试用例 2
        int[] nums2 = {10, 1, 2, 4, 7, 2};
        int limit2 = 5;
        int result2 = longestSubarray(nums2, limit2);
        System.out.println("\n 输入数组: " + Arrays.toString(nums2));
        System.out.println("限制值: " + limit2);
        System.out.println("最长子数组长度: " + result2);
        // 预期输出: 4 ([2,4,7,2])

        // 测试用例 3
        int[] nums3 = {4, 2, 2, 2, 4, 4, 2, 2};
        int limit3 = 0;
        int result3 = longestSubarray(nums3, limit3);
        System.out.println("\n 输入数组: " + Arrays.toString(nums3));
```

```
        System.out.println("限制值: " + limit3);
        System.out.println("最长子数组长度: " + result3);
        // 预期输出: 3 ([2,2,2])

        // 测试用例 4：空数组
        int[] nums4 = {};
        int limit4 = 1;
        int result4 = longestSubarray(nums4, limit4);
        System.out.println("\n 输入数组: " + Arrays.toString(nums4));
        System.out.println("限制值: " + limit4);
        System.out.println("最长子数组长度: " + result4);
        // 预期输出: 0
    }
}
```

==========

文件: Code12_LongestSubarrayWithLimitedDifference.py
==========

```
# -*- coding: utf-8 -*-
"""
绝对差不超过限制的最长连续子数组问题解决方案

问题描述：
给你一个整数数组 nums ，和一个表示限制的整数 limit，
请你返回最长连续子数组的长度，该子数组中的任意两个元素之间的绝对差必须小于或者等于 limit 。

解题思路：
使用滑动窗口配合两个堆来维护窗口内的最大值和最小值：
1. 右指针不断扩展窗口，将元素加入最大堆和最小堆
2. 当窗口内最大值与最小值的差超过 limit 时，收缩左指针
3. 使用延迟删除技术处理堆中已移出窗口的元素
4. 最大堆和最小堆的堆顶分别获取窗口的最大值和最小值

算法复杂度分析：
时间复杂度: O(n * log n) - 每个元素最多入队和出队一次，堆操作需要 O(log n)
空间复杂度: O(n) - 堆最多存储 n 个元素

是否最优解: 是，这是处理该问题的较优解法之一，还可以用单调队列优化到 O(n)

相关题目链接：
LeetCode 1438. 绝对差不超过限制的最长连续子数组
https://leetcode.cn/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-
```

to-limit/

其他平台类似题目：
1. 牛客网 - 绝对差不超过限制的最长连续子数组
   https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
2. LintCode 1438. 绝对差不超过限制的最长连续子数组
   https://www.lintcode.com/problem/1438/
3. HackerRank - Longest Subarray with Limited Difference
   https://www.hackerrank.com/challenges/longest-subarray-with-limited-difference/problem
4. CodeChef - SUBARR - Subarray with Limited Difference
   https://www.codechef.com/problems/SUBARR
5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空数组等边界情况
2. 性能优化：使用堆维护窗口元素最值，避免重复计算
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
from collections import deque
import heapq


def longest_subarray(nums, limit):
    """
    计算绝对差不超过限制的最长连续子数组长度

    Args:
        nums (List[int]): 输入的整数数组
        limit (int): 限制值，子数组中任意两个元素的绝对差不能超过此值
```

```
Returns:
    int: 最长连续子数组的长度

Examples:
    >>> longest_subarray([8, 2, 4, 7], 4)
    2
    >>> longest_subarray([10, 1, 2, 4, 7, 2], 5)
    4
"""
# 异常情况处理
if not nums:
    return 0

# 最大堆和最小堆，存储 (值, 索引) 元组
# Python 的 heapq 是最小堆，存储负值来模拟最大堆
max_heap = []  # 最大堆（存储负值）
min_heap = []  # 最小堆

left = 0  # 滑动窗口左指针
result = 0  # 记录最长子数组长度

# 右指针扩展窗口
for right in range(len(nums)):
    # 将右指针元素加入两个堆
    # 存储负值模拟最大堆
    heapq.heappush(max_heap, (-nums[right], right))
    heapq.heappush(min_heap, (nums[right], right))

    # 当窗口内最大值与最小值的差超过 limit 时，需要收缩左指针
    # -max_heap[0][0] 获取最大值，min_heap[0][0] 获取最小值
    while -max_heap[0][0] - min_heap[0][0] > limit:
        # 移除堆顶已过期的元素（索引小于 left 的元素）
        # 这是延迟删除技术，避免在堆中直接删除元素
        while max_heap and max_heap[0][1] <= left:
            heapq.heappop(max_heap)
        while min_heap and min_heap[0][1] <= left:
            heapq.heappop(min_heap)

        # 移动左指针
        left += 1

    # 更新最长子数组长度（当前窗口大小）
```

```
        result = max(result, right - left + 1)

    return result


# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    nums1 = [8, 2, 4, 7]
    limit1 = 4
    result1 = longest_subarray(nums1, limit1)
    print("输入数组:", nums1)
    print("限制值:", limit1)
    print("最长子数组长度:", result1)
    # 预期输出: 2 ([2,4] 或 [4,7])

    # 测试用例 2
    nums2 = [10, 1, 2, 4, 7, 2]
    limit2 = 5
    result2 = longest_subarray(nums2, limit2)
    print("\n输入数组:", nums2)
    print("限制值:", limit2)
    print("最长子数组长度:", result2)
    # 预期输出: 4 ([2,4,7,2])

    # 测试用例 3
    nums3 = [4, 2, 2, 2, 4, 4, 2, 2]
    limit3 = 0
    result3 = longest_subarray(nums3, limit3)
    print("\n输入数组:", nums3)
    print("限制值:", limit3)
    print("最长子数组长度:", result3)
    # 预期输出: 3 ([2,2,2])

    # 测试用例 4：空数组
    nums4 = []
    limit4 = 1
    result4 = longest_subarray(nums4, limit4)
    print("\n输入数组:", nums4)
    print("限制值:", limit4)
    print("最长子数组长度:", result4)
    # 预期输出: 0
```

==========

文件: Code13_GetEqualSubstringsWithinBudget.cpp
==========

```
/*
 * 尽可能使字符串相等问题解决方案
 *
 * 问题描述：
 * 给你两个长度相同的字符串，s 和 t。
 * 将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），
 * 也就是两个字符的 ASCII 码值的差的绝对值。
 * 用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，
 * 这也意味着字符串的转化可能是不完全的。
 * 如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。
 * 如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。
 *
 * 解题思路：
 * 使用滑动窗口来解决这个问题：
 * 1. 计算每个位置的转换成本：cost[i] = |s[i] - t[i]|
 * 2. 使用滑动窗口维护一个转换成本总和不超过 maxCost 的子数组
 * 3. 右指针不断扩展窗口，左指针在总成本超过 maxCost 时收缩
 * 4. 记录满足条件的最长窗口长度
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - 每个元素最多被访问两次
 * 空间复杂度: O(1) - 只使用常数额外空间
 *
 * 是否最优解: 是，这是该问题的最优解法
 *
 * 相关题目链接：
 * LeetCode 1208. 尽可能使字符串相等
 * https://leetcode.cn/problems/get-equal-substrings-within-budget/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 尽可能使字符串相等
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 1208. 尽可能使字符串相等
 *    https://www.lintcode.com/problem/1208/
 * 3. HackerRank - Get Equal Substrings Within Budget
 *    https://www.hackerrank.com/challenges/get-equal-substrings-within-budget/problem
 * 4. CodeChef - EQUALSTR - Equal Strings
 *    https://www.codechef.com/problems/EQUALSTR
 * 5. AtCoder - ABC146 D - Enough Array
```

```
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、长度不一致等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code13_GetEqualSubstringsWithinBudget.cpp -o Code13_GetEqualSubstringsWithinBudget
 */

// 算法实现（需要C++标准库支持）
/*
#include <iostream>
#include <string>
#include <algorithm>
#include <cmath>
using namespace std;

// 经典滑动窗口问题，计算字符串转换成本
class Solution {
public:
    // 计算在预算内可以转换的最大子字符串长度
    int equalSubstring(string s, string t, int maxCost) {
        // 异常情况处理
        if (s.empty() || t.empty() || s.length() != t.length()) {
            return 0;
        }
```

```
        int n = s.length();
        int left = 0;  // 滑动窗口左指针
        int currentCost = 0;  // 当前窗口内的总转换成本
        int maxLength = 0;  // 记录最大转换长度

        // 右指针扩展窗口
        for (int right = 0; right < n; right++) {
            // 计算当前位置的转换成本并加入窗口
            // 转换成本为两个字符 ASCII 码值差的绝对值
            currentCost += abs(s[right] - t[right]);

            // 当前窗口成本超过预算时，需要收缩左指针
            while (currentCost > maxCost) {
                // 移除左指针位置的转换成本
                currentCost -= abs(s[left] - t[left]);
                // 移动左指针
                left++;
            }

            // 更新最大长度（当前窗口大小）
            maxLength = max(maxLength, right - left + 1);
        }

        return maxLength;
    }
};

// 测试用例
int main() {
    Solution solution;

    // 测试用例 1
    string s1 = "abcd";
    string t1 = "bcdf";
    int maxCost1 = 3;
    int result1 = solution.equalSubstring(s1, t1, maxCost1);
    cout << "字符串 s: " << s1 << endl;
    cout << "字符串 t: " << t1 << endl;
    cout << "最大预算: " << maxCost1 << endl;
    cout << "最大转换长度: " << result1 << endl;
    // 预期输出: 3 ("abc" -> "bcd" 成本为 3)

    return 0;
```

```
}
*/

// 算法核心逻辑说明（伪代码形式）：
/*
class Solution {
public:
    int equalSubstring(string s, string t, int maxCost) {
        if (s.empty() || t.empty() || s.length() != t.length()) {
            return 0;
        }

        int n = s.length();
        int left = 0;
        int currentCost = 0;
        int maxLength = 0;

        for (int right = 0; right < n; right++) {
            currentCost += abs(s[right] - t[right]);

            while (currentCost > maxCost) {
                currentCost -= abs(s[left] - t[left]);
                left++;
            }

            maxLength = max(maxLength, right - left + 1);
        }

        return maxLength;
    }
};
*/

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

文件: Code13_GetEqualSubstringsWithinBudget.java
==========================================
package class049;

/**
 * 尽可能使字符串相等问题解决方案
 *
 * 问题描述：
```

* 给你两个长度相同的字符串，s 和 t。
* 将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），
* 也就是两个字符的 ASCII 码值的差的绝对值。
* 用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，
* 这也意味着字符串的转化可能是不完全的。
* 如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。
* 如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。
*
* 解题思路：
* 使用滑动窗口来解决这个问题：
* 1. 计算每个位置的转换成本：cost[i] = |s[i] - t[i]|
* 2. 使用滑动窗口维护一个转换成本总和不超过 maxCost 的子数组
* 3. 右指针不断扩展窗口，左指针在总成本超过 maxCost 时收缩
* 4. 记录满足条件的最长窗口长度
*
* 算法复杂度分析：
* 时间复杂度: O(n) - 每个元素最多被访问两次
* 空间复杂度: O(1) - 只使用常数额外空间
*
* 是否最优解: 是，这是该问题的最优解法
*
* 相关题目链接：
* LeetCode 1208. 尽可能使字符串相等
* https://leetcode.cn/problems/get-equal-substrings-within-budget/
*
* 其他平台类似题目：
* 1. 牛客网 - 尽可能使字符串相等
*   https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
* 2. LintCode 1208. 尽可能使字符串相等
*   https://www.lintcode.com/problem/1208/
* 3. HackerRank - Get Equal Substrings Within Budget
*   https://www.hackerrank.com/challenges/get-equal-substrings-within-budget/problem
* 4. CodeChef - EQUALSTR - Equal Strings
*   https://www.codechef.com/problems/EQUALSTR
* 5. AtCoder - ABC146 D - Enough Array
*   https://atcoder.jp/contests/abc146/tasks/abc146_d
* 6. 洛谷 P1886 滑动窗口
*   https://www.luogu.com.cn/problem/P1886
* 7. 杭电 OJ 4193 Sliding Window
*   http://acm.hdu.edu.cn/showproblem.php?pid=4193
* 8. POJ 2823 Sliding Window
*   http://poj.org/problem?id=2823
* 9. UVa OJ 11536 - Smallest Sub-Array

```
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、长度不一致等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code13_GetEqualSubstringsWithinBudget {

    /**
     * 计算在预算内可以转换的最大子字符串长度
     *
     * @param s        源字符串
     * @param t        目标字符串
     * @param maxCost  最大预算成本
     * @return 在预算内可以转换的最大子字符串长度
     */
    public static int equalSubstring(String s, String t, int maxCost) {
        // 异常情况处理
        if (s == null || t == null || s.length() != t.length()) {
            return 0;
        }

        int n = s.length();
        int left = 0;  // 滑动窗口左指针
        int currentCost = 0;  // 当前窗口内的总转换成本
        int maxLength = 0;  // 记录最大转换长度

        // 右指针扩展窗口
        for (int right = 0; right < n; right++) {
            // 计算当前位置的转换成本并加入窗口
            // 转换成本为两个字符 ASCII 码值差的绝对值
            currentCost += Math.abs(s.charAt(right) - t.charAt(right));

            // 当前窗口成本超过预算时，需要收缩左指针
            while (currentCost > maxCost) {
                // 移除左指针位置的转换成本
                currentCost -= Math.abs(s.charAt(left) - t.charAt(left));
                // 移动左指针
```

```
            left++;
        }

        // 更新最大长度（当前窗口大小）
        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    String s1 = "abcd";
    String t1 = "bcdf";
    int maxCost1 = 3;
    int result1 = equalSubstring(s1, t1, maxCost1);
    System.out.println("字符串 s: " + s1);
    System.out.println("字符串 t: " + t1);
    System.out.println("最大预算: " + maxCost1);
    System.out.println("最大转换长度: " + result1);
    // 预期输出: 3 ("abc" -> "bcd" 成本为 3)

    // 测试用例 2
    String s2 = "abcd";
    String t2 = "cdef";
    int maxCost2 = 3;
    int result2 = equalSubstring(s2, t2, maxCost2);
    System.out.println("\n 字符串 s: " + s2);
    System.out.println("字符串 t: " + t2);
    System.out.println("最大预算: " + maxCost2);
    System.out.println("最大转换长度: " + result2);
    // 预期输出: 1 ("a" -> "c" 成本为 2，"b" -> "d" 成本为 2，都超过预算)

    // 测试用例 3
    String s3 = "abcd";
    String t3 = "acde";
    int maxCost3 = 0;
    int result3 = equalSubstring(s3, t3, maxCost3);
    System.out.println("\n 字符串 s: " + s3);
    System.out.println("字符串 t: " + t3);
```

```
        System.out.println("最大预算: " + maxCost3);
        System.out.println("最大转换长度: " + result3);
        // 预期输出: 1 ("a" -> "a" 成本为 0)

        // 测试用例 4：空字符串
        String s4 = "";
        String t4 = "";
        int maxCost4 = 1;
        int result4 = equalSubstring(s4, t4, maxCost4);
        System.out.println("\n 字符串 s: " + s4);
        System.out.println("字符串 t: " + t4);
        System.out.println("最大预算: " + maxCost4);
        System.out.println("最大转换长度: " + result4);
        // 预期输出: 0
    }
}
```

==========

文件: Code13_GetEqualSubstringsWithinBudget.py

==========

```
# -*- coding: utf-8 -*-
"""
尽可能使字符串相等问题解决方案

问题描述：
给你两个长度相同的字符串，s 和 t。
将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），
也就是两个字符的 ASCII 码值的差的绝对值。
用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，
这也意味着字符串的转化可能是不完全的。
如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。
如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。

解题思路：
使用滑动窗口来解决这个问题：
1. 计算每个位置的转换成本：cost[i] = |s[i] - t[i]|
2. 使用滑动窗口维护一个转换成本总和不超过 maxCost 的子数组
3. 右指针不断扩展窗口，左指针在总成本超过 maxCost 时收缩
4. 记录满足条件的最长窗口长度

算法复杂度分析：
时间复杂度: O(n) - 每个元素最多被访问两次
```

空间复杂度: O(1) - 只使用常数额外空间

是否最优解: 是，这是该问题的最优解法

相关题目链接：
LeetCode 1208. 尽可能使字符串相等
https://leetcode.cn/problems/get-equal-substrings-within-budget/

其他平台类似题目：
1. 牛客网 - 尽可能使字符串相等
   https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
2. LintCode 1208. 尽可能使字符串相等
   https://www.lintcode.com/problem/1208/
3. HackerRank - Get Equal Substrings Within Budget
   https://www.hackerrank.com/challenges/get-equal-substrings-within-budget/problem
4. CodeChef - EQUALSTR - Equal Strings
   https://www.codechef.com/problems/EQUALSTR
5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电 OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空字符串、长度不一致等边界情况
2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
def equal_substring(s, t, max_cost):
    """
    计算在预算内可以转换的最大子字符串长度
```

```
    Args:
        s (str): 源字符串
        t (str): 目标字符串
        max_cost (int): 最大预算成本

    Returns:
        int: 在预算内可以转换的最大子字符串长度

    Examples:
        >>> equal_substring("abcd", "bcdf", 3)
        3
        >>> equal_substring("abcd", "cdef", 3)
        1
    """
    # 异常情况处理
    if not s or not t or len(s) != len(t):
        return 0

    n = len(s)
    left = 0  # 滑动窗口左指针
    current_cost = 0  # 当前窗口内的总转换成本
    max_length = 0  # 记录最大转换长度

    # 右指针扩展窗口
    for right in range(n):
        # 计算当前位置的转换成本并加入窗口
        # 转换成本为两个字符 ASCII 码值差的绝对值
        current_cost += abs(ord(s[right]) - ord(t[right]))

        # 当前窗口成本超过预算时，需要收缩左指针
        while current_cost > max_cost:
            # 移除左指针位置的转换成本
            current_cost -= abs(ord(s[left]) - ord(t[left]))
            # 移动左指针
            left += 1

        # 更新最大长度（当前窗口大小）
        max_length = max(max_length, right - left + 1)

    return max_length


# 测试用例
```

```
if __name__ == "__main__":
    # 测试用例 1
    s1 = "abcd"
    t1 = "bcdf"
    max_cost1 = 3
    result1 = equal_substring(s1, t1, max_cost1)
    print("字符串 s:", s1)
    print("字符串 t:", t1)
    print("最大预算:", max_cost1)
    print("最大转换长度:", result1)
    # 预期输出: 3 ("abc" -> "bcd" 成本为 3)

    # 测试用例 2
    s2 = "abcd"
    t2 = "cdef"
    max_cost2 = 3
    result2 = equal_substring(s2, t2, max_cost2)
    print("\n字符串 s:", s2)
    print("字符串 t:", t2)
    print("最大预算:", max_cost2)
    print("最大转换长度:", result2)
    # 预期输出: 1 ("a" -> "c" 成本为 2，"b" -> "d" 成本为 2，都超过预算)

    # 测试用例 3
    s3 = "abcd"
    t3 = "acde"
    max_cost3 = 0
    result3 = equal_substring(s3, t3, max_cost3)
    print("\n字符串 s:", s3)
    print("字符串 t:", t3)
    print("最大预算:", max_cost3)
    print("最大转换长度:", result3)
    # 预期输出: 1 ("a" -> "a" 成本为 0)

    # 测试用例 4：空字符串
    s4 = ""
    t4 = ""
    max_cost4 = 1
    result4 = equal_substring(s4, t4, max_cost4)
    print("\n字符串 s:", s4)
    print("字符串 t:", t4)
    print("最大预算:", max_cost4)
    print("最大转换长度:", result4)
```

```
    # 预期输出: 0

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

文件: Code14_SlidingWindowMinMax.cpp
==========================================
/*
 * 滑动窗口最大值和最小值问题解决方案
 *
 * 问题描述：
 * 现在有一堆数字共 N 个数字（N<=10^6），以及一个大小为 k 的窗口。
 * 现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。
 *
 * 解题思路：
 * 使用单调队列来解决滑动窗口的最值问题：
 * 1. 维护两个双端队列：
 *    - 一个单调递增队列用于维护窗口最小值
 *    - 一个单调递减队列用于维护窗口最大值
 * 2. 队列中存储数组元素的索引，便于判断元素是否在窗口范围内
 * 3. 当窗口形成后（i >= k-1），记录当前窗口的最值
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - 每个元素最多入队和出队一次
 * 空间复杂度: O(k) - 双端队列最多存储 k 个元素
 *
 * 是否最优解: 是，这是处理滑动窗口最值问题的最优解法
 *
 * 相关题目链接：
 * 1. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 2. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 3. LeetCode 239. 滑动窗口最大值
 *    https://leetcode.cn/problems/sliding-window-maximum/
 * 4. 牛客网 - 滑动窗口最大值
 *    https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
 * 5. LintCode 362. 滑动窗口的最大值
 *    https://www.lintcode.com/problem/362/
 * 6. HackerRank - Sliding Window Maximum
 *    https://www.hackerrank.com/challenges/sliding-window-maximum/problem
 * 7. CodeChef - MAXSWINDOW - Maximum in Sliding Window
 *    https://www.codechef.com/problems/MAXSWINDOW
 * 8. AtCoder - ABC146 D - Enough Array
```

```
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k为负数或0等边界情况
 * 2. 性能优化：使用单调队列避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code14_SlidingWindowMinMax.cpp -o Code14_SlidingWindowMinMax
 */

// 算法实现（需要C++标准库支持）
/*
#include <iostream>
#include <vector>
#include <deque>
#include <algorithm>
using namespace std;

// 经典单调队列问题，求滑动窗口内的最值
class Solution {
public:
    // 计算滑动窗口中的最大值和最小值
    vector<vector<int>> slidingWindowMinMax(vector<int>& nums, int k) {
        // 异常情况处理
        if (nums.empty() || k <= 0) {
            return {{}, {}};
        }

        int n = nums.size();
        // 结果数组，[0]存储最小值序列，[1]存储最大值序列
        vector<vector<int>> result(2, vector<int>(n - k + 1));
        // 单调递增队列，队首是当前窗口的最小值索引
        deque<int> minDeque;
        // 单调递减队列，队首是当前窗口的最大值索引
        deque<int> maxDeque;
```

```
        // 遍历数组中的每个元素
        for (int i = 0; i < n; i++) {
            // 移除队列中超出窗口范围的索引
            // 当前窗口范围是 [i-k+1, i]，所以队首索引小于 i-k+1 的元素已经不在窗口内
            while (!minDeque.empty() && minDeque.front() < i - k + 1) {
                minDeque.pop_front();
            }
            while (!maxDeque.empty() && maxDeque.front() < i - k + 1) {
                maxDeque.pop_front();
            }

            // 维护单调递增队列（用于最小值）
            // 移除所有大于等于当前元素的索引，保持队列单调递增
            while (!minDeque.empty() && nums[minDeque.back()] >= nums[i]) {
                minDeque.pop_back();
            }

            // 维护单调递减队列（用于最大值）
            // 移除所有小于等于当前元素的索引，保持队列单调递减
            while (!maxDeque.empty() && nums[maxDeque.back()] <= nums[i]) {
                maxDeque.pop_back();
            }

            // 将当前元素索引加入队列尾部
            minDeque.push_back(i);
            maxDeque.push_back(i);

            // 当窗口形成后（i >= k-1），记录当前窗口的最值
            // 窗口形成的条件是已经遍历了至少 k 个元素
            if (i >= k - 1) {
                result[0][i - k + 1] = nums[minDeque.front()];  // 最小值
                result[1][i - k + 1] = nums[maxDeque.front()];  // 最大值
            }
        }

        return result;
    }
};

// 测试用例
int main() {
    Solution solution;
```

```
    // 测试用例 1
    vector<int> nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    vector<vector<int>> result1 = solution.slidingWindowMinMax(nums1, k1);
    // 预期输出:
    // 最小值序列: -1 -3 -3 -3 3 3
    // 最大值序列: 3 3 5 5 6 7

    return 0;
}
*/

// 算法核心逻辑说明（伪代码形式）:
/*
class Solution {
public:
    vector<vector<int>> slidingWindowMinMax(vector<int>& nums, int k) {
        if (nums.empty() || k <= 0) {
            return {{}, {}};
        }

        int n = nums.size();
        vector<vector<int>> result(2, vector<int>(n - k + 1));
        deque<int> minDeque;  // 单调递增队列
        deque<int> maxDeque;  // 单调递减队列

        for (int i = 0; i < n; i++) {
            // 移除队列中超出窗口范围的索引
            while (!minDeque.empty() && minDeque.front() < i - k + 1) {
                minDeque.pop_front();
            }
            while (!maxDeque.empty() && maxDeque.front() < i - k + 1) {
                maxDeque.pop_front();
            }

            // 维护单调性
            while (!minDeque.empty() && nums[minDeque.back()] >= nums[i]) {
                minDeque.pop_back();
            }
            while (!maxDeque.empty() && nums[maxDeque.back()] <= nums[i]) {
                maxDeque.pop_back();
            }
```

```
            // 将当前元素索引加入队列
            minDeque.push_back(i);
            maxDeque.push_back(i);

            // 当窗口形成后，记录当前窗口的最值
            if (i >= k - 1) {
                result[0][i - k + 1] = nums[minDeque.front()];  // 最小值
                result[1][i - k + 1] = nums[maxDeque.front()];  // 最大值
            }
        }

        return result;
    }
};
*/

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

文件: Code14_SlidingWindowMinMax.java
==========================================
package class049;

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

/**
 * 滑动窗口最大值和最小值问题解决方案
 *
 * 问题描述：
 * 现在有一堆数字共N个数字（N<=10^6），以及一个大小为k的窗口。
 * 现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。
 *
 * 解题思路：
 * 使用单调队列来解决滑动窗口的最值问题：
 * 1. 维护两个双端队列：
 *    - 一个单调递增队列用于维护窗口最小值
 *    - 一个单调递减队列用于维护窗口最大值
 * 2. 队列中存储数组元素的索引，便于判断元素是否在窗口范围内
 * 3. 当窗口形成后（i >= k-1），记录当前窗口的最值
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - 每个元素最多入队和出队一次
```

```
 * 空间复杂度: O(k) - 双端队列最多存储 k 个元素
 *
 * 是否最优解: 是，这是处理滑动窗口最值问题的最优解法
 *
 * 相关题目链接：
 * 1. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 2. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 3. LeetCode 239. 滑动窗口最大值
 *    https://leetcode.cn/problems/sliding-window-maximum/
 * 4. 牛客网 - 滑动窗口最大值
 *    https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
 * 5. LintCode 362. 滑动窗口的最大值
 *    https://www.lintcode.com/problem/362/
 * 6. HackerRank - Sliding Window Maximum
 *    https://www.hackerrank.com/challenges/sliding-window-maximum/problem
 * 7. CodeChef - MAXSWINDOW - Maximum in Sliding Window
 *    https://www.codechef.com/problems/MAXSWINDOW
 * 8. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数或 0 等边界情况
 * 2. 性能优化：使用单调队列避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code14_SlidingWindowMinMax {

    /**
     * 计算滑动窗口中的最大值和最小值
     *
     * @param nums 输入的整数数组
     * @param k    滑动窗口的大小
     * @return 二维数组，[0]存储最小值序列，[1]存储最大值序列
     */
    public static int[][] slidingWindowMinMax(int[] nums, int k) {
```

```
// 异常情况处理
if (nums == null || nums.length == 0 || k <= 0) {
    return new int[2][0];
}

int n = nums.length;
// 结果数组，[0]存储最小值序列，[1]存储最大值序列
int[][] result = new int[2][n - k + 1];
// 单调递增队列，队首是当前窗口的最小值索引
Deque<Integer> minDeque = new ArrayDeque<>();
// 单调递减队列，队首是当前窗口的最大值索引
Deque<Integer> maxDeque = new ArrayDeque<>();

// 遍历数组中的每个元素
for (int i = 0; i < n; i++) {
    // 移除队列中超出窗口范围的索引
    // 当前窗口范围是 [i-k+1, i]，所以队首索引小于 i-k+1 的元素已经不在窗口内
    while (!minDeque.isEmpty() && minDeque.peekFirst() < i - k + 1) {
        minDeque.pollFirst();
    }
    while (!maxDeque.isEmpty() && maxDeque.peekFirst() < i - k + 1) {
        maxDeque.pollFirst();
    }

    // 维护单调递增队列（用于最小值）
    // 移除所有大于等于当前元素的索引，保持队列单调递增
    while (!minDeque.isEmpty() && nums[minDeque.peekLast()] >= nums[i]) {
        minDeque.pollLast();
    }

    // 维护单调递减队列（用于最大值）
    // 移除所有小于等于当前元素的索引，保持队列单调递减
    while (!maxDeque.isEmpty() && nums[maxDeque.peekLast()] <= nums[i]) {
        maxDeque.pollLast();
    }

    // 将当前元素索引加入队列尾部
    minDeque.offerLast(i);
    maxDeque.offerLast(i);

    // 当窗口形成后（i >= k-1），记录当前窗口的最值
    // 窗口形成的条件是已经遍历了至少 k 个元素
    if (i >= k - 1) {
```

```
            result[0][i - k + 1] = nums[minDeque.peekFirst()];  // 最小值
            result[1][i - k + 1] = nums[maxDeque.peekFirst()];  // 最大值
        }
    }

    return result;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    int[][] result1 = slidingWindowMinMax(nums1, k1);
    System.out.println("输入数组: " + Arrays.toString(nums1));
    System.out.println("窗口大小: " + k1);
    System.out.println("最小值序列: " + Arrays.toString(result1[0]));
    System.out.println("最大值序列: " + Arrays.toString(result1[1]));
    // 预期输出:
    // 最小值序列: [-1, -3, -3, -3, 3, 3]
    // 最大值序列: [3, 3, 5, 5, 6, 7]

    // 测试用例 2
    int[] nums2 = {1};
    int k2 = 1;
    int[][] result2 = slidingWindowMinMax(nums2, k2);
    System.out.println("\n 输入数组: " + Arrays.toString(nums2));
    System.out.println("窗口大小: " + k2);
    System.out.println("最小值序列: " + Arrays.toString(result2[0]));
    System.out.println("最大值序列: " + Arrays.toString(result2[1]));
    // 预期输出:
    // 最小值序列: [1]
    // 最大值序列: [1]

    // 测试用例 3
    int[] nums3 = {1, -1};
    int k3 = 1;
    int[][] result3 = slidingWindowMinMax(nums3, k3);
    System.out.println("\n 输入数组: " + Arrays.toString(nums3));
    System.out.println("窗口大小: " + k3);
    System.out.println("最小值序列: " + Arrays.toString(result3[0]));
```

```
        System.out.println("最大值序列: " + Arrays.toString(result3[1]));
        // 预期输出:
        // 最小值序列: [1, -1]
        // 最大值序列: [1, -1]

        // 测试用例 4：空数组
        int[] nums4 = {};
        int k4 = 1;
        int[][] result4 = slidingWindowMinMax(nums4, k4);
        System.out.println("\n 输入数组: " + Arrays.toString(nums4));
        System.out.println("窗口大小: " + k4);
        System.out.println("最小值序列长度: " + result4[0].length);
        System.out.println("最大值序列长度: " + result4[1].length);
        // 预期输出:
        // 最小值序列长度: 0
        // 最大值序列长度: 0
    }
}
```

==========

文件: Code14_SlidingWindowMinMax.py

==========

```
# -*- coding: utf-8 -*-
"""
滑动窗口最大值和最小值问题解决方案

问题描述：
现在有一堆数字共 N 个数字（N<=10^6），以及一个大小为 k 的窗口。
现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

解题思路：
使用单调队列来解决滑动窗口的最值问题：
1. 维护两个双端队列：
   - 一个单调递增队列用于维护窗口最小值
   - 一个单调递减队列用于维护窗口最大值
2. 队列中存储数组元素的索引，便于判断元素是否在窗口范围内
3. 当窗口形成后（i >= k-1），记录当前窗口的最值

算法复杂度分析：
时间复杂度: O(n) - 每个元素最多入队和出队一次
空间复杂度: O(k) - 双端队列最多存储 k 个元素
```

是否最优解：是，这是处理滑动窗口最值问题的最优解法

相关题目链接：
1. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
2. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
3. LeetCode 239. 滑动窗口最大值
   https://leetcode.cn/problems/sliding-window-maximum/
4. 牛客网 - 滑动窗口最大值
   https://www.nowcoder.com/practice/1624bc35a45c42c0bc17d17fa0cba788
5. LintCode 362. 滑动窗口的最大值
   https://www.lintcode.com/problem/362/
6. HackerRank - Sliding Window Maximum
   https://www.hackerrank.com/challenges/sliding-window-maximum/problem
7. CodeChef - MAXSWINDOW - Maximum in Sliding Window
   https://www.codechef.com/problems/MAXSWINDOW
8. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空数组、k 为负数或 0 等边界情况
2. 性能优化：使用单调队列避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
from collections import deque



def sliding_window_min_max(nums, k):
    """
    计算滑动窗口中的最大值和最小值

    Args:
        nums (List[int]): 输入的整数数组
        k (int): 滑动窗口的大小
```

```
Returns:
    List[List[int]]: 二维数组，[0]存储最小值序列，[1]存储最大值序列

Examples:
    >>> sliding_window_min_max([1, 3, -1, -3, 5, 3, 6, 7], 3)
    [[-1, -3, -3, -3, 3, 3], [3, 3, 5, 5, 6, 7]]
    >>> sliding_window_min_max([1], 1)
    [[1], [1]]
"""
# 异常情况处理
if not nums or k <= 0:
    return [[], []]

n = len(nums)
# 结果数组，[0]存储最小值序列，[1]存储最大值序列
result = [[], []]
# 单调递增队列，队首是当前窗口的最小值索引
min_deque = deque()
# 单调递减队列，队首是当前窗口的最大值索引
max_deque = deque()

# 遍历数组中的每个元素
for i in range(n):
    # 移除队列中超出窗口范围的索引
    # 当前窗口范围是 [i-k+1, i]，所以队首索引小于 i-k+1 的元素已经不在窗口内
    while min_deque and min_deque[0] < i - k + 1:
        min_deque.popleft()
    while max_deque and max_deque[0] < i - k + 1:
        max_deque.popleft()

    # 维护单调递增队列（用于最小值）
    # 移除所有大于等于当前元素的索引，保持队列单调递增
    while min_deque and nums[min_deque[-1]] >= nums[i]:
        min_deque.pop()

    # 维护单调递减队列（用于最大值）
    # 移除所有小于等于当前元素的索引，保持队列单调递减
    while max_deque and nums[max_deque[-1]] <= nums[i]:
        max_deque.pop()

    # 将当前元素索引加入队列尾部
    min_deque.append(i)
    max_deque.append(i)
```

```
        # 当窗口形成后（i >= k-1），记录当前窗口的最值
        # 窗口形成的条件是已经遍历了至少 k 个元素
        if i >= k - 1:
            result[0].append(nums[min_deque[0]])  # 最小值
            result[1].append(nums[max_deque[0]])  # 最大值

    return result


# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    nums1 = [1, 3, -1, -3, 5, 3, 6, 7]
    k1 = 3
    result1 = sliding_window_min_max(nums1, k1)
    print("输入数组:", nums1)
    print("窗口大小:", k1)
    print("最小值序列:", result1[0])
    print("最大值序列:", result1[1])
    # 预期输出:
    # 最小值序列: [-1, -3, -3, -3, 3, 3]
    # 最大值序列: [3, 3, 5, 5, 6, 7]

    # 测试用例 2
    nums2 = [1]
    k2 = 1
    result2 = sliding_window_min_max(nums2, k2)
    print("\n 输入数组:", nums2)
    print("窗口大小:", k2)
    print("最小值序列:", result2[0])
    print("最大值序列:", result2[1])
    # 预期输出:
    # 最小值序列: [1]
    # 最大值序列: [1]

    # 测试用例 3
    nums3 = [1, -1]
    k3 = 1
    result3 = sliding_window_min_max(nums3, k3)
    print("\n 输入数组:", nums3)
    print("窗口大小:", k3)
    print("最小值序列:", result3[0])
```

```
    print("最大值序列:", result3[1])
    # 预期输出:
    # 最小值序列: [1, -1]
    # 最大值序列: [1, -1]

    # 测试用例 4：空数组
    nums4 = []
    k4 = 1
    result4 = sliding_window_min_max(nums4, k4)
    print("\n 输入数组:", nums4)
    print("窗口大小:", k4)
    print("最小值序列长度:", len(result4[0]))
    print("最大值序列长度:", len(result4[1]))
    # 预期输出:
    # 最小值序列长度: 0
    # 最大值序列长度: 0
```

==========

文件: Code15_GrumpyBookstoreOwner.java

==========

```
package class049;

/**
 * 爱生气的书店老板问题解决方案
 *
 * 问题描述：
 * 有一个书店老板，他的书店开了 n 分钟。每分钟都有一些顾客进入这家商店。
 * 给定一个长度为 n 的整数数组 customers ，其中 customers[i] 是在第 i 分钟开始时进入商店的顾客数量，
 * 所有这些顾客在第 i 分钟结束后离开。
 * 在某些时候，书店老板会生气。如果书店老板在第 i 分钟生气，那么 grumpy[i] = 1，否则 grumpy[i] = 0。
 * 当书店老板生气时，那一分钟的顾客就会不满意，若老板不生气则顾客是满意的。
 * 书店老板知道一个秘密技巧，能抑制自己的情绪，可以让自己连续 minutes 分钟不生气，但却只能使用一次。
 * 请你返回这一天营业下来，最多有多少客户能够感到满意。
 *
 * 解题思路：
 * 使用滑动窗口来解决这个问题：
 * 1. 首先计算老板不使用技巧时的满意客户数（grumpy[i] = 0 时的 customers[i] 之和）
 * 2. 使用滑动窗口找出使用技巧能额外获得的最大满意客户数
 * 3. 窗口大小为 minutes，窗口内的 grumpy[i] = 1 的 customers[i] 就是额外获得的满意客户数
```

```
 * 4. 最终结果是基础满意客户数 + 使用技巧获得的最大额外客户数
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - 需要遍历数组两次
 * 空间复杂度: O(1) - 只使用常数额外空间
 *
 * 是否最优解: 是，这是该问题的最优解法
 *
 * 相关题目链接：
 * LeetCode 1052. 爱生气的书店老板
 * https://leetcode.cn/problems/grumpy-bookstore-owner/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 爱生气的书店老板
 *    https://www.nowcoder.com/practice/4d867d900e634e9fb9a0dae3480a374d
 * 2. LintCode 1052. 爱生气的书店老板
 *    https://www.lintcode.com/problem/1052/
 * 3. HackerRank - Grumpy Bookstore Owner
 *    https://www.hackerrank.com/challenges/grumpy-bookstore-owner/problem
 * 4. CodeChef - BOOKSTORE - Bookstore Owner
 *    https://www.codechef.com/problems/BOOKSTORE
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、长度不一致等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code15_GrumpyBookstoreOwner {
```

```
/**
 * 计算书店老板使用技巧后能让最多多少客户感到满意
 *
 * @param customers 顾客数组，customers[i]表示第 i 分钟进入的顾客数量
 * @param grumpy    生气数组，grumpy[i]=1 表示第 i 分钟老板生气，grumpy[i]=0 表示不生气
 * @param minutes   技巧持续时间，老板可以连续 minutes 分钟不生气
 * @return 最多能让多少客户感到满意
 */
public static int maxSatisfied(int[] customers, int[] grumpy, int minutes) {
    // 异常情况处理
    if (customers == null || grumpy == null || customers.length != grumpy.length) {
        return 0;
    }

    int n = customers.length;

    // 计算老板不使用技巧时的满意客户数
    // 当 grumpy[i] = 0 时，顾客是满意的
    int baseSatisfied = 0;
    for (int i = 0; i < n; i++) {
        if (grumpy[i] == 0) {
            baseSatisfied += customers[i];
        }
    }

    // 使用滑动窗口找出使用技巧能额外获得的最大满意客户数
    int extraSatisfied = 0;  // 记录使用技巧能获得的最大额外满意客户数
    int currentExtra = 0;    // 当前窗口内能额外获得的满意客户数

    // 初始化第一个窗口（前 minutes 分钟）
    for (int i = 0; i < minutes; i++) {
        // 只有当老板原本生气时（grumpy[i] = 1），使用技巧才能额外获得满意客户
        if (grumpy[i] == 1) {
            currentExtra += customers[i];
        }
    }
    extraSatisfied = currentExtra;

    // 滑动窗口，窗口大小为 minutes
    for (int i = minutes; i < n; i++) {
        // 添加新元素（窗口右边界）
        // 只有当老板原本生气时，使用技巧才能额外获得满意客户
        if (grumpy[i] == 1) {
```

```
            currentExtra += customers[i];
        }

        // 移除旧元素（窗口左边界）
        // 只有当移除的元素原本是生气状态时，才需要减去对应的客户数
        if (grumpy[i - minutes] == 1) {
            currentExtra -= customers[i - minutes];
        }

        // 更新最大额外满意客户数
        extraSatisfied = Math.max(extraSatisfied, currentExtra);
    }

    // 最终结果是基础满意客户数 + 使用技巧获得的最大额外客户数
    return baseSatisfied + extraSatisfied;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] customers1 = {1, 0, 1, 2, 1, 1, 7, 5};
    int[] grumpy1 = {0, 1, 0, 1, 0, 1, 0, 1};
    int minutes1 = 3;
    int result1 = maxSatisfied(customers1, grumpy1, minutes1);
    System.out.println("顾客数组: " + java.util.Arrays.toString(customers1));
    System.out.println("生气数组: " + java.util.Arrays.toString(grumpy1));
    System.out.println("技巧持续时间: " + minutes1);
    System.out.println("最大满意客户数: " + result1);
    // 预期输出: 16
    // 解释：老板在最后 3 分钟使用技巧，原本生气的第 6、8 分钟变为不生气
    // 基础满意客户：第 1、3、5、7 分钟的顾客(1+1+1+7=10)
    // 额外满意客户：第 6、8 分钟的顾客(1+5=6)
    // 总计：10+6=16

    // 测试用例 2
    int[] customers2 = {1};
    int[] grumpy2 = {0};
    int minutes2 = 1;
    int result2 = maxSatisfied(customers2, grumpy2, minutes2);
    System.out.println("\n 顾客数组: " + java.util.Arrays.toString(customers2));
    System.out.println("生气数组: " + java.util.Arrays.toString(grumpy2));
```

```
        System.out.println("技巧持续时间: " + minutes2);
        System.out.println("最大满意客户数: " + result2);
        // 预期输出: 1
        // 解释：老板本来就不生气，使用技巧没有额外效果

        // 测试用例 3：空数组
        int[] customers3 = {};
        int[] grumpy3 = {};
        int minutes3 = 1;
        int result3 = maxSatisfied(customers3, grumpy3, minutes3);
        System.out.println("\n 顾客数组: " + java.util.Arrays.toString(customers3));
        System.out.println("生气数组: " + java.util.Arrays.toString(grumpy3));
        System.out.println("技巧持续时间: " + minutes3);
        System.out.println("最大满意客户数: " + result3);
        // 预期输出: 0
    }
}
```

==========

文件: Code15_GrumpyBookstoreOwner.py

==========

```
# -*- coding: utf-8 -*-
"""
爱生气的书店老板问题解决方案

问题描述：
有一个书店老板，他的书店开了 n 分钟。每分钟都有一些顾客进入这家商店。
给定一个长度为 n 的整数数组 customers ，其中 customers[i] 是在第 i 分钟开始时进入商店的顾客数量，所有这些顾客在第 i 分钟结束后离开。
在某些时候，书店老板会生气。如果书店老板在第 i 分钟生气，那么 grumpy[i] = 1，否则 grumpy[i] = 0。当书店老板生气时，那一分钟的顾客就会不满意，若老板不生气则顾客是满意的。
书店老板知道一个秘密技巧，能抑制自己的情绪，可以让自己连续 minutes 分钟不生气，但却只能使用一次。
请你返回这一天营业下来，最多有多少客户能够感到满意。

解题思路：
使用滑动窗口来解决这个问题：
1. 首先计算老板不使用技巧时的满意客户数（grumpy[i] = 0 时的 customers[i] 之和）
2. 使用滑动窗口找出使用技巧能额外获得的最大满意客户数
3. 窗口大小为 minutes，窗口内的 grumpy[i] = 1 的 customers[i] 就是额外获得的满意客户数
4. 最终结果是基础满意客户数 + 使用技巧获得的最大额外客户数

算法复杂度分析：
```

时间复杂度: O(n) - 需要遍历数组两次
空间复杂度: O(1) - 只使用常数额外空间

是否最优解: 是，这是该问题的最优解法

相关题目链接：
LeetCode 1052. 爱生气的书店老板
https://leetcode.cn/problems/grumpy-bookstore-owner/

其他平台类似题目：
1. 牛客网 - 爱生气的书店老板
   https://www.nowcoder.com/practice/4d867d900e634e9fb9a0dae3480a374d
2. LintCode 1052. 爱生气的书店老板
   https://www.lintcode.com/problem/1052/
3. HackerRank - Grumpy Bookstore Owner
   https://www.hackerrank.com/challenges/grumpy-bookstore-owner/problem
4. CodeChef - BOOKSTORE - Bookstore Owner
   https://www.codechef.com/problems/BOOKSTORE
5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空数组、长度不一致等边界情况
2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
def max_satisfied(customers, grumpy, minutes):
    """
    计算书店老板使用技巧后能让最多多少客户感到满意
```

```
Args:
    customers (List[int]): 顾客数组，customers[i]表示第 i 分钟进入的顾客数量
    grumpy (List[int]): 生气数组，grumpy[i]=1 表示第 i 分钟老板生气，grumpy[i]=0 表示不生气
    minutes (int): 技巧持续时间，老板可以连续 minutes 分钟不生气

Returns:
    int: 最多能让多少客户感到满意

Examples:
    >>> max_satisfied([1, 0, 1, 2, 1, 1, 7, 5], [0, 1, 0, 1, 0, 1, 0, 1], 3)
    16
    >>> max_satisfied([1], [0], 1)
    1
"""
# 异常情况处理
if not customers or not grumpy or len(customers) != len(grumpy):
    return 0

n = len(customers)

# 计算老板不使用技巧时的满意客户数
# 当 grumpy[i] = 0 时，顾客是满意的
base_satisfied = 0
for i in range(n):
    if grumpy[i] == 0:
        base_satisfied += customers[i]

# 使用滑动窗口找出使用技巧能额外获得的最大满意客户数
extra_satisfied = 0  # 记录使用技巧能获得的最大额外满意客户数
current_extra = 0    # 当前窗口内能额外获得的满意客户数

# 初始化第一个窗口（前 minutes 分钟）
for i in range(minutes):
    # 只有当老板原本生气时（grumpy[i] = 1），使用技巧才能额外获得满意客户
    if grumpy[i] == 1:
        current_extra += customers[i]
extra_satisfied = current_extra

# 滑动窗口，窗口大小为 minutes
for i in range(minutes, n):
    # 添加新元素（窗口右边界）
    # 只有当老板原本生气时，使用技巧才能额外获得满意客户
```

```
        if grumpy[i] == 1:
            current_extra += customers[i]

        # 移除旧元素（窗口左边界）
        # 只有当移除的元素原本是生气状态时，才需要减去对应的客户数
        if grumpy[i - minutes] == 1:
            current_extra -= customers[i - minutes]

        # 更新最大额外满意客户数
        extra_satisfied = max(extra_satisfied, current_extra)

    # 最终结果是基础满意客户数 + 使用技巧获得的最大额外客户数
    return base_satisfied + extra_satisfied


# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    customers1 = [1, 0, 1, 2, 1, 1, 7, 5]
    grumpy1 = [0, 1, 0, 1, 0, 1, 0, 1]
    minutes1 = 3
    result1 = max_satisfied(customers1, grumpy1, minutes1)
    print("顾客数组:", customers1)
    print("生气数组:", grumpy1)
    print("技巧持续时间:", minutes1)
    print("最大满意客户数:", result1)
    # 预期输出: 16
    # 解释：老板在最后 3 分钟使用技巧，原本生气的第 6、8 分钟变为不生气
    # 基础满意客户：第 1、3、5、7 分钟的顾客(1+1+1+7=10)
    # 额外满意客户：第 6、8 分钟的顾客(1+5=6)
    # 总计：10+6=16

    # 测试用例 2
    customers2 = [1]
    grumpy2 = [0]
    minutes2 = 1
    result2 = max_satisfied(customers2, grumpy2, minutes2)
    print("\n 顾客数组:", customers2)
    print("生气数组:", grumpy2)
    print("技巧持续时间:", minutes2)
    print("最大满意客户数:", result2)
    # 预期输出: 1
    # 解释：老板本来就不生气，使用技巧没有额外效果
```

```
    # 测试用例 3：空数组
    customers3 = []
    grumpy3 = []
    minutes3 = 1
    result3 = max_satisfied(customers3, grumpy3, minutes3)
    print("\n 顾客数组:", customers3)
    print("生气数组:", grumpy3)
    print("技巧持续时间:", minutes3)
    print("最大满意客户数:", result3)
    # 预期输出: 0
```

==========

文件: Code16_MaximumPointsYouCanObtain.java

==========

```
package class049;

/**
 * 可获得的最大点数问题解决方案
 *
 * 问题描述：
 * 几张卡牌排成一行，每张卡牌都有一个对应的点数。点数由整数数组 cardPoints 给出。
 * 每次行动，你可以从行的开头或者末尾拿一张卡牌，最终你必须正好拿 k 张卡牌。
 * 你的点数就是你拿到手中的所有卡牌的点数之和。
 * 给你一个整数数组 cardPoints 和整数 k，请你返回可以获得的最大点数。
 *
 * 解题思路：
 * 这是一个转换思路的滑动窗口问题：
 * 1. 问题等价于：从数组中拿走 k 个数，使得拿走的数之和最大
 * 2. 由于只能从两端拿，所以剩下的 n-k 个数必然是连续的子数组
 * 3. 要使拿走的数之和最大，就要使剩下的连续子数组之和最小
 * 4. 使用滑动窗口找出长度为 n-k 的子数组的最小和
 * 5. 最大点数 = 总和 - 最小子数组和
 *
 * 算法复杂度分析：
 * 时间复杂度: O(n) - 需要遍历数组两次
 * 空间复杂度: O(1) - 只使用常数额外空间
 *
 * 是否最优解: 是，这是该问题的最优解法
 *
 * 相关题目链接：
 * LeetCode 1423. 可获得的最大点数
```

```
 * https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 可获得的最大点数
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 1423. 可获得的最大点数
 *    https://www.lintcode.com/problem/1423/
 * 3. HackerRank - Maximum Points You Can Obtain
 *    https://www.hackerrank.com/challenges/maximum-points-you-can-obtain/problem
 * 4. CodeChef - CARDGAME - Card Game
 *    https://www.codechef.com/problems/CARDGAME
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数或 0 等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code16_MaximumPointsYouCanObtain {

    /**
     * 计算可获得的最大点数
     *
     * @param cardPoints 卡牌点数数组
     * @param k          需要拿取的卡牌张数
     * @return 可获得的最大点数
     */
    public static int maxScore(int[] cardPoints, int k) {
        // 异常情况处理
        if (cardPoints == null || cardPoints.length == 0 || k <= 0) {
```

```
        return 0;
    }

    int n = cardPoints.length;
    // 如果 k 大于等于数组长度，拿走所有卡牌
    if (k >= n) {
        int sum = 0;
        for (int point : cardPoints) {
            sum += point;
        }
        return sum;
    }

    // 计算总和
    int totalSum = 0;
    for (int point : cardPoints) {
        totalSum += point;
    }

    // 滑动窗口大小为 n-k，找出子数组的最小和
    // 由于只能从两端拿牌，所以剩下的 n-k 张牌必然是连续的子数组
    int windowSize = n - k;
    int windowSum = 0;

    // 初始化第一个窗口（前 windowSize 个元素）
    for (int i = 0; i < windowSize; i++) {
        windowSum += cardPoints[i];
    }
    int minWindowSum = windowSum;

    // 滑动窗口，窗口大小为 windowSize
    for (int i = windowSize; i < n; i++) {
        // 添加新元素（窗口右边界），移除旧元素（窗口左边界）
        windowSum += cardPoints[i] - cardPoints[i - windowSize];
        // 更新最小子数组和
        minWindowSum = Math.min(minWindowSum, windowSum);
    }

    // 最大点数 = 总和 - 最小子数组和
    // 因为拿走 k 张牌的最大点数等于总点数减去剩下连续 n-k 张牌的最小点数
    return totalSum - minWindowSum;
}
```

```
/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] cardPoints1 = {1, 2, 3, 4, 5, 6, 1};
    int k1 = 3;
    int result1 = maxScore(cardPoints1, k1);
    System.out.println("卡牌点数: " + java.util.Arrays.toString(cardPoints1));
    System.out.println("拿取张数: " + k1);
    System.out.println("最大点数: " + result1);
    // 预期输出: 12 (拿取 1, 6, 5)
    // 解释：总点数 22，剩下连续 4 张牌的最小点数是 1+2+3+4=10，所以最大点数是 22-10=12

    // 测试用例 2
    int[] cardPoints2 = {2, 2, 2};
    int k2 = 2;
    int result2 = maxScore(cardPoints2, k2);
    System.out.println("\n卡牌点数: " + java.util.Arrays.toString(cardPoints2));
    System.out.println("拿取张数: " + k2);
    System.out.println("最大点数: " + result2);
    // 预期输出: 4
    // 解释：总点数 6，剩下连续 1 张牌的最小点数是 2，所以最大点数是 6-2=4

    // 测试用例 3
    int[] cardPoints3 = {9, 7, 7, 9, 7, 7, 9};
    int k3 = 7;
    int result3 = maxScore(cardPoints3, k3);
    System.out.println("\n卡牌点数: " + java.util.Arrays.toString(cardPoints3));
    System.out.println("拿取张数: " + k3);
    System.out.println("最大点数: " + result3);
    // 预期输出: 55 (拿取所有卡牌)
    // 解释：k 等于数组长度，拿取所有卡牌，点数为 55

    // 测试用例 4：空数组
    int[] cardPoints4 = {};
    int k4 = 1;
    int result4 = maxScore(cardPoints4, k4);
    System.out.println("\n卡牌点数: " + java.util.Arrays.toString(cardPoints4));
    System.out.println("拿取张数: " + k4);
    System.out.println("最大点数: " + result4);
    // 预期输出: 0
}
```

```
}

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

文件: Code16_MaximumPointsYouCanObtain.py
==========================================
# -*- coding: utf-8 -*-
"""
可获得的最大点数问题解决方案

问题描述：
几张卡牌排成一行，每张卡牌都有一个对应的点数。点数由整数数组 cardPoints 给出。
每次行动，你可以从行的开头或者末尾拿一张卡牌，最终你必须正好拿 k 张卡牌。
你的点数就是你拿到手中的所有卡牌的点数之和。
给你一个整数数组 cardPoints 和整数 k，请你返回可以获得的最大点数。

解题思路：
这是一个转换思路的滑动窗口问题：
1. 问题等价于：从数组中拿走 k 个数，使得拿走的数之和最大
2. 由于只能从两端拿，所以剩下的 n-k 个数必然是连续的子数组
3. 要使拿走的数之和最大，就要使剩下的连续子数组之和最小
4. 使用滑动窗口找出长度为 n-k 的子数组的最小和
5. 最大点数 = 总和 - 最小子数组和

算法复杂度分析：
时间复杂度: O(n) - 需要遍历数组两次
空间复杂度: O(1) - 只使用常数额外空间

是否最优解: 是，这是该问题的最优解法

相关题目链接：
LeetCode 1423. 可获得的最大点数
https://leetcode.cn/problems/maximum-points-you-can-obtain-from-cards/

其他平台类似题目：
1. 牛客网 - 可获得的最大点数
   https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
2. LintCode 1423. 可获得的最大点数
   https://www.lintcode.com/problem/1423/
3. HackerRank - Maximum Points You Can Obtain
   https://www.hackerrank.com/challenges/maximum-points-you-can-obtain/problem
4. CodeChef - CARDGAME - Card Game
   https://www.codechef.com/problems/CARDGAME
```

5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电 OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空数组、k 为负数或 0 等边界情况
2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
def max_score(card_points, k):
    """
    计算可获得的最大点数

    Args:
        card_points (List[int]): 卡牌点数数组
        k (int): 需要拿取的卡牌张数

    Returns:
        int: 可获得的最大点数

    Examples:
        >>> max_score([1, 2, 3, 4, 5, 6, 1], 3)
        12
        >>> max_score([2, 2, 2], 2)
        4
    """
    # 异常情况处理
    if not card_points or k <= 0:
        return 0
```

```
    n = len(card_points)
    # 如果 k 大于等于数组长度，拿走所有卡牌
    if k >= n:
        return sum(card_points)

    # 计算总和
    total_sum = sum(card_points)

    # 滑动窗口大小为 n-k，找出子数组的最小和
    # 由于只能从两端拿牌，所以剩下的 n-k 张牌必然是连续的子数组
    window_size = n - k
    window_sum = sum(card_points[:window_size])
    min_window_sum = window_sum

    # 滑动窗口，窗口大小为 window_size
    for i in range(window_size, n):
        # 添加新元素（窗口右边界），移除旧元素（窗口左边界）
        window_sum += card_points[i] - card_points[i - window_size]
        # 更新最小子数组和
        min_window_sum = min(min_window_sum, window_sum)

    # 最大点数 = 总和 - 最小子数组和
    # 因为拿走 k 张牌的最大点数等于总点数减去剩下连续 n-k 张牌的最小点数
    return total_sum - min_window_sum


# 测试用例
if __name__ == "__main__":
    # 测试用例 1
    card_points1 = [1, 2, 3, 4, 5, 6, 1]
    k1 = 3
    result1 = max_score(card_points1, k1)
    print("卡牌点数:", card_points1)
    print("拿取张数:", k1)
    print("最大点数:", result1)
    # 预期输出: 12 (拿取 1, 6, 5)
    # 解释：总点数 22，剩下连续 4 张牌的最小点数是 1+2+3+4=10，所以最大点数是 22-10=12

    # 测试用例 2
    card_points2 = [2, 2, 2]
    k2 = 2
    result2 = max_score(card_points2, k2)
    print("\n 卡牌点数:", card_points2)
```

```
    print("拿取张数:", k2)
    print("最大点数:", result2)
    # 预期输出: 4
    # 解释：总点数 6，剩下连续 1 张牌的最小点数是 2，所以最大点数是 6-2=4

    # 测试用例 3
    card_points3 = [9, 7, 7, 9, 7, 7, 9]
    k3 = 7
    result3 = max_score(card_points3, k3)
    print("\n卡牌点数:", card_points3)
    print("拿取张数:", k3)
    print("最大点数:", result3)
    # 预期输出: 55 (拿取所有卡牌)
    # 解释：k 等于数组长度，拿取所有卡牌，点数为 55

    # 测试用例 4：空数组
    card_points4 = []
    k4 = 1
    result4 = max_score(card_points4, k4)
    print("\n卡牌点数:", card_points4)
    print("拿取张数:", k4)
    print("最大点数:", result4)
    # 预期输出: 0
```

==========

文件: Code17_LongestRepeatingCharacterReplacement.cpp

==========

```
/*
 * 424. 替换后的最长重复字符问题解决方案
 *
 * 问题描述：
 * 给你一个字符串 s 和一个整数 k 。你可以选择字符串中的任一字符，并将其更改为任何其他大写英文字符。
 * 该操作最多可执行 k 次。
 * 在执行上述操作后，返回包含相同字母的最长子字符串的长度。
 *
 * 解题思路：
 * 使用滑动窗口维护一个窗口，窗口内最多有 k 个字符可以被替换成其他字符
 * 核心思想：窗口大小 - 窗口内出现次数最多的字符数量 <= k
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n)，其中 n 是字符串长度
```

```
 * 空间复杂度：O(1)，只需要26个字母的计数数组
 *
 * 是否最优解：是
 *
 * 相关题目链接：
 * LeetCode 424. 替换后的最长重复字符
 * https://leetcode.cn/problems/longest-repeating-character-replacement/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 替换后的最长重复字符
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 424. 替换后的最长重复字符
 *    https://www.lintcode.com/problem/424/
 * 3. HackerRank - Longest Repeating Character Replacement
 *    https://www.hackerrank.com/challenges/longest-repeating-character-replacement/problem
 * 4. CodeChef - REPLACE - Character Replacement
 *    https://www.codechef.com/problems/REPLACE
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code17_LongestRepeatingCharacterReplacement.cpp -o Code17_LongestRepeatingCharacterReplacement
 */
```

```
// 算法实现（需要 C++标准库支持）
/*
// 需要包含的头文件：
// #include <iostream>
// #include <string>
// #include <vector>
// #include <algorithm>
// using namespace std;

// 424. 替换后的最长重复字符
class Solution {
public:
    // 计算替换 k 个字符后能获得的最长重复字符子串长度
    int characterReplacement(string s, int k) {
        // 异常情况处理
        if (s.empty()) {
            return 0;
        }

        int n = s.length();
        // 记录窗口内各字符的出现次数（A-Z 共 26 个字母）
        vector<int> count(26, 0);
        int maxCount = 0; // 窗口内出现次数最多的字符数量
        int maxLength = 0; // 最长子串长度
        int left = 0; // 窗口左边界

        // 滑动窗口右边界
        for (int right = 0; right < n; right++) {
            // 当前右边界字符计数加 1
            count[s[right] - 'A']++;
            // 更新窗口内最大字符计数
            maxCount = max(maxCount, count[s[right] - 'A']);

            // 如果窗口大小减去最大字符计数大于 k，说明需要替换的字符超过 k 个
            // 需要收缩左边界
            // 核心条件：窗口大小 - 最多字符数量 > k 时，需要收缩窗口
            while (right - left + 1 - maxCount > k) {
                // 移除左边界字符
                count[s[left] - 'A']--;
                // 移动左边界
                left++;
                // 注意：这里不需要重新计算 maxCount，因为即使 maxCount 变小了
                // 也不会影响最终结果，我们只需要记录历史最大值
```

```
        }

        // 更新最大长度（当前窗口大小）
        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}

// 优化版本：使用历史最大值，避免每次重新计算 maxCount
int characterReplacementOptimized(string s, int k) {
    // 异常情况处理
    if (s.empty()) {
        return 0;
    }

    int n = s.length();
    // 记录窗口内各字符的出现次数
    vector<int> count(26, 0);
    int maxCount = 0; // 历史最大字符计数
    int maxLength = 0; // 最长子串长度
    int left = 0; // 窗口左边界

    // 滑动窗口遍历字符串
    for (int right = 0; right < n; right++) {
        // 右边界字符计数加 1
        count[s[right] - 'A']++;
        // 更新历史最大字符计数
        maxCount = max(maxCount, count[s[right] - 'A']);

        // 关键优化：使用历史最大值，即使窗口收缩后 maxCount 变小
        // 也不会影响结果，因为我们需要的是历史最大值
        // 当需要替换的字符数超过 k 时，收缩窗口
        if (right - left + 1 - maxCount > k) {
            // 移除左边界字符
            count[s[left] - 'A']--;
            // 移动左边界
            left++;
        }

        // 更新最大长度（当前窗口大小）
        maxLength = max(maxLength, right - left + 1);
    }
```

```
        return maxLength;
    }
};

// 测试函数
void testCharacterReplacement() {
    Solution solution;

    // 测试用例 1
    string s1 = "ABAB";
    int k1 = 2;
    int result1 = solution.characterReplacement(s1, k1);
    // 预期输出: 4

    // 测试用例 2
    string s2 = "AABABBA";
    int k2 = 1;
    int result2 = solution.characterReplacement(s2, k2);
    // 预期输出: 4
}

int main() {
    testCharacterReplacement();
    return 0;
}
*/

// 算法核心逻辑说明（伪代码形式）：
/*
class Solution {
public:
    int characterReplacement(string s, int k) {
        if (s.empty()) {
            return 0;
        }

        int n = s.length();
        vector<int> count(26, 0);  // 字符计数数组
        int maxCount = 0;  // 窗口内最大字符计数
        int maxLength = 0;  // 最长子串长度
        int left = 0;  // 窗口左边界
```

```
        for (int right = 0; right < n; right++) {
            // 右边界字符计数加1
            count[s[right] - 'A']++;
            // 更新最大字符计数
            maxCount = max(maxCount, count[s[right] - 'A']);

            // 当需要替换的字符数超过k时，收缩窗口
            while (right - left + 1 - maxCount > k) {
                count[s[left] - 'A']--;
                left++;
            }

            // 更新最大长度
            maxLength = max(maxLength, right - left + 1);
        }

        return maxLength;
    }
};
*/

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

文件: Code17_LongestRepeatingCharacterReplacement.java
==========================================
package class049;

import java.util.*;

/**
 * 424. 替换后的最长重复字符问题解决方案
 *
 * 问题描述：
 * 给你一个字符串 s 和一个整数 k 。你可以选择字符串中的任一字符，并将其更改为任何其他大写英文字符。
 * 该操作最多可执行 k 次。
 * 在执行上述操作后，返回包含相同字母的最长子字符串的长度。
 *
 * 解题思路：
 * 使用滑动窗口维护一个窗口，窗口内最多有k个字符可以被替换成其他字符
 * 核心思想：窗口大小 - 窗口内出现次数最多的字符数量 <= k
 *
 * 算法复杂度分析：
```

```
 * 时间复杂度：O(n)，其中 n 是字符串长度
 * 空间复杂度：O(1)，只需要 26 个字母的计数数组
 *
 * 是否最优解：是
 *
 * 相关题目链接：
 * LeetCode 424. 替换后的最长重复字符
 * https://leetcode.cn/problems/longest-repeating-character-replacement/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 替换后的最长重复字符
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 424. 替换后的最长重复字符
 *    https://www.lintcode.com/problem/424/
 * 3. HackerRank - Longest Repeating Character Replacement
 *    https://www.hackerrank.com/challenges/longest-repeating-character-replacement/problem
 * 4. CodeChef - REPLACE - Character Replacement
 *    https://www.codechef.com/problems/REPLACE
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空字符串、null 等边界情况
 * 2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code17_LongestRepeatingCharacterReplacement {

    /**
     * 计算替换 k 个字符后能获得的最长重复字符子串长度
     *
```

```
     * @param s 输入字符串，只包含大写英文字母
     * @param k 最多可以替换的字符次数
     * @return 最长重复字符子串的长度
     */
    public static int characterReplacement(String s, int k) {
        // 异常情况处理
        if (s == null || s.length() == 0) {
            return 0;
        }

        char[] chars = s.toCharArray();
        int n = chars.length;
        // 记录窗口内各字符的出现次数（A-Z 共 26 个字母）
        int[] count = new int[26];
        int maxCount = 0; // 窗口内出现次数最多的字符数量
        int maxLength = 0; // 最长子串长度
        int left = 0; // 窗口左边界

        // 滑动窗口右边界
        for (int right = 0; right < n; right++) {
            // 当前右边界字符计数加 1
            count[chars[right] - 'A']++;
            // 更新窗口内最大字符计数
            maxCount = Math.max(maxCount, count[chars[right] - 'A']);

            // 如果窗口大小减去最大字符计数大于 k，说明需要替换的字符超过 k 个
            // 需要收缩左边界
            // 核心条件：窗口大小 - 最多字符数量 > k 时，需要收缩窗口
            while (right - left + 1 - maxCount > k) {
                // 移除左边界字符
                count[chars[left] - 'A']--;
                // 移动左边界
                left++;
                // 注意：这里不需要重新计算 maxCount，因为即使 maxCount 变小了
                // 也不会影响最终结果，我们只需要记录历史最大值
            }

            // 更新最大长度（当前窗口大小）
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }
```

```
/**
 * 优化版本：使用历史最大值，避免每次重新计算 maxCount
 * 时间复杂度：O(n)，空间复杂度：O(1)
 *
 * @param s 输入字符串，只包含大写英文字母
 * @param k 最多可以替换的字符次数
 * @return 最长重复字符子串的长度
 */
public static int characterReplacementOptimized(String s, int k) {
    // 异常情况处理
    if (s == null || s.length() == 0) {
        return 0;
    }

    char[] chars = s.toCharArray();
    int n = chars.length;
    // 记录窗口内各字符的出现次数
    int[] count = new int[26];
    int maxCount = 0; // 历史最大字符计数
    int maxLength = 0; // 最长子串长度
    int left = 0; // 窗口左边界

    // 滑动窗口遍历字符串
    for (int right = 0; right < n; right++) {
        // 右边界字符计数加 1
        count[chars[right] - 'A']++;
        // 更新历史最大字符计数
        maxCount = Math.max(maxCount, count[chars[right] - 'A']);

        // 关键优化：使用历史最大值，即使窗口收缩后 maxCount 变小
        // 也不会影响结果，因为我们需要的是历史最大值
        // 当需要替换的字符数超过 k 时，收缩窗口
        if (right - left + 1 - maxCount > k) {
            // 移除左边界字符
            count[chars[left] - 'A']--;
            // 移动左边界
            left++;
        }

        // 更新最大长度（当前窗口大小）
        maxLength = Math.max(maxLength, right - left + 1);
    }
```

```
        return maxLength;
    }

    /**
     * 测试用例
     */
    public static void main(String[] args) {
        // 测试用例 1
        String s1 = "ABAB";
        int k1 = 2;
        int result1 = characterReplacement(s1, k1);
        System.out.println("输入: s = \"" + s1 + "\", k = " + k1);
        System.out.println("输出: " + result1);
        System.out.println("预期: 4");
        System.out.println("解释: 将两个'A'替换为'B', 得到'BBBB', 长度为 4");
        System.out.println();

        // 测试用例 2
        String s2 = "AABABBA";
        int k2 = 1;
        int result2 = characterReplacement(s2, k2);
        System.out.println("输入: s = \"" + s2 + "\", k = " + k2);
        System.out.println("输出: " + result2);
        System.out.println("预期: 4");
        System.out.println("解释: 将中间的'A'替换为'B', 得到'AABBBBA', 最长'B'子串长度为 4");
        System.out.println();

        // 测试用例 3：边界情况
        String s3 = "AAAA";
        int k3 = 2;
        int result3 = characterReplacement(s3, k3);
        System.out.println("输入: s = \"" + s3 + "\", k = " + k3);
        System.out.println("输出: " + result3);
        System.out.println("预期: 4");
        System.out.println("解释: 所有字符都相同, 无需替换");
        System.out.println();

        // 测试用例 4：空字符串
        String s4 = "";
        int k4 = 0;
        int result4 = characterReplacement(s4, k4);
        System.out.println("输入: s = \"" + s4 + "\", k = " + k4);
```

```
        System.out.println("输出: " + result4);
        System.out.println("预期: 0");
        System.out.println("解释: 空字符串");
        System.out.println();

        // 测试用例 5：k=0 的情况
        String s5 = "ABCDE";
        int k5 = 0;
        int result5 = characterReplacement(s5, k5);
        System.out.println("输入: s = \"" + s5 + "\", k = " + k5);
        System.out.println("输出: " + result5);
        System.out.println("预期: 1");
        System.out.println("解释: 不能替换任何字符，最长重复字符子串长度为 1");

        // 测试优化版本
        System.out.println("\n=== 优化版本测试 ===");
        int result1Opt = characterReplacementOptimized(s1, k1);
        System.out.println("优化版本结果 1: " + result1Opt);

        int result2Opt = characterReplacementOptimized(s2, k2);
        System.out.println("优化版本结果 2: " + result2Opt);
    }
}
```

==========

文件: Code17_LongestRepeatingCharacterReplacement.py

==========

```
# -*- coding: utf-8 -*-
"""
```

424. 替换后的最长重复字符问题解决方案

问题描述：
给你一个字符串 s 和一个整数 k 。你可以选择字符串中的任一字符，并将其更改为任何其他大写英文字符。
该操作最多可执行 k 次。
在执行上述操作后，返回包含相同字母的最长子字符串的长度。

解题思路：
使用滑动窗口维护一个窗口，窗口内最多有 k 个字符可以被替换成其他字符
核心思想：窗口大小 - 窗口内出现次数最多的字符数量 <= k

算法复杂度分析：
时间复杂度：O(n)，其中 n 是字符串长度

空间复杂度：O(1)，只需要 26 个字母的计数数组

是否最优解：是

相关题目链接：
LeetCode 424. 替换后的最长重复字符
https://leetcode.cn/problems/longest-repeating-character-replacement/

其他平台类似题目：
1. 牛客网 - 替换后的最长重复字符
   https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
2. LintCode 424. 替换后的最长重复字符
   https://www.lintcode.com/problem/424/
3. HackerRank - Longest Repeating Character Replacement
   https://www.hackerrank.com/challenges/longest-repeating-character-replacement/problem
4. CodeChef - REPLACE - Character Replacement
   https://www.codechef.com/problems/REPLACE
5. AtCoder - ABC146 D - Enough Array
   https://atcoder.jp/contests/abc146/tasks/abc146_d
6. 洛谷 P1886 滑动窗口
   https://www.luogu.com.cn/problem/P1886
7. 杭电 OJ 4193 Sliding Window
   http://acm.hdu.edu.cn/showproblem.php?pid=4193
8. POJ 2823 Sliding Window
   http://poj.org/problem?id=2823
9. UVa OJ 11536 - Smallest Sub-Array

https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
10. SPOJ - ADAFRIEN - Ada and Friends
    https://www.spoj.com/problems/ADAFRIEN/

工程化考量：
1. 异常处理：处理空字符串等边界情况
2. 性能优化：使用滑动窗口避免重复计算，达到线性时间复杂度
3. 可读性：变量命名清晰，添加详细注释，提供测试用例
"""

```
class Solution:
    """
    424. 替换后的最长重复字符解决方案类
    """
```

```
def characterReplacement(self, s: str, k: int) -> int:
    """
    计算替换 k 个字符后能获得的最长重复字符子串长度

    Args:
        s (str): 输入字符串，只包含大写英文字母
        k (int): 最多可以替换的字符次数

    Returns:
        int: 最长重复字符子串的长度

    Examples:
        >>> solution = Solution()
        >>> solution.characterReplacement("ABAB", 2)
        4
        >>> solution.characterReplacement("AABABBA", 1)
        4
    """
    # 异常情况处理
    if not s:
        return 0

    n = len(s)
    # 记录窗口内各字符的出现次数（A-Z 共 26 个字母）
    count = [0] * 26
    max_count = 0     # 窗口内出现次数最多的字符数量
    max_length = 0    # 最长子串长度
    left = 0          # 窗口左边界

    # 滑动窗口右边界
    for right in range(n):
        # 当前右边界字符计数加 1
        count[ord(s[right]) - ord('A')] += 1
        # 更新窗口内最大字符计数
        max_count = max(max_count, count[ord(s[right]) - ord('A')])

        # 如果窗口大小减去最大字符计数大于 k，说明需要替换的字符超过 k 个
        # 需要收缩左边界
        # 核心条件：窗口大小 - 最多字符数量 > k 时，需要收缩窗口
        while right - left + 1 - max_count > k:
            # 移除左边界字符
            count[ord(s[left]) - ord('A')] -= 1
```

```
            # 移动左边界
            left += 1
            # 注意：这里不需要重新计算 max_count，因为即使 max_count 变小了
            # 也不会影响最终结果，我们只需要记录历史最大值

        # 更新最大长度（当前窗口大小）
        max_length = max(max_length, right - left + 1)

    return max_length

def characterReplacementOptimized(self, s: str, k: int) -> int:
    """
    优化版本：使用历史最大值，避免每次重新计算 max_count
    时间复杂度：O(n)，空间复杂度：O(1)

    Args:
        s (str): 输入字符串，只包含大写英文字母
        k (int): 最多可以替换的字符次数

    Returns:
        int: 最长重复字符子串的长度
    """
    # 异常情况处理
    if not s:
        return 0

    n = len(s)
    # 记录窗口内各字符的出现次数
    count = [0] * 26
    max_count = 0     # 历史最大字符计数
    max_length = 0    # 最长子串长度
    left = 0          # 窗口左边界

    # 滑动窗口遍历字符串
    for right in range(n):
        # 右边界字符计数加 1
        count[ord(s[right]) - ord('A')] += 1
        # 更新历史最大字符计数
        max_count = max(max_count, count[ord(s[right]) - ord('A')])

        # 关键优化：使用历史最大值，即使窗口收缩后 max_count 变小
        # 也不会影响结果，因为我们需要的是历史最大值
        # 当需要替换的字符数超过 k 时，收缩窗口
```

```
            if right - left + 1 - max_count > k:
                # 移除左边界字符
                count[ord(s[left]) - ord('A')] -= 1
                # 移动左边界
                left += 1

            # 更新最大长度（当前窗口大小）
            max_length = max(max_length, right - left + 1)

        return max_length


def test_character_replacement():
    """
    测试函数
    """
    solution = Solution()

    # 测试用例 1
    s1 = "ABAB"
    k1 = 2
    result1 = solution.characterReplacement(s1, k1)
    print(f"输入: s = \"{s1}\", k = {k1}")
    print(f"输出: {result1}")
    print("预期: 4")
    print("解释: 将两个'A'替换为'B'，得到'BBBB'，长度为 4")
    print()

    # 测试用例 2
    s2 = "AABABBA"
    k2 = 1
    result2 = solution.characterReplacement(s2, k2)
    print(f"输入: s = \"{s2}\", k = {k2}")
    print(f"输出: {result2}")
    print("预期: 4")
    print("解释: 将中间的'A'替换为'B'，得到'AABBBBA'，最长'B'子串长度为 4")
    print()

    # 测试用例 3：边界情况
    s3 = "AAAA"
    k3 = 2
    result3 = solution.characterReplacement(s3, k3)
    print(f"输入: s = \"{s3}\", k = {k3}")
```

```
    print(f"输出: {result3}")
    print("预期: 4")
    print("解释: 所有字符都相同，无需替换")
    print()

    # 测试用例 4：空字符串
    s4 = ""
    k4 = 0
    result4 = solution.characterReplacement(s4, k4)
    print(f"输入: s = \"{s4}\", k = {k4}")
    print(f"输出: {result4}")
    print("预期: 0")
    print("解释: 空字符串")
    print()

    # 测试用例 5：k=0 的情况
    s5 = "ABCDE"
    k5 = 0
    result5 = solution.characterReplacement(s5, k5)
    print(f"输入: s = \"{s5}\", k = {k5}")
    print(f"输出: {result5}")
    print("预期: 1")
    print("解释: 不能替换任何字符，最长重复字符子串长度为 1")

    # 测试优化版本
    print("\n=== 优化版本测试 ===")
    result1_opt = solution.characterReplacementOptimized(s1, k1)
    print(f"优化版本结果 1: {result1_opt}")

    result2_opt = solution.characterReplacementOptimized(s2, k2)
    print(f"优化版本结果 2: {result2_opt}")


if __name__ == "__main__":
    test_character_replacement()

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

文件: Code18_SlidingWindowMedian.cpp
==========================================
#include <iostream>
#include <vector>
#include <queue>
```

```
#include <unordered_map>
#include <algorithm>
#include <string>
#include <functional> // 用于 greater<int>

using namespace std;

/**
 * 480. 滑动窗口中位数问题解决方案
 *
 * 问题描述：
 * 中位数是有序序列最中间的那个数。如果序列的长度是偶数，则没有最中间的数；此时中位数是最中间的两个数的平均数。
 * 给你一个数组 nums，有一个长度为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。
 * 你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。
 *
 * 解题思路：
 * 使用两个堆（最大堆和最小堆）来维护滑动窗口的中位数
 * 最大堆存储窗口左半部分（较小的一半），最小堆存储窗口右半部分（较大的一半）
 * 保持两个堆的大小平衡，最大堆的大小等于最小堆的大小或比最小堆大 1
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n*log k)，其中 n 是数组长度，k 是窗口大小
 * 空间复杂度：O(k)，用于存储窗口内的元素
 *
 * 是否最优解：是，这是处理滑动窗口中位数的最优解法
 *
 * 相关题目链接：
 * LeetCode 480. 滑动窗口中位数
 * https://leetcode.cn/problems/sliding-window-median/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 滑动窗口中位数
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 480. 滑动窗口中位数
 *    https://www.lintcode.com/problem/480/
 * 3. HackerRank - Sliding Window Median
 *    https://www.hackerrank.com/challenges/sliding-window-median/problem
 * 4. CodeChef - MEDIAN - Window Median
 *    https://www.codechef.com/problems/MEDIAN
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
```

```
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数或 0 等边界情况
 * 2. 性能优化：使用双堆维护中位数，避免重复排序
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 *
 * 编译说明：
 * 此代码需要 C++标准库支持，编译时请确保包含正确的头文件路径
 * 编译命令示例：g++ -std=c++11 Code18_SlidingWindowMedian.cpp -o Code18_SlidingWindowMedian
 */

// 算法实现（需要 C++标准库支持）
/*
// 需要包含的头文件：
// #include <iostream>
// #include <vector>
// #include <queue>
// #include <unordered_map>
// #include <algorithm>
// #include <string>
// #include <functional> // 用于 greater<int>
// using namespace std;

// 480. 滑动窗口中位数
class Solution {
public:
    // 计算滑动窗口中位数
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        // 异常情况处理
        if (nums.empty() || k <= 0) {
            return {};
```

```
        }

        int n = nums.size();
        vector<double> result(n - k + 1);

        // 最大堆（存储较小的一半），最小堆（存储较大的一半）
        // C++的 priority_queue 默认是最大堆
        priority_queue<int> maxHeap; // 存储较小的一半
        priority_queue<int, vector<int>, greater<int>> minHeap; // 最小堆，存储较大的一半

        // 初始化第一个窗口（前 k 个元素）
        for (int i = 0; i < k; i++) {
            addNumber(nums[i], maxHeap, minHeap);
        }

        // 计算第一个窗口的中位数
        result[0] = getMedian(maxHeap, minHeap, k);

        // 滑动窗口处理后续元素
        for (int i = k; i < n; i++) {
            // 移除窗口最左边的元素（i-k 位置的元素）
            removeNumber(nums[i - k], maxHeap, minHeap);
            // 添加新元素（i 位置的元素）
            addNumber(nums[i], maxHeap, minHeap);
            // 计算当前窗口中位数
            result[i - k + 1] = getMedian(maxHeap, minHeap, k);
        }

        return result;
    }

private:
    // 添加数字到堆中，保持堆的平衡
    void addNumber(int num, priority_queue<int>& maxHeap, priority_queue<int, vector<int>,
greater<int>>& minHeap) {
        // 先添加到最大堆（较小的一半）
        maxHeap.push(num);
        // 将最大堆的最大值移动到最小堆（较大的一半）
        minHeap.push(maxHeap.top());
        maxHeap.pop();

        // 如果最小堆的大小大于最大堆，重新平衡
        // 保持最大堆的大小等于最小堆的大小或比最小堆大 1
```

```
        if (minHeap.size() > maxHeap.size()) {
            maxHeap.push(minHeap.top());
            minHeap.pop();
        }
    }

    // 从堆中移除数字，保持堆的平衡
    void removeNumber(int num, priority_queue<int>& maxHeap, priority_queue<int, vector<int>,
greater<int>>& minHeap) {
        // 判断数字在哪个堆中
        if (num <= maxHeap.top()) {
            // 数字在最大堆中，从最大堆中移除
            vector<int> temp;
            // 找到要移除的元素并移除它
            while (!maxHeap.empty() && maxHeap.top() != num) {
                temp.push_back(maxHeap.top());
                maxHeap.pop();
            }
            if (!maxHeap.empty()) {
                maxHeap.pop();
            }
            // 将临时存储的元素重新放回堆中
            for (int val : temp) {
                maxHeap.push(val);
            }

            // 如果最大堆的大小小于最小堆，从最小堆移动一个元素到最大堆
            if (maxHeap.size() < minHeap.size()) {
                maxHeap.push(minHeap.top());
                minHeap.pop();
            }
        } else {
            // 数字在最小堆中，从最小堆中移除
            vector<int> temp;
            // 找到要移除的元素并移除它
            while (!minHeap.empty() && minHeap.top() != num) {
                temp.push_back(minHeap.top());
                minHeap.pop();
            }
            if (!minHeap.empty()) {
                minHeap.pop();
            }
            // 将临时存储的元素重新放回堆中
```

```
            for (int val : temp) {
                minHeap.push(val);
            }

            // 如果最大堆的大小比最小堆大 1 以上，从最大堆移动一个元素到最小堆
            if (maxHeap.size() > minHeap.size() + 1) {
                minHeap.push(maxHeap.top());
                maxHeap.pop();
            }
        }
    }

    // 获取当前中位数
    double getMedian(priority_queue<int>& maxHeap, priority_queue<int, vector<int>,
greater<int>>& minHeap, int k) {
        if (k % 2 == 1) {
            // 奇数长度，中位数是最大堆的堆顶（较小一半的最大值）
            return static_cast<double>(maxHeap.top());
        } else {
            // 偶数长度，中位数是两个堆顶的平均值
            return (static_cast<double>(maxHeap.top()) + static_cast<double>(minHeap.top())) /
2.0;
        }
    }
};

// 测试函数
void testMedianSlidingWindow() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
    int k1 = 3;
    vector<double> result1 = solution.medianSlidingWindow(nums1, k1);
    // 预期: [1.0, -1.0, -1.0, 3.0, 5.0, 6.0]

    // 测试用例 2
    vector<int> nums2 = {1, 2, 3, 4, 2, 3, 1, 4, 2};
    int k2 = 3;
    vector<double> result2 = solution.medianSlidingWindow(nums2, k2);
}

int main() {
```

```
    testMedianSlidingWindow();
    return 0;
}
*/

// 算法核心逻辑说明（伪代码形式）：
/*
class Solution {
public:
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        if (nums.empty() || k <= 0) {
            return {};
        }

        int n = nums.size();
        vector<double> result(n - k + 1);

        // 两个堆维护中位数
        priority_queue<int> maxHeap;  // 存储较小的一半
        priority_queue<int, vector<int>, greater<int>> minHeap;  // 存储较大的一半

        // 初始化第一个窗口
        for (int i = 0; i < k; i++) {
            addNumber(nums[i], maxHeap, minHeap);
        }
        result[0] = getMedian(maxHeap, minHeap, k);

        // 滑动窗口
        for (int i = k; i < n; i++) {
            removeNumber(nums[i - k], maxHeap, minHeap);
            addNumber(nums[i], maxHeap, minHeap);
            result[i - k + 1] = getMedian(maxHeap, minHeap, k);
        }

        return result;
    }
};

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

文件: Code18_SlidingWindowMedian.java
==========================================
package class049;
```

```
import java.util.*;

/**
 * 480. 滑动窗口中位数问题解决方案
 *
 * 问题描述：
 * 中位数是有序序列最中间的那个数。如果序列的长度是偶数，则没有最中间的数；此时中位数是最中间的两个数的平均数。
 * 给你一个数组 nums，有一个长度为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。
 * 你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。
 *
 * 解题思路：
 * 使用两个堆（最大堆和最小堆）来维护滑动窗口的中位数
 * 最大堆存储窗口左半部分（较小的一半），最小堆存储窗口右半部分（较大的一半）
 * 保持两个堆的大小平衡，最大堆的大小等于最小堆的大小或比最小堆大 1
 *
 * 算法复杂度分析：
 * 时间复杂度：O(n*log k)，其中 n 是数组长度，k 是窗口大小
 * 空间复杂度：O(k)，用于存储窗口内的元素
 *
 * 是否最优解：是，这是处理滑动窗口中位数的最优解法
 *
 * 相关题目链接：
 * LeetCode 480. 滑动窗口中位数
 * https://leetcode.cn/problems/sliding-window-median/
 *
 * 其他平台类似题目：
 * 1. 牛客网 - 滑动窗口中位数
 *    https://www.nowcoder.com/practice/1266570c4a06487981ed50e84e8b720d
 * 2. LintCode 480. 滑动窗口中位数
 *    https://www.lintcode.com/problem/480/
 * 3. HackerRank - Sliding Window Median
 *    https://www.hackerrank.com/challenges/sliding-window-median/problem
 * 4. CodeChef - MEDIAN - Window Median
 *    https://www.codechef.com/problems/MEDIAN
 * 5. AtCoder - ABC146 D - Enough Array
 *    https://atcoder.jp/contests/abc146/tasks/abc146_d
 * 6. 洛谷 P1886 滑动窗口
 *    https://www.luogu.com.cn/problem/P1886
 * 7. 杭电 OJ 4193 Sliding Window
 *    http://acm.hdu.edu.cn/showproblem.php?pid=4193
```

```
 * 8. POJ 2823 Sliding Window
 *    http://poj.org/problem?id=2823
 * 9. UVa OJ 11536 - Smallest Sub-Array
 *
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=2531
 * 10. SPOJ - ADAFRIEN - Ada and Friends
 *     https://www.spoj.com/problems/ADAFRIEN/
 *
 * 工程化考量：
 * 1. 异常处理：处理空数组、k 为负数或 0 等边界情况
 * 2. 性能优化：使用双堆维护中位数，避免重复排序
 * 3. 可读性：变量命名清晰，添加详细注释，提供测试用例
 */
public class Code18_SlidingWindowMedian {

    /**
     * 计算滑动窗口中位数
     *
     * @param nums 输入数组
     * @param k 窗口大小
     * @return 每个窗口中位数的数组
     */
    public static double[] medianSlidingWindow(int[] nums, int k) {
        // 异常情况处理
        if (nums == null || nums.length == 0 || k <= 0) {
            return new double[0];
        }

        int n = nums.length;
        double[] result = new double[n - k + 1];

        // 最大堆（存储较小的一半），最小堆（存储较大的一半）
        // Java 的 PriorityQueue 默认是最小堆，使用 Collections.reverseOrder() 创建最大堆
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder()); // 存储较小的一半
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(); // 存储较大的一半

        // 初始化第一个窗口（前 k 个元素）
        for (int i = 0; i < k; i++) {
            addNumber(nums[i], maxHeap, minHeap);
        }
```

```
        // 计算第一个窗口的中位数
        result[0] = getMedian(maxHeap, minHeap, k);

        // 滑动窗口处理后续元素
        for (int i = k; i < n; i++) {
            // 移除窗口最左边的元素（i-k 位置的元素）
            removeNumber(nums[i - k], maxHeap, minHeap);
            // 添加新元素（i 位置的元素）
            addNumber(nums[i], maxHeap, minHeap);
            // 计算当前窗口中位数
            result[i - k + 1] = getMedian(maxHeap, minHeap, k);
        }

        return result;
    }

    /**
     * 添加数字到堆中，保持堆的平衡
     *
     * @param num 要添加的数字
     * @param maxHeap 最大堆（存储较小的一半）
     * @param minHeap 最小堆（存储较大的一半）
     */
    private static void addNumber(int num, PriorityQueue<Integer> maxHeap, PriorityQueue<Integer>
minHeap) {
        // 先添加到最大堆（较小的一半）
        maxHeap.offer(num);
        // 将最大堆的最大值移动到最小堆（较大的一半）
        minHeap.offer(maxHeap.poll());

        // 如果最小堆的大小大于最大堆，重新平衡
        // 保持最大堆的大小等于最小堆的大小或比最小堆大 1
        if (minHeap.size() > maxHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }
    }

    /**
     * 从堆中移除数字，保持堆的平衡
     *
     * @param num 要移除的数字
     * @param maxHeap 最大堆（存储较小的一半）
     * @param minHeap 最小堆（存储较大的一半）
```

```
     */
    private static void removeNumber(int num, PriorityQueue<Integer> maxHeap,
PriorityQueue<Integer> minHeap) {
        // 判断数字在哪个堆中
        if (num <= maxHeap.peek()) {
            // 数字在最大堆中
            maxHeap.remove(num);
            // 如果最大堆的大小小于最小堆，从最小堆移动一个元素到最大堆
            if (maxHeap.size() < minHeap.size()) {
                maxHeap.offer(minHeap.poll());
            }
        } else {
            // 数字在最小堆中
            minHeap.remove(num);
            // 如果最大堆的大小比最小堆大 1 以上，从最大堆移动一个元素到最小堆
            if (maxHeap.size() > minHeap.size() + 1) {
                minHeap.offer(maxHeap.poll());
            }
        }
    }

    /**
     * 获取当前中位数
     *
     * @param maxHeap 最大堆（存储较小的一半）
     * @param minHeap 最小堆（存储较大的一半）
     * @param k 窗口大小
     * @return 当前窗口的中位数
     */
    private static double getMedian(PriorityQueue<Integer> maxHeap, PriorityQueue<Integer>
minHeap, int k) {
        if (k % 2 == 1) {
            // 奇数长度，中位数是最大堆的堆顶（较小一半的最大值）
            return (double) maxHeap.peek();
        } else {
            // 偶数长度，中位数是两个堆顶的平均值
            // 注意：使用 long 避免整数溢出
            return ((double) maxHeap.peek() + (double) minHeap.peek()) / 2.0;
        }
    }

    /**
     * 优化版本：使用延迟删除技术处理重复元素
```

```
     * 时间复杂度：O(n*log k)，空间复杂度：O(k)
     *
     * @param nums 输入数组
     * @param k 窗口大小
     * @return 每个窗口中位数的数组
     */
    public static double[] medianSlidingWindowOptimized(int[] nums, int k) {
        // 异常情况处理
        if (nums == null || nums.length == 0 || k <= 0) {
            return new double[0];
        }

        int n = nums.length;
        double[] result = new double[n - k + 1];

        // 使用延迟删除技术
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder()); // 存储较小的一半
        PriorityQueue<Integer> minHeap = new PriorityQueue<>(); // 存储较大的一半
        Map<Integer, Integer> delayed = new HashMap<>(); // 延迟删除的计数器

        // 平衡因子：maxHeap.size() - minHeap.size()
        int balance = 0;

        // 初始化第一个窗口（前 k 个元素）
        for (int i = 0; i < k; i++) {
            addNumberOptimized(nums[i], maxHeap, minHeap, delayed, balance);
        }

        result[0] = getMedianOptimized(maxHeap, minHeap, k, delayed);

        // 滑动窗口处理后续元素
        for (int i = k; i < n; i++) {
            // 移除窗口最左边的元素（i-k 位置的元素）
            removeNumberOptimized(nums[i - k], maxHeap, minHeap, delayed, balance);
            // 添加新元素（i 位置的元素）
            addNumberOptimized(nums[i], maxHeap, minHeap, delayed, balance);
            // 清理延迟删除的元素
            pruneHeaps(maxHeap, minHeap, delayed);
            // 计算当前窗口中位数
            result[i - k + 1] = getMedianOptimized(maxHeap, minHeap, k, delayed);
        }
```

```
        return result;
    }

    /**
     * 优化版本：添加数字到堆中
     */
    private static void addNumberOptimized(int num, PriorityQueue<Integer> maxHeap,
PriorityQueue<Integer> minHeap,
                                           Map<Integer, Integer> delayed, int balance) {
        if (maxHeap.isEmpty() || num <= maxHeap.peek()) {
            maxHeap.offer(num);
            balance++;
        } else {
            minHeap.offer(num);
            balance--;
        }

        // 重新平衡堆
        rebalanceHeaps(maxHeap, minHeap, delayed, balance);
    }

    /**
     * 优化版本：从堆中移除数字
     */
    private static void removeNumberOptimized(int num, PriorityQueue<Integer> maxHeap,
PriorityQueue<Integer> minHeap,
                                              Map<Integer, Integer> delayed, int balance) {
        delayed.put(num, delayed.getOrDefault(num, 0) + 1);

        if (!maxHeap.isEmpty() && num <= maxHeap.peek()) {
            balance--;
        } else {
            balance++;
        }

        // 重新平衡堆
        rebalanceHeaps(maxHeap, minHeap, delayed, balance);
    }

    /**
     * 重新平衡堆的大小
     */
    private static void rebalanceHeaps(PriorityQueue<Integer> maxHeap, PriorityQueue<Integer>
```

```
minHeap,
                                    Map<Integer, Integer> delayed, int balance) {
        // 平衡堆的大小
        if (balance > 1) {
            minHeap.offer(maxHeap.poll());
            balance -= 2;
        } else if (balance < -1) {
            maxHeap.offer(minHeap.poll());
            balance += 2;
        }
    }

    /**
     * 清理堆中延迟删除的元素
     */
    private static void pruneHeaps(PriorityQueue<Integer> maxHeap, PriorityQueue<Integer>
minHeap,
                                   Map<Integer, Integer> delayed) {
        // 清理最大堆顶部的延迟删除元素
        while (!maxHeap.isEmpty() && delayed.getOrDefault(maxHeap.peek(), 0) > 0) {
            int num = maxHeap.poll();
            delayed.put(num, delayed.get(num) - 1);
            if (delayed.get(num) == 0) {
                delayed.remove(num);
            }
        }

        // 清理最小堆顶部的延迟删除元素
        while (!minHeap.isEmpty() && delayed.getOrDefault(minHeap.peek(), 0) > 0) {
            int num = minHeap.poll();
            delayed.put(num, delayed.get(num) - 1);
            if (delayed.get(num) == 0) {
                delayed.remove(num);
            }
        }
    }

    /**
     * 优化版本：获取当前中位数
     */
    private static double getMedianOptimized(PriorityQueue<Integer> maxHeap,
PriorityQueue<Integer> minHeap,
                                             int k, Map<Integer, Integer> delayed) {
```

```
        pruneHeaps(maxHeap, minHeap, delayed);

        if (k % 2 == 1) {
            return (double) maxHeap.peek();
        } else {
            return ((double) maxHeap.peek() + (double) minHeap.peek()) / 2.0;
        }
    }

    /**
     * 测试用例
     */
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {1, 3, -1, -3, 5, 3, 6, 7};
        int k1 = 3;
        double[] result1 = medianSlidingWindow(nums1, k1);
        System.out.println("输入数组: " + Arrays.toString(nums1));
        System.out.println("窗口大小: " + k1);
        System.out.println("中位数序列: " + Arrays.toString(result1));
        System.out.println("预期: [1.0, -1.0, -1.0, 3.0, 5.0, 6.0]");
        System.out.println("解释: 窗口[1,3,-1]中位数 1.0, 窗口[3,-1,-3]中位数-1.0, ...");
        System.out.println();

        // 测试用例 2
        int[] nums2 = {1, 2, 3, 4, 2, 3, 1, 4, 2};
        int k2 = 3;
        double[] result2 = medianSlidingWindow(nums2, k2);
        System.out.println("输入数组: " + Arrays.toString(nums2));
        System.out.println("窗口大小: " + k2);
        System.out.println("中位数序列: " + Arrays.toString(result2));
        System.out.println();

        // 测试用例 3：边界情况，k=1
        int[] nums3 = {5};
        int k3 = 1;
        double[] result3 = medianSlidingWindow(nums3, k3);
        System.out.println("输入数组: " + Arrays.toString(nums3));
        System.out.println("窗口大小: " + k3);
        System.out.println("中位数序列: " + Arrays.toString(result3));
        System.out.println("预期: [5.0]");
        System.out.println("解释: 每个窗口只有一个元素，中位数就是该元素");
        System.out.println();
```

```
        // 测试用例 4：k 等于数组长度
        int[] nums4 = {1, 2, 3, 4, 5};
        int k4 = 5;
        double[] result4 = medianSlidingWindow(nums4, k4);
        System.out.println("输入数组: " + Arrays.toString(nums4));
        System.out.println("窗口大小: " + k4);
        System.out.println("中位数序列: " + Arrays.toString(result4));
        System.out.println("预期: [3.0]");
        System.out.println("解释: 整个数组作为一个窗口，中位数是 3.0");

        // 测试优化版本
        System.out.println("\n=== 优化版本测试 ===");
        double[] result1Opt = medianSlidingWindowOptimized(nums1, k1);
        System.out.println("优化版本结果 1: " + Arrays.toString(result1Opt));
    }
}

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

文件: Code18_SlidingWindowMedian.py
==========================================
import heapq
from typing import List
import collections

class Solution:
    """
    480. 滑动窗口中位数
    中位数是有序序列最中间的那个数。如果序列的长度是偶数，则没有最中间的数；此时中位数是最中间的两个数的平均数。
    给你一个数组 nums，有一个长度为 k 的窗口从最左端滑动到最右端。窗口中有 k 个数，每次窗口向右移动 1 位。
    你的任务是找出每次窗口移动后得到的新窗口中元素的中位数，并输出由它们组成的数组。

    解题思路：
    使用两个堆（最大堆和最小堆）来维护滑动窗口的中位数
    最大堆存储窗口左半部分（较小的一半），最小堆存储窗口右半部分（较大的一半）
    保持两个堆的大小平衡，最大堆的大小等于最小堆的大小或比最小堆大 1

    时间复杂度：O(n*log k)，其中 n 是数组长度，k 是窗口大小
    空间复杂度：O(k)，用于存储窗口内的元素
```

是否最优解：是，这是处理滑动窗口中位数的最优解法

测试链接：https://leetcode.cn/problems/sliding-window-median/
"""

```
def medianSlidingWindow(self, nums: List[int], k: int) -> List[float]:
    """
    计算滑动窗口中位数

    Args:
        nums: 输入数组
        k: 窗口大小

    Returns:
        每个窗口中位数的数组
    """
    if not nums or k <= 0:
        return []

    n = len(nums)
    result = []

    # 最大堆（存储较小的一半），最小堆（存储较大的一半）
    # Python 中最小堆可以通过取负数实现最大堆
    max_heap = []  # 存储负数，实现最大堆
    min_heap = []  # 正常的最小堆

    # 初始化第一个窗口
    for i in range(k):
        self.add_number(nums[i], max_heap, min_heap)

    result.append(self.get_median(max_heap, min_heap, k))

    # 滑动窗口
    for i in range(k, n):
        # 移除窗口最左边的元素
        self.remove_number(nums[i - k], max_heap, min_heap)
        # 添加新元素
        self.add_number(nums[i], max_heap, min_heap)
        # 计算当前窗口中位数
        result.append(self.get_median(max_heap, min_heap, k))

    return result
```

```
def add_number(self, num: int, max_heap: List[int], min_heap: List[int]) -> None:
    """
    添加数字到堆中，保持堆的平衡
    """
    # 先添加到最大堆（存储负数）
    heapq.heappush(max_heap, -num)
    # 将最大堆的最大值移动到最小堆
    heapq.heappush(min_heap, -heapq.heappop(max_heap))

    # 如果最小堆的大小大于最大堆，重新平衡
    if len(min_heap) > len(max_heap):
        heapq.heappush(max_heap, -heapq.heappop(min_heap))

def remove_number(self, num: int, max_heap: List[int], min_heap: List[int]) -> None:
    """
    从堆中移除数字，保持堆的平衡
    """
    # 判断数字在哪个堆中
    if num <= -max_heap[0]:
        # 从最大堆中移除
        # 由于 Python 堆不支持直接删除，需要重建堆
        temp = []
        while max_heap and -max_heap[0] != num:
            temp.append(heapq.heappop(max_heap))
        if max_heap:
            heapq.heappop(max_heap)
        for val in temp:
            heapq.heappush(max_heap, val)

        # 如果最大堆的大小小于最小堆，从最小堆移动一个元素到最大堆
        if len(max_heap) < len(min_heap):
            heapq.heappush(max_heap, -heapq.heappop(min_heap))
    else:
        # 从最小堆中移除
        temp = []
        while min_heap and min_heap[0] != num:
            temp.append(heapq.heappop(min_heap))
        if min_heap:
            heapq.heappop(min_heap)
        for val in temp:
            heapq.heappush(min_heap, val)
```

```
            # 如果最大堆的大小比最小堆大 1 以上，从最大堆移动一个元素到最小堆
            if len(max_heap) > len(min_heap) + 1:
                heapq.heappush(min_heap, -heapq.heappop(max_heap))

    def get_median(self, max_heap: List[int], min_heap: List[int], k: int) -> float:
        """
        获取当前中位数
        """
        if k % 2 == 1:
            # 奇数长度，中位数是最大堆的堆顶（取负数）
            return -max_heap[0]
        else:
            # 偶数长度，中位数是两个堆顶的平均值
            return (-max_heap[0] + min_heap[0]) / 2.0


class SolutionOptimized:
    """
    优化版本：使用延迟删除技术处理重复元素
    时间复杂度：O(n*log k)，空间复杂度：O(k)
    """

    def medianSlidingWindow(self, nums: List[int], k: int) -> List[float]:
        if not nums or k <= 0:
            return []

        n = len(nums)
        result = []

        # 使用延迟删除技术
        max_heap = []  # 存储负数，实现最大堆
        min_heap = []  # 正常的最小堆
        delayed = collections.Counter()  # 延迟删除的计数器

        # 平衡因子：max_heap 的大小 - min_heap 的大小
        balance = 0

        # 初始化第一个窗口
        for i in range(k):
            self.add_number(nums[i], max_heap, min_heap, delayed, balance)

        result.append(self.get_median(max_heap, min_heap, k, delayed))
```

```
    # 滑动窗口
    for i in range(k, n):
        # 移除窗口最左边的元素
        self.remove_number(nums[i - k], max_heap, min_heap, delayed, balance)
        # 添加新元素
        self.add_number(nums[i], max_heap, min_heap, delayed, balance)
        # 清理延迟删除的元素
        self.prune_heaps(max_heap, min_heap, delayed)
        # 计算当前窗口中位数
        result.append(self.get_median(max_heap, min_heap, k, delayed))

    return result

def add_number(self, num: int, max_heap: List[int], min_heap: List[int],
               delayed: collections.Counter, balance: int) -> None:
    if not max_heap or num <= -max_heap[0]:
        heapq.heappush(max_heap, -num)
        balance += 1
    else:
        heapq.heappush(min_heap, num)
        balance -= 1

    # 重新平衡堆
    self.rebalance_heaps(max_heap, min_heap, delayed, balance)

def remove_number(self, num: int, max_heap: List[int], min_heap: List[int],
                  delayed: collections.Counter, balance: int) -> None:
    delayed[num] += 1

    if not max_heap or num <= -max_heap[0]:
        balance -= 1
    else:
        balance += 1

    # 重新平衡堆
    self.rebalance_heaps(max_heap, min_heap, delayed, balance)

def rebalance_heaps(self, max_heap: List[int], min_heap: List[int],
                    delayed: collections.Counter, balance: int) -> None:
    # 平衡堆的大小
    if balance > 1:
        heapq.heappush(min_heap, -heapq.heappop(max_heap))
        balance -= 2
```

```
        elif balance < -1:
            heapq.heappush(max_heap, -heapq.heappop(min_heap))
            balance += 2

    def prune_heaps(self, max_heap: List[int], min_heap: List[int],
                    delayed: collections.Counter) -> None:
        # 清理最大堆顶部的延迟删除元素
        while max_heap and delayed.get(-max_heap[0], 0) > 0:
            num = -heapq.heappop(max_heap)
            delayed[num] -= 1
            if delayed[num] == 0:
                del delayed[num]

        # 清理最小堆顶部的延迟删除元素
        while min_heap and delayed.get(min_heap[0], 0) > 0:
            num = heapq.heappop(min_heap)
            delayed[num] -= 1
            if delayed[num] == 0:
                del delayed[num]

    def get_median(self, max_heap: List[int], min_heap: List[int],
                   k: int, delayed: collections.Counter) -> float:
        self.prune_heaps(max_heap, min_heap, delayed)

        if k % 2 == 1:
            return -max_heap[0]
        else:
            return (-max_heap[0] + min_heap[0]) / 2.0


def test_median_sliding_window():
    """
    测试函数
    """
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 3, -1, -3, 5, 3, 6, 7]
    k1 = 3
    result1 = solution.medianSlidingWindow(nums1, k1)
    print(f"输入数组: {nums1}")
    print(f"窗口大小: {k1}")
    print(f"中位数序列: {result1}")
```

```
    print("预期: [1.0, -1.0, -1.0, 3.0, 5.0, 6.0]")
    print()

    # 测试用例 2
    nums2 = [1, 2, 3, 4, 2, 3, 1, 4, 2]
    k2 = 3
    result2 = solution.medianSlidingWindow(nums2, k2)
    print(f"输入数组: {nums2}")
    print(f"窗口大小: {k2}")
    print(f"中位数序列: {result2}")
    print()

    # 测试用例 3：边界情况，k=1
    nums3 = [5]
    k3 = 1
    result3 = solution.medianSlidingWindow(nums3, k3)
    print(f"输入数组: {nums3}")
    print(f"窗口大小: {k3}")
    print(f"中位数序列: {result3}")
    print("预期: [5.0]")
    print()

    # 测试用例 4：k 等于数组长度
    nums4 = [1, 2, 3, 4, 5]
    k4 = 5
    result4 = solution.medianSlidingWindow(nums4, k4)
    print(f"输入数组: {nums4}")
    print(f"窗口大小: {k4}")
    print(f"中位数序列: {result4}")
    print("预期: [3.0]")


if __name__ == "__main__":
    test_median_sliding_window()

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

文件: Code19_FindDuplicateSubtrees.cpp
==============================================
#include <iostream>
#include <vector>
#include <unordered_map>
#include <string>
```

```
#include <chrono>

/**
 * LeetCode 652. 寻找重复的子树 (Find Duplicate Subtrees)
 *
 * 题目描述:
 * 给定一棵二叉树，返回所有重复的子树。
 * 对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。
 * 两棵树重复是指它们具有相同的结构以及相同的结点值。
 *
 * 示例 1:
 *         1
 *       / \
 *      2   3
 *     /   / \
 *    4   2   4
 *       /
 *      4
 *
 * 输出:
 * [[2,4],[4]]
 *
 * 解释:
 * 上面的二叉树有两个重复子树。
 * 第一个重复子树是 4，如蓝色节点所示。
 * 第二个重复子树是 2 -> 4，如橙色节点所示。
 *
 * 题目链接: https://leetcode.com/problems/find-duplicate-subtrees/
 *
 * 解题思路:
 * 这道题需要我们找出二叉树中所有重复的子树。解决这个问题的关键在于能够唯一地表示每个子树，并能够快速判断是否已经存在相同的子树。
 *
 * 解法: 递归 + 哈希表
 * 1. 对于每个子树，我们需要生成一个唯一标识符，可以通过序列化的方式实现
 * 2. 使用哈希表来记录每个子树标识符出现的次数
 * 3. 当一个子树标识符出现次数为 2 时，将该子树的根节点加入结果列表
 *
 * 时间复杂度: O(n^2)，其中 n 是树中的节点数。在最坏情况下，序列化每个节点需要 O(n)时间，共有 n 个节点。
 * 空间复杂度: O(n^2)，存储所有子树的序列化表示。
 *
 * 优化版本：使用 ID 来代替完整序列化字符串，可以将时间和空间复杂度优化到 O(n)。
```

```
 */

// 定义二叉树节点
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * 解法一: 使用序列化 + 哈希表
     * 将每个子树序列化为字符串，然后使用哈希表记录每个子树出现的次数。
     */
    std::vector<TreeNode*> findDuplicateSubtrees(TreeNode* root) {
        std::vector<TreeNode*> result;
        if (!root) {
            return result;
        }

        // 哈希表用于记录每个子树序列化字符串出现的次数
        std::unordered_map<std::string, int> subtreeCount;

        // 递归函数，用于序列化子树并检查重复
        serializeAndCheckDuplicates(root, subtreeCount, result);

        return result;
    }

    /**
     * 递归序列化子树并检查重复
     *
     * @param node 当前节点
     * @param subtreeCount 子树计数哈希表
     * @param result 结果列表
     * @return 当前子树的序列化字符串
     */
    std::string serializeAndCheckDuplicates(TreeNode* node,
                                            std::unordered_map<std::string, int>& subtreeCount,
```

```
                                    std::vector<TreeNode*>& result) {
    if (!node) {
        return "#"; // 用#表示空节点
    }

    // 序列化当前节点的左子树、当前节点的值、右子树
    std::string key = std::to_string(node->val) + "," +
                      serializeAndCheckDuplicates(node->left, subtreeCount, result) + "," +
                      serializeAndCheckDuplicates(node->right, subtreeCount, result);

    // 获取当前子树出现的次数，如果是第二次出现，则添加到结果中
    subtreeCount[key]++;
    if (subtreeCount[key] == 2) {
        result.push_back(node); // 只有当子树出现次数为 2 时添加，避免重复添加
    }

    return key;
}

/**
 * 解法二: 使用 ID 代替完整序列化字符串（优化版本）
 * 为每个不同的子树分配一个唯一 ID，使用 ID 来标识子树而不是完整的序列化字符串。
 */
std::vector<TreeNode*> findDuplicateSubtreesOptimized(TreeNode* root) {
    std::vector<TreeNode*> result;
    if (!root) {
        return result;
    }

    // 哈希表用于将子树的序列化字符串映射到唯一 ID
    std::unordered_map<std::string, int> subtreeId;
    // 哈希表用于记录每个 ID（子树）出现的次数
    std::unordered_map<int, int> idCount;
    // 当前可用的下一个 ID
    int nextId = 1;

    // 递归函数，使用 ID 检查重复子树
    findDuplicatesWithId(root, subtreeId, idCount, nextId, result);

    return result;
}

/**
```

```
     * 递归函数，使用 ID 检查重复子树
     *
     * @param node 当前节点
     * @param subtreeId 子树到 ID 的映射
     * @param idCount ID 出现次数的映射
     * @param nextId 下一个可用的 ID
     * @param result 结果列表
     * @return 当前子树的 ID
     */
    int findDuplicatesWithId(TreeNode* node,
                             std::unordered_map<std::string, int>& subtreeId,
                             std::unordered_map<int, int>& idCount,
                             int& nextId,
                             std::vector<TreeNode*>& result) {
        if (!node) {
            return 0; // 空节点的 ID 为 0
        }

        // 构建当前子树的键
        std::string key = std::to_string(node->val) + "," +
                          std::to_string(findDuplicatesWithId(node->left, subtreeId, idCount,
nextId, result)) + "," +
                          std::to_string(findDuplicatesWithId(node->right, subtreeId, idCount,
nextId, result));

        // 如果当前子树还没有分配 ID，则分配一个新 ID
        if (subtreeId.find(key) == subtreeId.end()) {
            subtreeId[key] = nextId++; // 分配新 ID
        }

        int id = subtreeId[key];

        // 增加当前 ID 的计数，并在计数为 2 时添加到结果中
        idCount[id]++;
        if (idCount[id] == 2) {
            result.push_back(node);
        }

        return id;
    }
};

/**
```

```
 * 将树转换为字符串表示（用于打印结果）
 */
std::string treeToString(TreeNode* root) {
    if (!root) {
        return "null";
    }

    std::string result = std::to_string(root->val);
    if (root->left || root->right) {
        result += "[";
        result += treeToString(root->left);
        result += ",";
        result += treeToString(root->right);
        result += "]";
    }

    return result;
}

/**
 * 打印结果列表
 */
void printResult(const std::vector<TreeNode*>& result) {
    std::cout << "[";
    for (size_t i = 0; i < result.size(); i++) {
        std::cout << treeToString(result[i]);
        if (i < result.size() - 1) {
            std::cout << ", ";
        }
    }
    std::cout << "]" << std::endl;
}

/**
 * 构建测试用例中的树
 */
TreeNode* buildExampleTree() {
    // 构建示例中的树
    //       1
    //      / \
    //     2   3
    //    /   / \
    //   4   2   4
```

```
    //      /
    //     4
    TreeNode* node1 = new TreeNode(1);
    TreeNode* node2 = new TreeNode(2);
    TreeNode* node3 = new TreeNode(3);
    TreeNode* node4 = new TreeNode(4);
    TreeNode* node5 = new TreeNode(2);
    TreeNode* node6 = new TreeNode(4);
    TreeNode* node7 = new TreeNode(4);

    node1->left = node2;
    node1->right = node3;
    node2->left = node4;
    node3->left = node5;
    node3->right = node6;
    node5->left = node7;

    return node1;
}

/**
 * 释放树的内存
 */
void deleteTree(TreeNode* root) {
    if (!root) return;
    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
}

/**
 * 构建一个带有重复子树的平衡二叉树
 *
 * @param start 起始值
 * @param end 结束值
 * @return 构建的树的根节点
 */
TreeNode* buildBalancedTreeWithDuplicates(int start, int end) {
    if (start > end) {
        return nullptr;
    }

    int mid = start + (end - start) / 2;
```

```
    TreeNode* root = new TreeNode(mid);

    // 为了创建重复子树，我们可以使部分子树的值重复
    if (start <= end - 2) {
        root->left = buildBalancedTreeWithDuplicates(start, mid - 1);
        root->right = buildBalancedTreeWithDuplicates(start, mid - 1); // 重复左子树的结构
    } else {
        root->left = buildBalancedTreeWithDuplicates(start, mid - 1);
        root->right = buildBalancedTreeWithDuplicates(mid + 1, end);
    }

    return root;
}

int main() {
    Solution solution;

    // 测试用例 1: 示例中的树
    TreeNode* root1 = buildExampleTree();
    std::cout << "测试用例 1:" << std::endl;
    std::cout << "解法一（序列化）结果: ";
    printResult(solution.findDuplicateSubtrees(root1)); // 预期输出类似: [2[4,null],4]

    // 重新构建树，因为解法一可能修改了树的状态（虽然这里不会，但为了保险起见）
    TreeNode* root1Again = buildExampleTree();
    std::cout << "解法二（ID 优化）结果: ";
    printResult(solution.findDuplicateSubtreesOptimized(root1Again)); // 预期输出类似:
[2[4,null],4]
    std::cout << std::endl;

    // 释放内存
    deleteTree(root1);
    deleteTree(root1Again);

    // 测试用例 2: 空树
    TreeNode* root2 = nullptr;
    std::cout << "测试用例 2（空树）:" << std::endl;
    std::cout << "解法一（序列化）结果: ";
    printResult(solution.findDuplicateSubtrees(root2)); // 预期输出: []
    std::cout << "解法二（ID 优化）结果: ";
    printResult(solution.findDuplicateSubtreesOptimized(root2)); // 预期输出: []
    std::cout << std::endl;
```

```
// 测试用例 3: 只有一个节点的树
TreeNode* root3 = new TreeNode(1);
std::cout << "测试用例 3（单节点树）:" << std::endl;
std::cout << "解法一（序列化）结果: ";
printResult(solution.findDuplicateSubtrees(root3)); // 预期输出: []
std::cout << "解法二（ID 优化）结果: ";
printResult(solution.findDuplicateSubtreesOptimized(root3)); // 预期输出: []
std::cout << std::endl;

// 释放内存
deleteTree(root3);

// 测试用例 4: 所有节点都相同的树
TreeNode* root4 = new TreeNode(0);
root4->left = new TreeNode(0);
root4->right = new TreeNode(0);
root4->left->left = new TreeNode(0);
root4->right->right = new TreeNode(0);
std::cout << "测试用例 4（所有节点都相同）:" << std::endl;
std::cout << "解法一（序列化）结果: ";
printResult(solution.findDuplicateSubtrees(root4)); // 预期输出类似: [0,0]

// 重新构建树
TreeNode* root4Again = new TreeNode(0);
root4Again->left = new TreeNode(0);
root4Again->right = new TreeNode(0);
root4Again->left->left = new TreeNode(0);
root4Again->right->right = new TreeNode(0);
std::cout << "解法二（ID 优化）结果: ";
printResult(solution.findDuplicateSubtreesOptimized(root4Again)); // 预期输出类似: [0,0]
std::cout << std::endl;

// 释放内存
deleteTree(root4);
deleteTree(root4Again);

// 性能测试 - 构建一个较大的树，其中有重复子树
std::cout << "性能测试:" << std::endl;
// 构建一个平衡树，其中有重复子树
TreeNode* balancedTree = buildBalancedTreeWithDuplicates(1, 7);

auto startTime = std::chrono::high_resolution_clock::now();
std::vector<TreeNode*> result1 = solution.findDuplicateSubtrees(balancedTree);
```

```
    auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
    std::cout << "解法一（序列化）- 找到的重复子树数量: " << result1.size() << std::endl;
    std::cout << "解法一（序列化）- 耗时: " << duration.count() << "ms" << std::endl;

    // 重新构建树
    TreeNode* balancedTreeAgain = buildBalancedTreeWithDuplicates(1, 7);
    startTime = std::chrono::high_resolution_clock::now();
    std::vector<TreeNode*> result2 = solution.findDuplicateSubtreesOptimized(balancedTreeAgain);
    endTime = std::chrono::high_resolution_clock::now();
    duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
    std::cout << "解法二（ID优化）- 找到的重复子树数量: " << result2.size() << std::endl;
    std::cout << "解法二（ID优化）- 耗时: " << duration.count() << "ms" << std::endl;

    // 释放内存
    deleteTree(balancedTree);
    deleteTree(balancedTreeAgain);

    return 0;
}

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

文件: Code19_FindDuplicateSubtrees.java
==========================================
import java.util.*;

/**
 * LeetCode 652. 寻找重复的子树 (Find Duplicate Subtrees)
 *
 * 题目描述:
 * 给定一棵二叉树，返回所有重复的子树。
 * 对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。
 * 两棵树重复是指它们具有相同的结构以及相同的结点值。
 *
 * 示例 1:
 *         1
 *        / \
 *       2   3
 *      /   / \
 *     4   2   4
 *        /
 *       4
```

```
 *
 * 输出:
 * [[2,4],[4]]
 *
 * 解释:
 * 上面的二叉树有两个重复子树。
 * 第一个重复子树是 4，如蓝色节点所示。
 * 第二个重复子树是 2 -> 4，如橙色节点所示。
 *
 * 题目链接: https://leetcode.com/problems/find-duplicate-subtrees/
 *
 * 解题思路:
 * 这道题需要我们找出二叉树中所有重复的子树。解决这个问题的关键在于能够唯一地表示每个子树，并能够快速判断是否已经存在相同的子树。
 *
 * 解法: 递归 + 哈希表
 * 1. 对于每个子树，我们需要生成一个唯一标识符，可以通过序列化的方式实现
 * 2. 使用哈希表来记录每个子树标识符出现的次数
 * 3. 当一个子树标识符出现次数为 2 时，将该子树的根节点加入结果列表
 *
 * 时间复杂度: O(n^2)，其中 n 是树中的节点数。在最坏情况下，序列化每个节点需要 O(n)时间，共有 n 个节点。
 * 空间复杂度: O(n^2)，存储所有子树的序列化表示。
 *
 * 优化版本：使用 ID 来代替完整序列化字符串，可以将时间和空间复杂度优化到 O(n)。
 */

// 定义二叉树节点
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

public class Code19_FindDuplicateSubtrees {
```

```
    /**
     * 解法一: 使用序列化 + 哈希表
     * 将每个子树序列化为字符串，然后使用哈希表记录每个子树出现的次数。
     */
    public static List<TreeNode> findDuplicateSubtrees(TreeNode root) {
        List<TreeNode> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        // 哈希表用于记录每个子树序列化字符串出现的次数
        Map<String, Integer> subtreeCount = new HashMap<>();

        // 递归函数，用于序列化子树并检查重复
        serializeAndCheckDuplicates(root, subtreeCount, result);

        return result;
    }

    /**
     * 递归序列化子树并检查重复
     *
     * @param node 当前节点
     * @param subtreeCount 子树计数哈希表
     * @param result 结果列表
     * @return 当前子树的序列化字符串
     */
    private static String serializeAndCheckDuplicates(TreeNode node, Map<String, Integer>
subtreeCount, List<TreeNode> result) {
        if (node == null) {
            return "#"; // 用#表示空节点
        }

        // 序列化当前节点的左子树、当前节点的值、右子树
        String key = node.val + "," +
                     serializeAndCheckDuplicates(node.left, subtreeCount, result) + "," +
                     serializeAndCheckDuplicates(node.right, subtreeCount, result);

        // 获取当前子树出现的次数，如果是第二次出现，则添加到结果中
        subtreeCount.put(key, subtreeCount.getOrDefault(key, 0) + 1);
        if (subtreeCount.get(key) == 2) {
            result.add(node); // 只有当子树出现次数为 2 时添加，避免重复添加
        }
```

```
    return key;
}

/**
 * 解法二: 使用 ID 代替完整序列化字符串（优化版本）
 * 为每个不同的子树分配一个唯一 ID，使用 ID 来标识子树而不是完整的序列化字符串。
 */
public static List<TreeNode> findDuplicateSubtreesOptimized(TreeNode root) {
    List<TreeNode> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    // 哈希表用于将子树的序列化字符串映射到唯一 ID
    Map<String, Integer> subtreeId = new HashMap<>();
    // 哈希表用于记录每个 ID（子树）出现的次数
    Map<Integer, Integer> idCount = new HashMap<>();
    // 当前可用的下一个 ID
    int[] nextId = {1};

    // 递归函数，使用 ID 检查重复子树
    findDuplicatesWithId(root, subtreeId, idCount, nextId, result);

    return result;
}

/**
 * 递归函数，使用 ID 检查重复子树
 *
 * @param node 当前节点
 * @param subtreeId 子树到 ID 的映射
 * @param idCount ID 出现次数的映射
 * @param nextId 下一个可用的 ID
 * @param result 结果列表
 * @return 当前子树的 ID
 */
private static int findDuplicatesWithId(TreeNode node, Map<String, Integer> subtreeId,
                                        Map<Integer, Integer> idCount, int[] nextId,
                                        List<TreeNode> result) {
    if (node == null) {
        return 0; // 空节点的 ID 为 0
    }
```

```
        // 构建当前子树的键
        String key = node.val + "," +
                     findDuplicatesWithId(node.left, subtreeId, idCount, nextId, result) + "," +
                     findDuplicatesWithId(node.right, subtreeId, idCount, nextId, result);

        // 如果当前子树还没有分配 ID，则分配一个新 ID
        int id = subtreeId.computeIfAbsent(key, k -> nextId[0]++);

        // 增加当前 ID 的计数，并在计数为 2 时添加到结果中
        idCount.put(id, idCount.getOrDefault(id, 0) + 1);
        if (idCount.get(id) == 2) {
            result.add(node);
        }

        return id;
    }

    /**
     * 将树转换为字符串表示（用于打印结果）
     */
    public static List<String> treesToString(List<TreeNode> trees) {
        List<String> result = new ArrayList<>();
        for (TreeNode root : trees) {
            StringBuilder sb = new StringBuilder();
            buildTreeString(root, sb);
            result.add(sb.toString());
        }
        return result;
    }

    private static void buildTreeString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("null");
            return;
        }
        sb.append(node.val);
        if (node.left != null || node.right != null) {
            sb.append("[");
            buildTreeString(node.left, sb);
            sb.append(",");
            buildTreeString(node.right, sb);
            sb.append("]");
```

```
        }
    }

    /**
     * 构建测试用例中的树
     */
    public static TreeNode buildExampleTree() {
        // 构建示例中的树
        //       1
        //      / \
        //     2   3
        //    /   / \
        //   4   2   4
        //      /
        //     4
        TreeNode node1 = new TreeNode(1);
        TreeNode node2 = new TreeNode(2);
        TreeNode node3 = new TreeNode(3);
        TreeNode node4 = new TreeNode(4);
        TreeNode node5 = new TreeNode(2);
        TreeNode node6 = new TreeNode(4);
        TreeNode node7 = new TreeNode(4);

        node1.left = node2;
        node1.right = node3;
        node2.left = node4;
        node3.left = node5;
        node3.right = node6;
        node5.left = node7;

        return node1;
    }

    public static void main(String[] args) {
        // 测试用例 1: 示例中的树
        TreeNode root1 = buildExampleTree();
        System.out.println("测试用例 1:");
        System.out.println("解法一（序列化）结果: " +
treesToString(findDuplicateSubtrees(root1))); // 预期输出类似: [2[4,null],4]

        // 重新构建树，因为解法一可能修改了树的状态（虽然这里不会，但为了保险起见）
        TreeNode root1Again = buildExampleTree();
        System.out.println("解法二（ID 优化）结果: " +
```

```
treesToString(findDuplicateSubtreesOptimized(root1Again))); // 预期输出类似: [2[4,null],4]
        System.out.println();

        // 测试用例 2: 空树
        TreeNode root2 = null;
        System.out.println("测试用例 2（空树）:");
        System.out.println("解法一（序列化）结果: " +
treesToString(findDuplicateSubtrees(root2))); // 预期输出: []
        System.out.println("解法二（ID 优化）结果: " +
treesToString(findDuplicateSubtreesOptimized(root2))); // 预期输出: []
        System.out.println();

        // 测试用例 3: 只有一个节点的树
        TreeNode root3 = new TreeNode(1);
        System.out.println("测试用例 3（单节点树）:");
        System.out.println("解法一（序列化）结果: " +
treesToString(findDuplicateSubtrees(root3))); // 预期输出: []
        System.out.println("解法二（ID 优化）结果: " +
treesToString(findDuplicateSubtreesOptimized(root3))); // 预期输出: []
        System.out.println();

        // 测试用例 4: 所有节点都相同的树
        TreeNode root4 = new TreeNode(0);
        root4.left = new TreeNode(0);
        root4.right = new TreeNode(0);
        root4.left.left = new TreeNode(0);
        root4.right.right = new TreeNode(0);
        System.out.println("测试用例 4（所有节点都相同）:");
        System.out.println("解法一（序列化）结果: " +
treesToString(findDuplicateSubtrees(root4))); // 预期输出类似: [0,0]

        // 重新构建树
        TreeNode root4Again = new TreeNode(0);
        root4Again.left = new TreeNode(0);
        root4Again.right = new TreeNode(0);
        root4Again.left.left = new TreeNode(0);
        root4Again.right.right = new TreeNode(0);
        System.out.println("解法二（ID 优化）结果: " +
treesToString(findDuplicateSubtreesOptimized(root4Again))); // 预期输出类似: [0,0]
        System.out.println();

        // 性能测试 - 构建一个较大的树，其中有重复子树
        System.out.println("性能测试:");
```

```
    // 构建一个平衡树，其中有重复子树
    TreeNode balancedTree = buildBalancedTreeWithDuplicates(1, 7);

    long startTime = System.currentTimeMillis();
    List<TreeNode> result1 = findDuplicateSubtrees(balancedTree);
    long endTime = System.currentTimeMillis();
    System.out.println("解法一（序列化）- 找到的重复子树数量: " + result1.size());
    System.out.println("解法一（序列化）- 耗时: " + (endTime - startTime) + "ms");

    // 重新构建树
    TreeNode balancedTreeAgain = buildBalancedTreeWithDuplicates(1, 7);
    startTime = System.currentTimeMillis();
    List<TreeNode> result2 = findDuplicateSubtreesOptimized(balancedTreeAgain);
    endTime = System.currentTimeMillis();
    System.out.println("解法二（ID 优化）- 找到的重复子树数量: " + result2.size());
    System.out.println("解法二（ID 优化）- 耗时: " + (endTime - startTime) + "ms");
}

/**
 * 构建一个带有重复子树的平衡二叉树
 *
 * @param start 起始值
 * @param end 结束值
 * @return 构建的树的根节点
 */
private static TreeNode buildBalancedTreeWithDuplicates(int start, int end) {
    if (start > end) {
        return null;
    }

    int mid = start + (end - start) / 2;
    TreeNode root = new TreeNode(mid);

    // 为了创建重复子树，我们可以使部分子树的值重复
    if (start <= end - 2) {
        root.left = buildBalancedTreeWithDuplicates(start, mid - 1);
        root.right = buildBalancedTreeWithDuplicates(start, mid - 1); // 重复左子树的结构
    } else {
        root.left = buildBalancedTreeWithDuplicates(start, mid - 1);
        root.right = buildBalancedTreeWithDuplicates(mid + 1, end);
    }

    return root;
```

```
    }
}
```

==========

文件: Code19_FindDuplicateSubtrees.py
==========

```
import time
from typing import List, Dict, Optional

"""
LeetCode 652. 寻找重复的子树 (Find Duplicate Subtrees)

题目描述:
给定一棵二叉树，返回所有重复的子树。
对于同一类的重复子树，你只需要返回其中任意一棵的根结点即可。
两棵树重复是指它们具有相同的结构以及相同的结点值。

示例 1:
        1
       / \
      2   3
     /   / \
    4   2   4
       /
      4

输出:
[[2,4],[4]]

解释:
上面的二叉树有两个重复子树。
第一个重复子树是 4，如蓝色节点所示。
第二个重复子树是 2 -> 4，如橙色节点所示。

题目链接: https://leetcode.com/problems/find-duplicate-subtrees/

解题思路:
这道题需要我们找出二叉树中所有重复的子树。解决这个问题的关键在于能够唯一地表示每个子树，并能够快速判断是否已经存在相同的子树。

解法: 递归 + 哈希表
1. 对于每个子树，我们需要生成一个唯一标识符，可以通过序列化的方式实现
```

2. 使用哈希表来记录每个子树标识符出现的次数
3. 当一个子树标识符出现次数为 2 时，将该子树的根节点加入结果列表

时间复杂度: O(n^2)，其中 n 是树中的节点数。在最坏情况下，序列化每个节点需要 O(n)时间，共有 n 个节点。
空间复杂度: O(n^2)，存储所有子树的序列化表示。

优化版本：使用 ID 来代替完整序列化字符串，可以将时间和空间复杂度优化到 O(n)。
"""

```
# 定义二叉树节点
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def findDuplicateSubtrees(self, root: Optional[TreeNode]) -> List[Optional[TreeNode]]:
        """
        解法一: 使用序列化 + 哈希表
        将每个子树序列化为字符串，然后使用哈希表记录每个子树出现的次数。

        Args:
            root: 二叉树的根节点

        Returns:
            包含重复子树根节点的列表
        """
        result = []
        if not root:
            return result

        # 哈希表用于记录每个子树序列化字符串出现的次数
        subtree_count = {}

        # 递归函数，用于序列化子树并检查重复
        def serialize_and_check_duplicates(node):
            if not node:
                return "#"  # 用#表示空节点

            # 序列化当前节点的左子树、当前节点的值、右子树
            key =
```

```
f"{node.val},{serialize_and_check_duplicates(node.left)},{serialize_and_check_duplicates(node.right)}"

            # 获取当前子树出现的次数，如果是第二次出现，则添加到结果中
            subtree_count[key] = subtree_count.get(key, 0) + 1
            if subtree_count[key] == 2:
                result.append(node)  # 只有当子树出现次数为 2 时添加，避免重复添加

            return key

        serialize_and_check_duplicates(root)
        return result

    def findDuplicateSubtreesOptimized(self, root: Optional[TreeNode]) ->
List[Optional[TreeNode]]:
        """
        解法二: 使用 ID 代替完整序列化字符串（优化版本）
        为每个不同的子树分配一个唯一 ID，使用 ID 来标识子树而不是完整的序列化字符串。

        Args:
            root: 二叉树的根节点

        Returns:
            包含重复子树根节点的列表
        """
        result = []
        if not root:
            return result

        # 哈希表用于将子树的序列化字符串映射到唯一 ID
        subtree_id = {}
        # 哈希表用于记录每个 ID（子树）出现的次数
        id_count = {}
        # 当前可用的下一个 ID
        next_id = [1]  # 使用列表作为可变对象

        # 递归函数，使用 ID 检查重复子树
        def find_duplicates_with_id(node):
            if not node:
                return 0  # 空节点的 ID 为 0

            # 构建当前子树的键，使用子树 ID 而不是完整的序列化字符串
            left_id = find_duplicates_with_id(node.left)
```

```
            right_id = find_duplicates_with_id(node.right)
            key = f"{node.val},{left_id},{right_id}"

            # 如果当前子树还没有分配 ID，则分配一个新 ID
            if key not in subtree_id:
                subtree_id[key] = next_id[0]
                next_id[0] += 1

            id = subtree_id[key]

            # 增加当前 ID 的计数，并在计数为 2 时添加到结果中
            id_count[id] = id_count.get(id, 0) + 1
            if id_count[id] == 2:
                result.append(node)

            return id

        find_duplicates_with_id(root)
        return result

"""
将树转换为字符串表示（用于打印结果）
"""
def tree_to_string(root: Optional[TreeNode]) -> str:
    if not root:
        return "null"

    result = str(root.val)
    if root.left or root.right:
        result += f"[{tree_to_string(root.left)},{tree_to_string(root.right)}]"

    return result

"""
打印结果列表
"""
def print_result(result: List[Optional[TreeNode]]) -> None:
    strings = [tree_to_string(node) for node in result]
    print(f"[{', '.join(strings)}]")

"""
构建测试用例中的树
"""
```

```
def build_example_tree() -> Optional[TreeNode]:
    # 构建示例中的树
    #       1
    #      / \
    #     2   3
    #    /   / \
    #   4   2   4
    #      /
    #     4
    node1 = TreeNode(1)
    node2 = TreeNode(2)
    node3 = TreeNode(3)
    node4 = TreeNode(4)
    node5 = TreeNode(2)
    node6 = TreeNode(4)
    node7 = TreeNode(4)

    node1.left = node2
    node1.right = node3
    node2.left = node4
    node3.left = node5
    node3.right = node6
    node5.left = node7

    return node1

"""
构建一个带有重复子树的平衡二叉树

Args:
    start: 起始值
    end: 结束值

Returns:
    构建的树的根节点
"""
def build_balanced_tree_with_duplicates(start: int, end: int) -> Optional[TreeNode]:
    if start > end:
        return None

    mid = start + (end - start) // 2
    root = TreeNode(mid)
```

```
    # 为了创建重复子树，我们可以使部分子树的值重复
    if start <= end - 2:
        root.left = build_balanced_tree_with_duplicates(start, mid - 1)
        root.right = build_balanced_tree_with_duplicates(start, mid - 1)  # 重复左子树的结构
    else:
        root.left = build_balanced_tree_with_duplicates(start, mid - 1)
        root.right = build_balanced_tree_with_duplicates(mid + 1, end)

    return root

# 测试代码
def main():
    solution = Solution()

    # 测试用例 1: 示例中的树
    root1 = build_example_tree()
    print("测试用例 1:")
    print("解法一（序列化）结果: ")
    print_result(solution.findDuplicateSubtrees(root1))  # 预期输出类似: [2[4,null],4]

    # 重新构建树，因为解法一可能修改了树的状态（虽然这里不会，但为了保险起见）
    root1_again = build_example_tree()
    print("解法二（ID 优化）结果: ")
    print_result(solution.findDuplicateSubtreesOptimized(root1_again))  # 预期输出类似:
[2[4,null],4]
    print()

    # 测试用例 2: 空树
    root2 = None
    print("测试用例 2（空树）:")
    print("解法一（序列化）结果: ")
    print_result(solution.findDuplicateSubtrees(root2))  # 预期输出: []
    print("解法二（ID 优化）结果: ")
    print_result(solution.findDuplicateSubtreesOptimized(root2))  # 预期输出: []
    print()

    # 测试用例 3: 只有一个节点的树
    root3 = TreeNode(1)
    print("测试用例 3（单节点树）:")
    print("解法一（序列化）结果: ")
    print_result(solution.findDuplicateSubtrees(root3))  # 预期输出: []
    print("解法二（ID 优化）结果: ")
    print_result(solution.findDuplicateSubtreesOptimized(root3))  # 预期输出: []
```

```
    print()

    # 测试用例 4: 所有节点都相同的树
    root4 = TreeNode(0)
    root4.left = TreeNode(0)
    root4.right = TreeNode(0)
    root4.left.left = TreeNode(0)
    root4.right.right = TreeNode(0)
    print("测试用例 4（所有节点都相同）:")
    print("解法一（序列化）结果: ")
    print_result(solution.findDuplicateSubtrees(root4))  # 预期输出类似: [0,0]

    # 重新构建树
    root4_again = TreeNode(0)
    root4_again.left = TreeNode(0)
    root4_again.right = TreeNode(0)
    root4_again.left.left = TreeNode(0)
    root4_again.right.right = TreeNode(0)
    print("解法二（ID 优化）结果: ")
    print_result(solution.findDuplicateSubtreesOptimized(root4_again))  # 预期输出类似: [0,0]
    print()

    # 性能测试 - 构建一个较大的树，其中有重复子树
    print("性能测试:")
    # 构建一个平衡树，其中有重复子树
    balanced_tree = build_balanced_tree_with_duplicates(1, 7)

    start_time = time.time()
    result1 = solution.findDuplicateSubtrees(balanced_tree)
    end_time = time.time()
    print(f"解法一（序列化）- 找到的重复子树数量: {len(result1)}")
    print(f"解法一（序列化）- 耗时: {(end_time - start_time) * 1000:.2f}ms")

    # 重新构建树
    balanced_tree_again = build_balanced_tree_with_duplicates(1, 7)
    start_time = time.time()
    result2 = solution.findDuplicateSubtreesOptimized(balanced_tree_again)
    end_time = time.time()
    print(f"解法二（ID 优化）- 找到的重复子树数量: {len(result2)}")
    print(f"解法二（ID 优化）- 耗时: {(end_time - start_time) * 1000:.2f}ms")

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

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

文件: Code19_SubarraysWithKDifferentIntegers.cpp
==========================================
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <functional>

using namespace std;

/**
 * 992. K 个不同整数的子数组
 * 给定一个正整数数组 nums 和一个整数 k，返回 nums 中「好子数组」的数目。
 * 如果某个子数组中不同整数的个数恰好为 k，则称其为「好子数组」。
 *
 * 解题思路：
 * 使用滑动窗口的变种：恰好K个不同整数的子数组数量 = 最多K个不同整数的子数组数量 - 最多K-1个不同整数的子数组数量
 *
 * 时间复杂度：O(n)，其中n是数组长度
 * 空间复杂度：O(k)，用于存储不同整数的哈希表
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/subarrays-with-k-different-integers/
 */
class Solution {
public:
    /**
     * 计算恰好包含K个不同整数的子数组数量
     *
     * @param nums 输入数组
     * @param k 不同整数的个数
     * @return 恰好包含K个不同整数的子数组数量
     */
    int subarraysWithKDistinct(vector<int>& nums, int k) {
        // 恰好K个不同 = 最多K个不同 - 最多K-1个不同
        return atMostKDistinct(nums, k) - atMostKDistinct(nums, k - 1);
    }
```

```
private:
    /**
     * 计算最多包含 K 个不同整数的子数组数量
     *
     * @param nums 输入数组
     * @param k 最多不同整数的个数
     * @return 最多包含 K 个不同整数的子数组数量
     */
    int atMostKDistinct(vector<int>& nums, int k) {
        if (k < 0) {
            return 0;
        }

        int n = nums.size();
        int count = 0; // 子数组数量
        int left = 0; // 窗口左边界
        unordered_map<int, int> freq; // 记录每个数字的出现频率

        // 滑动窗口右边界
        for (int right = 0; right < n; right++) {
            // 添加右边界元素
            freq[nums[right]]++;

            // 如果不同数字数量超过 k，收缩左边界
            while (freq.size() > k) {
                // 移除左边界元素
                freq[nums[left]]--;
                if (freq[nums[left]] == 0) {
                    freq.erase(nums[left]);
                }
                left++;
            }

            // 以 right 结尾的，满足条件的子数组数量为 right - left + 1
            count += right - left + 1;
        }

        return count;
    }
};

/**
 * 直接解法：使用双指针和哈希表
```

```
 * 时间复杂度：O(n)，空间复杂度：O(k)
 */
class SolutionDirect {
public:
    int subarraysWithKDistinct(vector<int>& nums, int k) {
        int n = nums.size();
        int count = 0;

        // 记录每个数字最后一次出现的位置
        unordered_map<int, int> lastSeen;
        int left = 0; // 窗口左边界
        int right = 0; // 窗口右边界

        while (right < n) {
            // 更新当前数字的最后出现位置
            lastSeen[nums[right]] = right;

            // 如果不同数字数量超过 k，移动左边界
            while (lastSeen.size() > k) {
                // 如果左边界数字的最后出现位置就是当前位置，从 map 中移除
                if (lastSeen[nums[left]] == left) {
                    lastSeen.erase(nums[left]);
                }
                left++;
            }

            // 如果恰好有 k 个不同数字，计算以 right 结尾的子数组数量
            if (lastSeen.size() == k) {
                // 找到最小的位置，使得从该位置到 right 的子数组恰好有 k 个不同数字
                int minIndex = right;
                for (auto& pair : lastSeen) {
                    minIndex = min(minIndex, pair.second);
                }
                count += minIndex - left + 1;
            }

            right++;
        }

        return count;
    }
};
```

```
/**
 * 优化版本：使用数组代替哈希表（当数字范围有限时）
 * 时间复杂度：O(n)，空间复杂度：O(max_value)
 */
class SolutionOptimized {
public:
    int subarraysWithKDistinct(vector<int>& nums, int k) {
        int n = nums.size();
        if (n == 0 || k == 0) {
            return 0;
        }

        // 找到数组中的最大值，用于确定数组大小
        int maxVal = 0;
        for (int num : nums) {
            maxVal = max(maxVal, num);
        }

        vector<int> freq(maxVal + 1, 0); // 频率数组
        int distinct = 0; // 当前不同数字的数量
        int count = 0;
        int left = 0;

        // 使用双指针技巧
        for (int right = 0; right < n; right++) {
            // 添加右边界元素
            if (freq[nums[right]] == 0) {
                distinct++;
            }
            freq[nums[right]]++;

            // 收缩左边界，直到不同数字数量不超过 k
            while (distinct > k) {
                freq[nums[left]]--;
                if (freq[nums[left]] == 0) {
                    distinct--;
                }
                left++;
            }

            // 如果恰好有 k 个不同数字，计算数量
            if (distinct == k) {
                int tempLeft = left;
```

```
                int tempDistinct = distinct;
                vector<int> tempFreq = freq; // 复制频率数组

                // 计算以 right 结尾的恰好 k 个不同的子数组数量
                while (tempDistinct == k) {
                    count++;
                    tempFreq[nums[tempLeft]]--;
                    if (tempFreq[nums[tempLeft]] == 0) {
                        tempDistinct--;
                    }
                    tempLeft++;
                }
            }
        }

        return count;
    }
};

// 测试函数
void testSubarraysWithKDistinct() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {1, 2, 1, 2, 3};
    int k1 = 2;
    int result1 = solution.subarraysWithKDistinct(nums1, k1);
    cout << "输入数组: ";
    for (int num : nums1) cout << num << " ";
    cout << endl;
    cout << "k = " << k1 << endl;
    cout << "恰好包含" << k1 << "个不同整数的子数组数量: " << result1 << endl;
    cout << "预期: 7" << endl;
    cout << endl;

    // 测试用例 2
    vector<int> nums2 = {1, 2, 1, 3, 4};
    int k2 = 3;
    int result2 = solution.subarraysWithKDistinct(nums2, k2);
    cout << "输入数组: ";
    for (int num : nums2) cout << num << " ";
    cout << endl;
    cout << "k = " << k2 << endl;
```

```
    cout << "恰好包含" << k2 << "个不同整数的子数组数量: " << result2 << endl;
    cout << "预期: 3" << endl;
    cout << endl;

    // 测试用例 3：边界情况，k=0
    vector<int> nums3 = {1, 2, 3};
    int k3 = 0;
    int result3 = solution.subarraysWithKDistinct(nums3, k3);
    cout << "输入数组: ";
    for (int num : nums3) cout << num << " ";
    cout << endl;
    cout << "k = " << k3 << endl;
    cout << "恰好包含" << k3 << "个不同整数的子数组数量: " << result3 << endl;
    cout << "预期: 0" << endl;
    cout << endl;

    // 测试用例 4：k=1
    vector<int> nums4 = {1, 1, 1, 2, 2, 3};
    int k4 = 1;
    int result4 = solution.subarraysWithKDistinct(nums4, k4);
    cout << "输入数组: ";
    for (int num : nums4) cout << num << " ";
    cout << endl;
    cout << "k = " << k4 << endl;
    cout << "恰好包含" << k4 << "个不同整数的子数组数量: " << result4 << endl;
    cout << "预期: 9" << endl;
    cout << endl;

    // 测试用例 5：k 等于数组长度
    vector<int> nums5 = {1, 2, 3, 4, 5};
    int k5 = 5;
    int result5 = solution.subarraysWithKDistinct(nums5, k5);
    cout << "输入数组: ";
    for (int num : nums5) cout << num << " ";
    cout << endl;
    cout << "k = " << k5 << endl;
    cout << "恰好包含" << k5 << "个不同整数的子数组数量: " << result5 << endl;
    cout << "预期: 1" << endl;
}

int main() {
    testSubarraysWithKDistinct();
    return 0;
```

```
}

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

文件: Code19_SubarraysWithKDifferentIntegers.java
==========================================
package class049;

import java.util.*;

/**
 * 992. K 个不同整数的子数组
 * 给定一个正整数数组 nums 和一个整数 k，返回 nums 中「好子数组」的数目。
 * 如果某个子数组中不同整数的个数恰好为 k，则称其为「好子数组」。
 *
 * 解题思路：
 * 使用滑动窗口的变种：恰好 K 个不同整数的子数组数量 = 最多 K 个不同整数的子数组数量 - 最多 K-1 个不同整数的子数组数量
 *
 * 时间复杂度：O(n)，其中 n 是数组长度
 * 空间复杂度：O(k)，用于存储不同整数的哈希表
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/subarrays-with-k-different-integers/
 */
public class Code19_SubarraysWithKDifferentIntegers {

    /**
     * 计算恰好包含 K 个不同整数的子数组数量
     *
     * @param nums 输入数组
     * @param k 不同整数的个数
     * @return 恰好包含 K 个不同整数的子数组数量
     */
    public static int subarraysWithKDistinct(int[] nums, int k) {
        // 恰好 K 个不同 = 最多 K 个不同 - 最多 K-1 个不同
        return atMostKDistinct(nums, k) - atMostKDistinct(nums, k - 1);
    }

    /**
     * 计算最多包含 K 个不同整数的子数组数量
     *
```

```
     * @param nums 输入数组
     * @param k 最多不同整数的个数
     * @return 最多包含 K 个不同整数的子数组数量
     */
    private static int atMostKDistinct(int[] nums, int k) {
        if (k < 0) {
            return 0;
        }

        int n = nums.length;
        int count = 0; // 子数组数量
        int left = 0; // 窗口左边界
        Map<Integer, Integer> freq = new HashMap<>(); // 记录每个数字的出现频率

        // 滑动窗口右边界
        for (int right = 0; right < n; right++) {
            // 添加右边界元素
            freq.put(nums[right], freq.getOrDefault(nums[right], 0) + 1);

            // 如果不同数字数量超过 k，收缩左边界
            while (freq.size() > k) {
                // 移除左边界元素
                freq.put(nums[left], freq.get(nums[left]) - 1);
                if (freq.get(nums[left]) == 0) {
                    freq.remove(nums[left]);
                }
                left++;
            }

            // 以 right 结尾的，满足条件的子数组数量为 right - left + 1
            count += right - left + 1;
        }

        return count;
    }

    /**
     * 直接解法：使用双指针和哈希表
     * 时间复杂度：O(n)，空间复杂度：O(k)
     */
    public static int subarraysWithKDistinctDirect(int[] nums, int k) {
        int n = nums.length;
        int count = 0;
```

```
    // 记录每个数字最后一次出现的位置
    Map<Integer, Integer> lastSeen = new HashMap<>();
    int left = 0; // 窗口左边界
    int right = 0; // 窗口右边界

    while (right < n) {
        // 更新当前数字的最后出现位置
        lastSeen.put(nums[right], right);

        // 如果不同数字数量超过 k，移动左边界
        while (lastSeen.size() > k) {
            // 如果左边界数字的最后出现位置就是当前位置，从 map 中移除
            if (lastSeen.get(nums[left]) == left) {
                lastSeen.remove(nums[left]);
            }
            left++;
        }

        // 如果恰好有 k 个不同数字，计算以 right 结尾的子数组数量
        if (lastSeen.size() == k) {
            // 找到最小的位置，使得从该位置到 right 的子数组恰好有 k 个不同数字
            int minIndex = right;
            for (int index : lastSeen.values()) {
                minIndex = Math.min(minIndex, index);
            }
            count += minIndex - left + 1;
        }

        right++;
    }

    return count;
}

/**
 * 优化版本：使用数组代替哈希表（当数字范围有限时）
 * 时间复杂度：O(n)，空间复杂度：O(max_value)
 */
public static int subarraysWithKDistinctOptimized(int[] nums, int k) {
    int n = nums.length;
    if (n == 0 || k == 0) {
        return 0;
```

```
}

// 找到数组中的最大值，用于确定数组大小
int maxVal = 0;
for (int num : nums) {
    maxVal = Math.max(maxVal, num);
}

int[] freq = new int[maxVal + 1]; // 频率数组
int distinct = 0; // 当前不同数字的数量
int count = 0;
int left = 0;

// 使用双指针技巧
for (int right = 0; right < n; right++) {
    // 添加右边界元素
    if (freq[nums[right]] == 0) {
        distinct++;
    }
    freq[nums[right]]++;

    // 收缩左边界，直到不同数字数量不超过 k
    while (distinct > k) {
        freq[nums[left]]--;
        if (freq[nums[left]] == 0) {
            distinct--;
        }
        left++;
    }

    // 如果恰好有 k 个不同数字，计算数量
    if (distinct == k) {
        int tempLeft = left;
        int tempDistinct = distinct;
        int[] tempFreq = freq.clone();

        // 计算以 right 结尾的恰好 k 个不同的子数组数量
        while (tempDistinct == k) {
            count++;
            tempFreq[nums[tempLeft]]--;
            if (tempFreq[nums[tempLeft]] == 0) {
                tempDistinct--;
            }
```

```
                tempLeft++;
            }
        }
    }

    return count;
}

// 测试用例
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 2, 1, 2, 3};
    int k1 = 2;
    int result1 = subarraysWithKDistinct(nums1, k1);
    System.out.println("输入数组: " + Arrays.toString(nums1));
    System.out.println("k = " + k1);
    System.out.println("恰好包含" + k1 + "个不同整数的子数组数量: " + result1);
    System.out.println("预期: 7");
    System.out.println();

    // 测试用例 2
    int[] nums2 = {1, 2, 1, 3, 4};
    int k2 = 3;
    int result2 = subarraysWithKDistinct(nums2, k2);
    System.out.println("输入数组: " + Arrays.toString(nums2));
    System.out.println("k = " + k2);
    System.out.println("恰好包含" + k2 + "个不同整数的子数组数量: " + result2);
    System.out.println("预期: 3");
    System.out.println();

    // 测试用例 3：边界情况，k=0
    int[] nums3 = {1, 2, 3};
    int k3 = 0;
    int result3 = subarraysWithKDistinct(nums3, k3);
    System.out.println("输入数组: " + Arrays.toString(nums3));
    System.out.println("k = " + k3);
    System.out.println("恰好包含" + k3 + "个不同整数的子数组数量: " + result3);
    System.out.println("预期: 0");
    System.out.println();

    // 测试用例 4：k=1
    int[] nums4 = {1, 1, 1, 2, 2, 3};
    int k4 = 1;
```

```
        int result4 = subarraysWithKDistinct(nums4, k4);
        System.out.println("输入数组: " + Arrays.toString(nums4));
        System.out.println("k = " + k4);
        System.out.println("恰好包含" + k4 + "个不同整数的子数组数量: " + result4);
        System.out.println("预期: 9");
        System.out.println();

        // 测试用例 5：k 等于数组长度
        int[] nums5 = {1, 2, 3, 4, 5};
        int k5 = 5;
        int result5 = subarraysWithKDistinct(nums5, k5);
        System.out.println("输入数组: " + Arrays.toString(nums5));
        System.out.println("k = " + k5);
        System.out.println("恰好包含" + k5 + "个不同整数的子数组数量: " + result5);
        System.out.println("预期: 1");
    }
}
```

==========

文件: Code19_SubarraysWithKDifferentIntegers.py

==========

```
from typing import List
from collections import defaultdict

class Solution:
    """
    992. K 个不同整数的子数组
    给定一个正整数数组 nums 和一个整数 k，返回 nums 中「好子数组」的数目。
    如果某个子数组中不同整数的个数恰好为 k，则称其为「好子数组」。

    解题思路：
    使用滑动窗口的变种：恰好 K 个不同整数的子数组数量 = 最多 K 个不同整数的子数组数量 - 最多 K-1 个不同整数的子数组数量

    时间复杂度：O(n)，其中 n 是数组长度
    空间复杂度：O(k)，用于存储不同整数的哈希表

    是否最优解：是

    测试链接：https://leetcode.cn/problems/subarrays-with-k-different-integers/
    """
```

```
def subarraysWithKDistinct(self, nums: List[int], k: int) -> int:
    """
    计算恰好包含 K 个不同整数的子数组数量

    Args:
        nums: 输入数组
        k: 不同整数的个数

    Returns:
        恰好包含 K 个不同整数的子数组数量
    """
    # 恰好 K 个不同 = 最多 K 个不同 - 最多 K-1 个不同
    return self.at_most_k_distinct(nums, k) - self.at_most_k_distinct(nums, k - 1)

def at_most_k_distinct(self, nums: List[int], k: int) -> int:
    """
    计算最多包含 K 个不同整数的子数组数量

    Args:
        nums: 输入数组
        k: 最多不同整数的个数

    Returns:
        最多包含 K 个不同整数的子数组数量
    """
    if k < 0:
        return 0

    n = len(nums)
    count = 0  # 子数组数量
    left = 0   # 窗口左边界
    freq = defaultdict(int)  # 记录每个数字的出现频率

    # 滑动窗口右边界
    for right in range(n):
        # 添加右边界元素
        freq[nums[right]] += 1

        # 如果不同数字数量超过 k，收缩左边界
        while len(freq) > k:
            # 移除左边界元素
            freq[nums[left]] -= 1
            if freq[nums[left]] == 0:
```

```
                del freq[nums[left]]
            left += 1

        # 以 right 结尾的，满足条件的子数组数量为 right - left + 1
        count += right - left + 1

    return count


class SolutionDirect:
    """
    直接解法：使用双指针和哈希表
    时间复杂度：O(n)，空间复杂度：O(k)
    """

    def subarraysWithKDistinct(self, nums: List[int], k: int) -> int:
        n = len(nums)
        count = 0

        # 记录每个数字最后一次出现的位置
        last_seen = {}
        left = 0  # 窗口左边界
        right = 0  # 窗口右边界

        while right < n:
            # 更新当前数字的最后出现位置
            last_seen[nums[right]] = right

            # 如果不同数字数量超过 k，移动左边界
            while len(last_seen) > k:
                # 如果左边界数字的最后出现位置就是当前位置，从 map 中移除
                if last_seen.get(nums[left]) == left:
                    del last_seen[nums[left]]
                left += 1

            # 如果恰好有 k 个不同数字，计算以 right 结尾的子数组数量
            if len(last_seen) == k:
                # 找到最小的位置，使得从该位置到 right 的子数组恰好有 k 个不同数字
                min_index = right
                for index in last_seen.values():
                    min_index = min(min_index, index)
                count += min_index - left + 1
```

```
            right += 1

        return count


class SolutionOptimized:
    """
    优化版本：使用数组代替哈希表（当数字范围有限时）
    时间复杂度：O(n)，空间复杂度：O(max_value)
    """

    def subarraysWithKDistinct(self, nums: List[int], k: int) -> int:
        n = len(nums)
        if n == 0 or k == 0:
            return 0

        # 找到数组中的最大值，用于确定数组大小
        max_val = max(nums) if nums else 0

        freq = [0] * (max_val + 1)  # 频率数组
        distinct = 0  # 当前不同数字的数量
        count = 0
        left = 0

        # 使用双指针技巧
        for right in range(n):
            # 添加右边界元素
            if freq[nums[right]] == 0:
                distinct += 1
            freq[nums[right]] += 1

            # 收缩左边界，直到不同数字数量不超过 k
            while distinct > k:
                freq[nums[left]] -= 1
                if freq[nums[left]] == 0:
                    distinct -= 1
                left += 1

            # 如果恰好有 k 个不同数字，计算数量
            if distinct == k:
                temp_left = left
                temp_distinct = distinct
                temp_freq = freq.copy()  # 复制频率数组
```

```
                # 计算以 right 结尾的恰好 k 个不同的子数组数量
                while temp_distinct == k:
                    count += 1
                    temp_freq[nums[temp_left]] -= 1
                    if temp_freq[nums[temp_left]] == 0:
                        temp_distinct -= 1
                    temp_left += 1

        return count


def test_subarrays_with_k_distinct():
    """
    测试函数
    """
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 2, 1, 2, 3]
    k1 = 2
    result1 = solution.subarraysWithKDistinct(nums1, k1)
    print(f"输入数组: {nums1}")
    print(f"k = {k1}")
    print(f"恰好包含{k1}个不同整数的子数组数量: {result1}")
    print("预期: 7")
    print()

    # 测试用例 2
    nums2 = [1, 2, 1, 3, 4]
    k2 = 3
    result2 = solution.subarraysWithKDistinct(nums2, k2)
    print(f"输入数组: {nums2}")
    print(f"k = {k2}")
    print(f"恰好包含{k2}个不同整数的子数组数量: {result2}")
    print("预期: 3")
    print()

    # 测试用例 3：边界情况，k=0
    nums3 = [1, 2, 3]
    k3 = 0
    result3 = solution.subarraysWithKDistinct(nums3, k3)
    print(f"输入数组: {nums3}")
```

```
    print(f"k = {k3}")
    print(f"恰好包含{k3}个不同整数的子数组数量: {result3}")
    print("预期: 0")
    print()

    # 测试用例 4：k=1
    nums4 = [1, 1, 1, 2, 2, 3]
    k4 = 1
    result4 = solution.subarraysWithKDistinct(nums4, k4)
    print(f"输入数组: {nums4}")
    print(f"k = {k4}")
    print(f"恰好包含{k4}个不同整数的子数组数量: {result4}")
    print("预期: 9")
    print()

    # 测试用例 5：k 等于数组长度
    nums5 = [1, 2, 3, 4, 5]
    k5 = 5
    result5 = solution.subarraysWithKDistinct(nums5, k5)
    print(f"输入数组: {nums5}")
    print(f"k = {k5}")
    print(f"恰好包含{k5}个不同整数的子数组数量: {result5}")
    print("预期: 1")


if __name__ == "__main__":
    test_subarrays_with_k_distinct()

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

文件: Code20_LongestSubarrayOf1sAfterDeletingOneElement.cpp
================================================
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/**
 * 1493. 删掉一个元素以后全为 1 的最长子数组
 * 给你一个二进制数组 nums ，你需要从中删掉一个元素。
 * 请你在删掉元素的结果数组中，返回最长的且只包含 1 的非空子数组的长度。
 * 如果不存在这样的子数组，请返回 0 。
```

```
 *
 * 解题思路：
 * 使用滑动窗口维护一个最多包含 1 个 0 的窗口
 * 当窗口内 0 的个数超过 1 时，收缩左边界
 * 最终结果是窗口大小减 1（因为要删除一个元素）
 *
 * 时间复杂度：O(n)，其中 n 是数组长度
 * 空间复杂度：O(1)
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/longest-subarray-of-1s-after-deleting-one-element/
 */
class Solution {
public:
    /**
     * 计算删掉一个元素后全为 1 的最长子数组长度
     *
     * @param nums 二进制数组
     * @return 最长子数组长度
     */
    int longestSubarray(vector<int>& nums) {
        int n = nums.size();
        int maxLength = 0; // 最大长度
        int left = 0; // 窗口左边界
        int zeroCount = 0; // 窗口内 0 的个数

        // 滑动窗口右边界
        for (int right = 0; right < n; right++) {
            // 如果当前元素是 0，增加 0 的计数
            if (nums[right] == 0) {
                zeroCount++;
            }

            // 如果窗口内 0 的个数超过 1，收缩左边界
            while (zeroCount > 1) {
                if (nums[left] == 0) {
                    zeroCount--;
                }
                left++;
            }

            // 更新最大长度（窗口大小减 1，因为要删除一个元素）
```

```
        maxLength = max(maxLength, right - left);
    }

    return maxLength;
}

/**
 * 优化版本：使用更简洁的写法
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
int longestSubarrayOptimized(vector<int>& nums) {
    int n = nums.size();
    int maxLength = 0;
    int left = 0;
    int zeroCount = 0;

    for (int right = 0; right < n; right++) {
        zeroCount += 1 - nums[right]; // 如果 nums[right]是 0，则 zeroCount 加 1

        while (zeroCount > 1) {
            zeroCount -= 1 - nums[left]; // 如果 nums[left]是 0，则 zeroCount 减 1
            left++;
        }

        maxLength = max(maxLength, right - left);
    }

    return maxLength;
}

/**
 * 另一种思路：计算连续 1 的段，然后考虑删除中间的一个 0 来连接两段 1
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
int longestSubarrayAlternative(vector<int>& nums) {
    int n = nums.size();
    int maxLength = 0;
    int prev = 0; // 前一段连续 1 的长度
    int curr = 0; // 当前连续 1 的长度
    bool hasZero = false; // 是否包含 0

    for (int i = 0; i < n; i++) {
        if (nums[i] == 1) {
```

```
                curr++;
            } else {
                hasZero = true;
                // 遇到0时，可以删除这个0来连接prev和curr
                maxLength = max(maxLength, prev + curr);
                prev = curr;
                curr = 0;
            }
        }

        // 处理最后一段
        maxLength = max(maxLength, prev + curr);

        // 如果整个数组都是1，需要删除一个元素
        if (!hasZero) {
            return n - 1;
        }

        return maxLength;
    }
};

// 测试函数
void testLongestSubarray() {
    Solution solution;

    // 测试用例1
    vector<int> nums1 = {1, 1, 0, 1};
    int result1 = solution.longestSubarray(nums1);
    cout << "输入数组: ";
    for (int num : nums1) cout << num << " ";
    cout << endl;
    cout << "最长子数组长度: " << result1 << endl;
    cout << "预期: 3" << endl;
    cout << endl;

    // 测试用例2
    vector<int> nums2 = {0, 1, 1, 1, 0, 1, 1, 0, 1};
    int result2 = solution.longestSubarray(nums2);
    cout << "输入数组: ";
    for (int num : nums2) cout << num << " ";
    cout << endl;
    cout << "最长子数组长度: " << result2 << endl;
```

```
    cout << "预期: 5" << endl;
    cout << endl;

    // 测试用例 3：全是 1
    vector<int> nums3 = {1, 1, 1};
    int result3 = solution.longestSubarray(nums3);
    cout << "输入数组: ";
    for (int num : nums3) cout << num << " ";
    cout << endl;
    cout << "最长子数组长度: " << result3 << endl;
    cout << "预期: 2" << endl;
    cout << endl;

    // 测试用例 4：全是 0
    vector<int> nums4 = {0, 0, 0};
    int result4 = solution.longestSubarray(nums4);
    cout << "输入数组: ";
    for (int num : nums4) cout << num << " ";
    cout << endl;
    cout << "最长子数组长度: " << result4 << endl;
    cout << "预期: 0" << endl;
    cout << endl;

    // 测试用例 5：边界情况，单个元素
    vector<int> nums5 = {1};
    int result5 = solution.longestSubarray(nums5);
    cout << "输入数组: ";
    for (int num : nums5) cout << num << " ";
    cout << endl;
    cout << "最长子数组长度: " << result5 << endl;
    cout << "预期: 0" << endl;
    cout << endl;

    // 测试用例 6：交替的 0 和 1
    vector<int> nums6 = {1, 0, 1, 0, 1};
    int result6 = solution.longestSubarray(nums6);
    cout << "输入数组: ";
    for (int num : nums6) cout << num << " ";
    cout << endl;
    cout << "最长子数组长度: " << result6 << endl;
    cout << "预期: 2" << endl;
}
```

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

==========

文件: Code20_LongestSubarrayOf1sAfterDeletingOneElement.java

==========

```
package class049;

/**
 * 1493. 删掉一个元素以后全为 1 的最长子数组
 * 给你一个二进制数组 nums ，你需要从中删掉一个元素。
 * 请你在删掉元素的结果数组中，返回最长的且只包含 1 的非空子数组的长度。
 * 如果不存在这样的子数组，请返回 0 。
 *
 * 解题思路：
 * 使用滑动窗口维护一个最多包含 1 个 0 的窗口
 * 当窗口内 0 的个数超过 1 时，收缩左边界
 * 最终结果是窗口大小减 1（因为要删除一个元素）
 *
 * 时间复杂度：O(n)，其中 n 是数组长度
 * 空间复杂度：O(1)
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/longest-subarray-of-1s-after-deleting-one-element/
 */
public class Code20_LongestSubarrayOf1sAfterDeletingOneElement {

    /**
     * 计算删掉一个元素后全为 1 的最长子数组长度
     *
     * @param nums 二进制数组
     * @return 最长子数组长度
     */
    public static int longestSubarray(int[] nums) {
        int n = nums.length;
        int maxLength = 0; // 最大长度
        int left = 0; // 窗口左边界
        int zeroCount = 0; // 窗口内 0 的个数
```

```
    // 滑动窗口右边界
    for (int right = 0; right < n; right++) {
        // 如果当前元素是0，增加0的计数
        if (nums[right] == 0) {
            zeroCount++;
        }

        // 如果窗口内0的个数超过1，收缩左边界
        while (zeroCount > 1) {
            if (nums[left] == 0) {
                zeroCount--;
            }
            left++;
        }

        // 更新最大长度（窗口大小减1，因为要删除一个元素）
        maxLength = Math.max(maxLength, right - left);
    }

    return maxLength;
}

/**
 * 优化版本：使用更简洁的写法
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
public static int longestSubarrayOptimized(int[] nums) {
    int n = nums.length;
    int maxLength = 0;
    int left = 0;
    int zeroCount = 0;

    for (int right = 0; right < n; right++) {
        zeroCount += 1 - nums[right]; // 如果nums[right]是0，则zeroCount加1

        while (zeroCount > 1) {
            zeroCount -= 1 - nums[left]; // 如果nums[left]是0，则zeroCount减1
            left++;
        }

        maxLength = Math.max(maxLength, right - left);
    }
```

```
    return maxLength;
}

/**
 * 另一种思路：计算连续1的段，然后考虑删除中间的一个0来连接两段1
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
public static int longestSubarrayAlternative(int[] nums) {
    int n = nums.length;
    int maxLength = 0;
    int prev = 0; // 前一段连续1的长度
    int curr = 0; // 当前连续1的长度
    boolean hasZero = false; // 是否包含0

    for (int i = 0; i < n; i++) {
        if (nums[i] == 1) {
            curr++;
        } else {
            hasZero = true;
            // 遇到0时，可以删除这个0来连接prev和curr
            maxLength = Math.max(maxLength, prev + curr);
            prev = curr;
            curr = 0;
        }
    }

    // 处理最后一段
    maxLength = Math.max(maxLength, prev + curr);

    // 如果整个数组都是1，需要删除一个元素
    if (!hasZero) {
        return n - 1;
    }

    return maxLength;
}

// 测试用例
public static void main(String[] args) {
    // 测试用例1
    int[] nums1 = {1, 1, 0, 1};
    int result1 = longestSubarray(nums1);
    System.out.println("输入数组: " + java.util.Arrays.toString(nums1));
```

```
        System.out.println("最长子数组长度: " + result1);
        System.out.println("预期: 3");
        System.out.println();

        // 测试用例 2
        int[] nums2 = {0, 1, 1, 1, 0, 1, 1, 0, 1};
        int result2 = longestSubarray(nums2);
        System.out.println("输入数组: " + java.util.Arrays.toString(nums2));
        System.out.println("最长子数组长度: " + result2);
        System.out.println("预期: 5");
        System.out.println();

        // 测试用例 3：全是 1
        int[] nums3 = {1, 1, 1};
        int result3 = longestSubarray(nums3);
        System.out.println("输入数组: " + java.util.Arrays.toString(nums3));
        System.out.println("最长子数组长度: " + result3);
        System.out.println("预期: 2");
        System.out.println();

        // 测试用例 4：全是 0
        int[] nums4 = {0, 0, 0};
        int result4 = longestSubarray(nums4);
        System.out.println("输入数组: " + java.util.Arrays.toString(nums4));
        System.out.println("最长子数组长度: " + result4);
        System.out.println("预期: 0");
        System.out.println();

        // 测试用例 5：边界情况，单个元素
        int[] nums5 = {1};
        int result5 = longestSubarray(nums5);
        System.out.println("输入数组: " + java.util.Arrays.toString(nums5));
        System.out.println("最长子数组长度: " + result5);
        System.out.println("预期: 0");
        System.out.println();

        // 测试用例 6：交替的 0 和 1
        int[] nums6 = {1, 0, 1, 0, 1};
        int result6 = longestSubarray(nums6);
        System.out.println("输入数组: " + java.util.Arrays.toString(nums6));
        System.out.println("最长子数组长度: " + result6);
        System.out.println("预期: 2");
    }
```

```
}

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

文件: Code20_LongestSubarrayOf1sAfterDeletingOneElement.py
==========================================
from typing import List

class Solution:
    """
    1493. 删掉一个元素以后全为 1 的最长子数组
    给你一个二进制数组 nums ，你需要从中删掉一个元素。
    请你在删掉元素的结果数组中，返回最长的且只包含 1 的非空子数组的长度。
    如果不存在这样的子数组，请返回 0 。

    解题思路：
    使用滑动窗口维护一个最多包含 1 个 0 的窗口
    当窗口内 0 的个数超过 1 时，收缩左边界
    最终结果是窗口大小减 1（因为要删除一个元素）

    时间复杂度：O(n)，其中 n 是数组长度
    空间复杂度：O(1)

    是否最优解：是

    测试链接：https://leetcode.cn/problems/longest-subarray-of-1s-after-deleting-one-element/
    """

    def longestSubarray(self, nums: List[int]) -> int:
        """
        计算删掉一个元素后全为 1 的最长子数组长度

        Args:
            nums: 二进制数组

        Returns:
            最长子数组长度
        """
        n = len(nums)
        max_length = 0  # 最大长度
        left = 0        # 窗口左边界
        zero_count = 0  # 窗口内 0 的个数
```

```
    # 滑动窗口右边界
    for right in range(n):
        # 如果当前元素是0，增加0的计数
        if nums[right] == 0:
            zero_count += 1

        # 如果窗口内0的个数超过1，收缩左边界
        while zero_count > 1:
            if nums[left] == 0:
                zero_count -= 1
            left += 1

        # 更新最大长度（窗口大小减1，因为要删除一个元素）
        max_length = max(max_length, right - left)

    return max_length

def longestSubarrayOptimized(self, nums: List[int]) -> int:
    """
    优化版本：使用更简洁的写法
    时间复杂度：O(n)，空间复杂度：O(1)
    """
    n = len(nums)
    max_length = 0
    left = 0
    zero_count = 0

    for right in range(n):
        zero_count += 1 - nums[right]  # 如果nums[right]是0，则zero_count加1

        while zero_count > 1:
            zero_count -= 1 - nums[left]  # 如果nums[left]是0，则zero_count减1
            left += 1

        max_length = max(max_length, right - left)

    return max_length

def longestSubarrayAlternative(self, nums: List[int]) -> int:
    """
    另一种思路：计算连续1的段，然后考虑删除中间的一个0来连接两段1
    时间复杂度：O(n)，空间复杂度：O(1)
    """
```

```
        n = len(nums)
        max_length = 0
        prev = 0  # 前一段连续 1 的长度
        curr = 0  # 当前连续 1 的长度
        has_zero = False  # 是否包含 0

        for i in range(n):
            if nums[i] == 1:
                curr += 1
            else:
                has_zero = True
                # 遇到 0 时，可以删除这个 0 来连接 prev 和 curr
                max_length = max(max_length, prev + curr)
                prev = curr
                curr = 0

        # 处理最后一段
        max_length = max(max_length, prev + curr)

        # 如果整个数组都是 1，需要删除一个元素
        if not has_zero:
            return n - 1

        return max_length


def test_longest_subarray():
    """
    测试函数
    """
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 1, 0, 1]
    result1 = solution.longestSubarray(nums1)
    print(f"输入数组: {nums1}")
    print(f"最长子数组长度: {result1}")
    print("预期: 3")
    print()

    # 测试用例 2
    nums2 = [0, 1, 1, 1, 0, 1, 1, 0, 1]
    result2 = solution.longestSubarray(nums2)
```

```
    print(f"输入数组: {nums2}")
    print(f"最长子数组长度: {result2}")
    print("预期: 5")
    print()

    # 测试用例 3：全是 1
    nums3 = [1, 1, 1]
    result3 = solution.longestSubarray(nums3)
    print(f"输入数组: {nums3}")
    print(f"最长子数组长度: {result3}")
    print("预期: 2")
    print()

    # 测试用例 4：全是 0
    nums4 = [0, 0, 0]
    result4 = solution.longestSubarray(nums4)
    print(f"输入数组: {nums4}")
    print(f"最长子数组长度: {result4}")
    print("预期: 0")
    print()

    # 测试用例 5：边界情况，单个元素
    nums5 = [1]
    result5 = solution.longestSubarray(nums5)
    print(f"输入数组: {nums5}")
    print(f"最长子数组长度: {result5}")
    print("预期: 0")
    print()

    # 测试用例 6：交替的 0 和 1
    nums6 = [1, 0, 1, 0, 1]
    result6 = solution.longestSubarray(nums6)
    print(f"输入数组: {nums6}")
    print(f"最长子数组长度: {result6}")
    print("预期: 2")


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

==========

文件: Code21_MaximumErasureValue.cpp

```
==========================================
#include <iostream>
#include <vector>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>

using namespace std;

/**
 * 1695. 删除子数组的最大得分
 * 给你一个正整数数组 nums ，请你从中删除一个含有 若干不同元素 的子数组。删除子数组的 得分 就是子数组各元素之 和 。
 * 返回 只删除一个 子数组可获得的 最大得分 。
 * 如果数组为空，返回 0 。
 *
 * 解题思路：
 * 使用滑动窗口维护一个不含重复元素的子数组
 * 当遇到重复元素时，收缩左边界直到没有重复元素
 * 在滑动过程中记录最大和
 *
 * 时间复杂度：O(n)，其中 n 是数组长度
 * 空间复杂度：O(k)，k 是不同元素的数量
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/maximum-erasure-value/
 */
class Solution {
public:
    /**
     * 计算删除子数组的最大得分
     *
     * @param nums 正整数数组
     * @return 最大得分
     */
    int maximumUniqueSubarray(vector<int>& nums) {
        int n = nums.size();
        int maxScore = 0; // 最大得分
        int currentSum = 0; // 当前窗口的和
        int left = 0; // 窗口左边界
        unordered_set<int> window; // 记录窗口内的元素
```

```
    // 滑动窗口右边界
    for (int right = 0; right < n; right++) {
        // 如果当前元素已经在窗口中，收缩左边界
        while (window.find(nums[right]) != window.end()) {
            currentSum -= nums[left];
            window.erase(nums[left]);
            left++;
        }

        // 添加当前元素到窗口
        window.insert(nums[right]);
        currentSum += nums[right];

        // 更新最大得分
        maxScore = max(maxScore, currentSum);
    }

    return maxScore;
}

/**
 * 优化版本：使用哈希表记录元素最后一次出现的位置
 * 时间复杂度：O(n)，空间复杂度：O(k)
 */
int maximumUniqueSubarrayOptimized(vector<int>& nums) {
    int n = nums.size();
    int maxScore = 0;
    int currentSum = 0;
    int left = 0;
    unordered_map<int, int> lastSeen; // 记录元素最后一次出现的位置

    for (int right = 0; right < n; right++) {
        int num = nums[right];

        // 如果当前元素已经在窗口中，并且位置在 left 之后
        if (lastSeen.find(num) != lastSeen.end() && lastSeen[num] >= left) {
            // 移动左边界到重复元素的下一个位置
            int duplicateIndex = lastSeen[num];
            for (int i = left; i <= duplicateIndex; i++) {
                currentSum -= nums[i];
            }
            left = duplicateIndex + 1;
        }
```

```
        // 更新当前元素的位置
        lastSeen[num] = right;
        currentSum += num;

        // 更新最大得分
        maxScore = max(maxScore, currentSum);
    }

    return maxScore;
}

/**
 * 使用前缀和数组优化版本
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */
int maximumUniqueSubarrayWithPrefixSum(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    // 计算前缀和数组
    vector<int> prefixSum(n + 1, 0);
    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    int maxScore = 0;
    int left = 0;
    unordered_map<int, int> lastSeen;

    for (int right = 0; right < n; right++) {
        int num = nums[right];

        // 如果当前元素已经在窗口中，并且位置在 left 之后
        if (lastSeen.find(num) != lastSeen.end() && lastSeen[num] >= left) {
            left = lastSeen[num] + 1;
        }

        // 更新当前元素的位置
        lastSeen[num] = right;

        // 计算当前窗口的和
        int currentSum = prefixSum[right + 1] - prefixSum[left];
```

```
        maxScore = max(maxScore, currentSum);
    }

    return maxScore;
}

/**
 * 使用数组代替哈希表（当数字范围有限时）
 * 时间复杂度：O(n)，空间复杂度：O(max_value)
 */
int maximumUniqueSubarrayWithArray(vector<int>& nums) {
    int n = nums.size();
    if (n == 0) return 0;

    // 找到数组中的最大值
    int maxVal = 0;
    for (int num : nums) {
        maxVal = max(maxVal, num);
    }

    int maxScore = 0;
    int currentSum = 0;
    int left = 0;
    vector<bool> inWindow(maxVal + 1, false); // 记录元素是否在窗口中

    for (int right = 0; right < n; right++) {
        int num = nums[right];

        // 如果当前元素已经在窗口中，收缩左边界
        while (inWindow[num]) {
            currentSum -= nums[left];
            inWindow[nums[left]] = false;
            left++;
        }

        // 添加当前元素到窗口
        inWindow[num] = true;
        currentSum += num;

        // 更新最大得分
        maxScore = max(maxScore, currentSum);
    }
```

```
        return maxScore;
    }
};

// 测试函数
void testMaximumUniqueSubarray() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {4, 2, 4, 5, 6};
    int result1 = solution.maximumUniqueSubarray(nums1);
    cout << "输入数组: ";
    for (int num : nums1) cout << num << " ";
    cout << endl;
    cout << "最大得分: " << result1 << endl;
    cout << "预期: 17" << endl;
    cout << endl;

    // 测试用例 2
    vector<int> nums2 = {5, 2, 1, 2, 5, 2, 1, 2, 5};
    int result2 = solution.maximumUniqueSubarray(nums2);
    cout << "输入数组: ";
    for (int num : nums2) cout << num << " ";
    cout << endl;
    cout << "最大得分: " << result2 << endl;
    cout << "预期: 8" << endl;
    cout << endl;

    // 测试用例 3：所有元素都不同
    vector<int> nums3 = {1, 2, 3, 4, 5};
    int result3 = solution.maximumUniqueSubarray(nums3);
    cout << "输入数组: ";
    for (int num : nums3) cout << num << " ";
    cout << endl;
    cout << "最大得分: " << result3 << endl;
    cout << "预期: 15" << endl;
    cout << endl;

    // 测试用例 4：所有元素都相同
    vector<int> nums4 = {1, 1, 1, 1, 1};
    int result4 = solution.maximumUniqueSubarray(nums4);
    cout << "输入数组: ";
    for (int num : nums4) cout << num << " ";
```

```
    cout << endl;
    cout << "最大得分: " << result4 << endl;
    cout << "预期: 1" << endl;
    cout << endl;

    // 测试用例 5：边界情况，单个元素
    vector<int> nums5 = {5};
    int result5 = solution.maximumUniqueSubarray(nums5);
    cout << "输入数组: ";
    for (int num : nums5) cout << num << " ";
    cout << endl;
    cout << "最大得分: " << result5 << endl;
    cout << "预期: 5" << endl;
    cout << endl;

    // 测试用例 6：空数组
    vector<int> nums6 = {};
    int result6 = solution.maximumUniqueSubarray(nums6);
    cout << "输入数组: ";
    for (int num : nums6) cout << num << " ";
    cout << endl;
    cout << "最大得分: " << result6 << endl;
    cout << "预期: 0" << endl;
}

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

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

文件: Code21_MaximumErasureValue.java
==========================================
package class049;

import java.util.*;

/**
 * 1695. 删除子数组的最大得分
 * 给你一个正整数数组 nums ，请你从中删除一个含有 若干不同元素 的子数组。删除子数组的 得分 就是子数组各元素之 和 。
 * 返回 只删除一个 子数组可获得的 最大得分 。
```

```
 * 如果数组为空，返回 0 。
 *
 * 解题思路：
 * 使用滑动窗口维护一个不含重复元素的子数组
 * 当遇到重复元素时，收缩左边界直到没有重复元素
 * 在滑动过程中记录最大和
 *
 * 时间复杂度：O(n)，其中 n 是数组长度
 * 空间复杂度：O(k)，k 是不同元素的数量
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/maximum-erasure-value/
 */
public class Code21_MaximumErasureValue {

    /**
     * 计算删除子数组的最大得分
     *
     * @param nums 正整数数组
     * @return 最大得分
     */
    public static int maximumUniqueSubarray(int[] nums) {
        int n = nums.length;
        int maxScore = 0; // 最大得分
        int currentSum = 0; // 当前窗口的和
        int left = 0; // 窗口左边界
        Set<Integer> window = new HashSet<>(); // 记录窗口内的元素

        // 滑动窗口右边界
        for (int right = 0; right < n; right++) {
            // 如果当前元素已经在窗口中，收缩左边界
            while (window.contains(nums[right])) {
                currentSum -= nums[left];
                window.remove(nums[left]);
                left++;
            }

            // 添加当前元素到窗口
            window.add(nums[right]);
            currentSum += nums[right];

            // 更新最大得分
```

```
            maxScore = Math.max(maxScore, currentSum);
        }

        return maxScore;
    }

    /**
     * 优化版本：使用哈希表记录元素最后一次出现的位置
     * 时间复杂度：O(n)，空间复杂度：O(k)
     */
    public static int maximumUniqueSubarrayOptimized(int[] nums) {
        int n = nums.length;
        int maxScore = 0;
        int currentSum = 0;
        int left = 0;
        Map<Integer, Integer> lastSeen = new HashMap<>(); // 记录元素最后一次出现的位置

        for (int right = 0; right < n; right++) {
            int num = nums[right];

            // 如果当前元素已经在窗口中，并且位置在 left 之后
            if (lastSeen.containsKey(num) && lastSeen.get(num) >= left) {
                // 移动左边界到重复元素的下一个位置
                int duplicateIndex = lastSeen.get(num);
                for (int i = left; i <= duplicateIndex; i++) {
                    currentSum -= nums[i];
                }
                left = duplicateIndex + 1;
            }

            // 更新当前元素的位置
            lastSeen.put(num, right);
            currentSum += num;

            // 更新最大得分
            maxScore = Math.max(maxScore, currentSum);
        }

        return maxScore;
    }

    /**
     * 使用前缀和数组优化版本
```

```
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */
public static int maximumUniqueSubarrayWithPrefixSum(int[] nums) {
    int n = nums.length;
    if (n == 0) return 0;

    // 计算前缀和数组
    int[] prefixSum = new int[n + 1];
    for (int i = 0; i < n; i++) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    int maxScore = 0;
    int left = 0;
    Map<Integer, Integer> lastSeen = new HashMap<>();

    for (int right = 0; right < n; right++) {
        int num = nums[right];

        // 如果当前元素已经在窗口中，并且位置在 left 之后
        if (lastSeen.containsKey(num) && lastSeen.get(num) >= left) {
            left = lastSeen.get(num) + 1;
        }

        // 更新当前元素的位置
        lastSeen.put(num, right);

        // 计算当前窗口的和
        int currentSum = prefixSum[right + 1] - prefixSum[left];
        maxScore = Math.max(maxScore, currentSum);
    }

    return maxScore;
}

/**
 * 使用数组代替哈希表（当数字范围有限时）
 * 时间复杂度：O(n)，空间复杂度：O(max_value)
 */
public static int maximumUniqueSubarrayWithArray(int[] nums) {
    int n = nums.length;
    if (n == 0) return 0;
```

```
    // 找到数组中的最大值
    int maxVal = 0;
    for (int num : nums) {
        maxVal = Math.max(maxVal, num);
    }

    int maxScore = 0;
    int currentSum = 0;
    int left = 0;
    boolean[] inWindow = new boolean[maxVal + 1]; // 记录元素是否在窗口中

    for (int right = 0; right < n; right++) {
        int num = nums[right];

        // 如果当前元素已经在窗口中，收缩左边界
        while (inWindow[num]) {
            currentSum -= nums[left];
            inWindow[nums[left]] = false;
            left++;
        }

        // 添加当前元素到窗口
        inWindow[num] = true;
        currentSum += num;

        // 更新最大得分
        maxScore = Math.max(maxScore, currentSum);
    }

    return maxScore;
}

// 测试用例
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {4, 2, 4, 5, 6};
    int result1 = maximumUniqueSubarray(nums1);
    System.out.println("输入数组: " + Arrays.toString(nums1));
    System.out.println("最大得分: " + result1);
    System.out.println("预期: 17");
    System.out.println();

    // 测试用例 2
```

```
        int[] nums2 = {5, 2, 1, 2, 5, 2, 1, 2, 5};
        int result2 = maximumUniqueSubarray(nums2);
        System.out.println("输入数组: " + Arrays.toString(nums2));
        System.out.println("最大得分: " + result2);
        System.out.println("预期: 8");
        System.out.println();

        // 测试用例 3：所有元素都不同
        int[] nums3 = {1, 2, 3, 4, 5};
        int result3 = maximumUniqueSubarray(nums3);
        System.out.println("输入数组: " + Arrays.toString(nums3));
        System.out.println("最大得分: " + result3);
        System.out.println("预期: 15");
        System.out.println();

        // 测试用例 4：所有元素都相同
        int[] nums4 = {1, 1, 1, 1, 1};
        int result4 = maximumUniqueSubarray(nums4);
        System.out.println("输入数组: " + Arrays.toString(nums4));
        System.out.println("最大得分: " + result4);
        System.out.println("预期: 1");
        System.out.println();

        // 测试用例 5：边界情况，单个元素
        int[] nums5 = {5};
        int result5 = maximumUniqueSubarray(nums5);
        System.out.println("输入数组: " + Arrays.toString(nums5));
        System.out.println("最大得分: " + result5);
        System.out.println("预期: 5");
        System.out.println();

        // 测试用例 6：空数组
        int[] nums6 = {};
        int result6 = maximumUniqueSubarray(nums6);
        System.out.println("输入数组: " + Arrays.toString(nums6));
        System.out.println("最大得分: " + result6);
        System.out.println("预期: 0");
    }
}
```

==========

文件: Code21_MaximumErasureValue.py

```
==========================================
from typing import List

class Solution:
    """
    1695. 删除子数组的最大得分
    给你一个正整数数组 nums ，请你从中删除一个含有 若干不同元素 的子数组。删除子数组的 得分 就是子数组各元素之 和 。
    返回 只删除一个 子数组可获得的 最大得分 。
    如果数组为空，返回 0 。

    解题思路：
    使用滑动窗口维护一个不含重复元素的子数组
    当遇到重复元素时，收缩左边界直到没有重复元素
    在滑动过程中记录最大和

    时间复杂度：O(n)，其中 n 是数组长度
    空间复杂度：O(k)，k 是不同元素的数量

    是否最优解：是

    测试链接：https://leetcode.cn/problems/maximum-erasure-value/
    """

    def maximumUniqueSubarray(self, nums: List[int]) -> int:
        """
        计算删除子数组的最大得分

        Args:
            nums: 正整数数组

        Returns:
            最大得分
        """
        n = len(nums)
        max_score = 0  # 最大得分
        current_sum = 0  # 当前窗口的和
        left = 0  # 窗口左边界
        window = set()  # 记录窗口内的元素

        # 滑动窗口右边界
        for right in range(n):
            # 如果当前元素已经在窗口中，收缩左边界
```

```
        while nums[right] in window:
            current_sum -= nums[left]
            window.remove(nums[left])
            left += 1

        # 添加当前元素到窗口
        window.add(nums[right])
        current_sum += nums[right]

        # 更新最大得分
        max_score = max(max_score, current_sum)

    return max_score

def maximumUniqueSubarrayOptimized(self, nums: List[int]) -> int:
    """
    优化版本：使用哈希表记录元素最后一次出现的位置
    时间复杂度：O(n)，空间复杂度：O(k)
    """
    n = len(nums)
    max_score = 0
    current_sum = 0
    left = 0
    last_seen = {}  # 记录元素最后一次出现的位置

    for right in range(n):
        num = nums[right]

        # 如果当前元素已经在窗口中，并且位置在 left 之后
        if num in last_seen and last_seen[num] >= left:
            # 移动左边界到重复元素的下一个位置
            duplicate_index = last_seen[num]
            for i in range(left, duplicate_index + 1):
                current_sum -= nums[i]
            left = duplicate_index + 1

        # 更新当前元素的位置
        last_seen[num] = right
        current_sum += num

        # 更新最大得分
        max_score = max(max_score, current_sum)
```

```
    return max_score

def maximumUniqueSubarrayWithPrefixSum(self, nums: List[int]) -> int:
    """
    使用前缀和数组优化版本
    时间复杂度：O(n)，空间复杂度：O(n)
    """
    n = len(nums)
    if n == 0:
        return 0

    # 计算前缀和数组
    prefix_sum = [0] * (n + 1)
    for i in range(n):
        prefix_sum[i + 1] = prefix_sum[i] + nums[i]

    max_score = 0
    left = 0
    last_seen = {}

    for right in range(n):
        num = nums[right]

        # 如果当前元素已经在窗口中，并且位置在 left 之后
        if num in last_seen and last_seen[num] >= left:
            left = last_seen[num] + 1

        # 更新当前元素的位置
        last_seen[num] = right

        # 计算当前窗口的和
        current_sum = prefix_sum[right + 1] - prefix_sum[left]
        max_score = max(max_score, current_sum)

    return max_score

def maximumUniqueSubarrayWithArray(self, nums: List[int]) -> int:
    """
    使用数组代替哈希表（当数字范围有限时）
    时间复杂度：O(n)，空间复杂度：O(max_value)
    """
    n = len(nums)
    if n == 0:
```

```
            return 0

        # 找到数组中的最大值
        max_val = max(nums) if nums else 0

        max_score = 0
        current_sum = 0
        left = 0
        in_window = [False] * (max_val + 1)  # 记录元素是否在窗口中

        for right in range(n):
            num = nums[right]

            # 如果当前元素已经在窗口中，收缩左边界
            while in_window[num]:
                current_sum -= nums[left]
                in_window[nums[left]] = False
                left += 1

            # 添加当前元素到窗口
            in_window[num] = True
            current_sum += num

            # 更新最大得分
            max_score = max(max_score, current_sum)

        return max_score


def test_maximum_unique_subarray():
    """
    测试函数
    """
    solution = Solution()

    # 测试用例 1
    nums1 = [4, 2, 4, 5, 6]
    result1 = solution.maximumUniqueSubarray(nums1)
    print(f"输入数组: {nums1}")
    print(f"最大得分: {result1}")
    print("预期: 17")
    print()
```

```
    # 测试用例 2
    nums2 = [5, 2, 1, 2, 5, 2, 1, 2, 5]
    result2 = solution.maximumUniqueSubarray(nums2)
    print(f"输入数组: {nums2}")
    print(f"最大得分: {result2}")
    print("预期: 8")
    print()

    # 测试用例 3：所有元素都不同
    nums3 = [1, 2, 3, 4, 5]
    result3 = solution.maximumUniqueSubarray(nums3)
    print(f"输入数组: {nums3}")
    print(f"最大得分: {result3}")
    print("预期: 15")
    print()

    # 测试用例 4：所有元素都相同
    nums4 = [1, 1, 1, 1, 1]
    result4 = solution.maximumUniqueSubarray(nums4)
    print(f"输入数组: {nums4}")
    print(f"最大得分: {result4}")
    print("预期: 1")
    print()

    # 测试用例 5：边界情况，单个元素
    nums5 = [5]
    result5 = solution.maximumUniqueSubarray(nums5)
    print(f"输入数组: {nums5}")
    print(f"最大得分: {result5}")
    print("预期: 5")
    print()

    # 测试用例 6：空数组
    nums6 = []
    result6 = solution.maximumUniqueSubarray(nums6)
    print(f"输入数组: {nums6}")
    print(f"最大得分: {result6}")
    print("预期: 0")


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

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

文件: Code22_MaxConsecutiveOnesIII.cpp
==========================================
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/**
 * 1004. 最大连续1的个数 III
 * 给定一个二进制数组 nums 和一个整数 k，如果可以翻转最多 k 个 0 ，则返回 数组中连续 1 的最大个数 。
 *
 * 解题思路：
 * 使用滑动窗口维护一个最多包含k个0的窗口
 * 当窗口内0的个数超过k时，收缩左边界
 * 在滑动过程中记录最大窗口大小
 *
 * 时间复杂度：O(n)，其中n是数组长度
 * 空间复杂度：O(1)
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/max-consecutive-ones-iii/
 */
class Solution {
public:
    /**
     * 计算最大连续1的个数（最多翻转k个0）
     *
     * @param nums 二进制数组
     * @param k 最多可以翻转的0的个数
     * @return 最大连续1的个数
     */
    int longestOnes(vector<int>& nums, int k) {
        int n = nums.size();
        int maxLength = 0; // 最大长度
        int left = 0; // 窗口左边界
        int zeroCount = 0; // 窗口内0的个数

        // 滑动窗口右边界
```

```
    for (int right = 0; right < n; right++) {
        // 如果当前元素是0，增加0的计数
        if (nums[right] == 0) {
            zeroCount++;
        }

        // 如果窗口内0的个数超过k，收缩左边界
        while (zeroCount > k) {
            if (nums[left] == 0) {
                zeroCount--;
            }
            left++;
        }

        // 更新最大长度
        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * 优化版本：使用更简洁的写法
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
int longestOnesOptimized(vector<int>& nums, int k) {
    int n = nums.size();
    int maxLength = 0;
    int left = 0;
    int zeroCount = 0;

    for (int right = 0; right < n; right++) {
        zeroCount += 1 - nums[right]; // 如果nums[right]是0，则zeroCount加1

        if (zeroCount > k) {
            zeroCount -= 1 - nums[left]; // 如果nums[left]是0，则zeroCount减1
            left++;
        }

        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
```

```
}

/**
 * 另一种思路：使用双指针，不显式维护 zeroCount
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
int longestOnesAlternative(vector<int>& nums, int k) {
    int n = nums.size();
    int maxLength = 0;
    int left = 0;
    int right = 0;
    int zeros = 0;

    while (right < n) {
        // 扩展右边界
        if (nums[right] == 0) {
            zeros++;
        }
        right++;

        // 如果 0 的个数超过 k，收缩左边界
        while (zeros > k) {
            if (nums[left] == 0) {
                zeros--;
            }
            left++;
        }

        // 更新最大长度
        maxLength = max(maxLength, right - left);
    }

    return maxLength;
}

/**
 * 使用前缀和思想（当 k 较大时效率更高）
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
int longestOnesWithPrefixSum(vector<int>& nums, int k) {
    int n = nums.size();
    int maxLength = 0;
    int left = 0;
```

```
        int zeroCount = 0;

        for (int right = 0; right < n; right++) {
            if (nums[right] == 0) {
                zeroCount++;
            }

            // 如果0的个数超过k，移动左边界
            if (zeroCount > k) {
                if (nums[left] == 0) {
                    zeroCount--;
                }
                left++;
            }

            // 更新最大长度
            maxLength = max(maxLength, right - left + 1);
        }

        return maxLength;
    }
};

// 测试函数
void testLongestOnes() {
    Solution solution;

    // 测试用例1
    vector<int> nums1 = {1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0};
    int k1 = 2;
    int result1 = solution.longestOnes(nums1, k1);
    cout << "输入数组: ";
    for (int num : nums1) cout << num << " ";
    cout << endl;
    cout << "k = " << k1 << endl;
    cout << "最大连续1的个数: " << result1 << endl;
    cout << "预期: 6" << endl;
    cout << endl;

    // 测试用例2
    vector<int> nums2 = {0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1};
    int k2 = 3;
    int result2 = solution.longestOnes(nums2, k2);
```

```
cout << "输入数组: ";
for (int num : nums2) cout << num << " ";
cout << endl;
cout << "k = " << k2 << endl;
cout << "最大连续 1 的个数: " << result2 << endl;
cout << "预期: 10" << endl;
cout << endl;

// 测试用例 3：k=0
vector<int> nums3 = {1, 1, 0, 1, 1, 1};
int k3 = 0;
int result3 = solution.longestOnes(nums3, k3);
cout << "输入数组: ";
for (int num : nums3) cout << num << " ";
cout << endl;
cout << "k = " << k3 << endl;
cout << "最大连续 1 的个数: " << result3 << endl;
cout << "预期: 3" << endl;
cout << endl;

// 测试用例 4：k 大于 0 的个数
vector<int> nums4 = {0, 0, 0, 0};
int k4 = 2;
int result4 = solution.longestOnes(nums4, k4);
cout << "输入数组: ";
for (int num : nums4) cout << num << " ";
cout << endl;
cout << "k = " << k4 << endl;
cout << "最大连续 1 的个数: " << result4 << endl;
cout << "预期: 2" << endl;
cout << endl;

// 测试用例 5：全是 1
vector<int> nums5 = {1, 1, 1, 1, 1};
int k5 = 2;
int result5 = solution.longestOnes(nums5, k5);
cout << "输入数组: ";
for (int num : nums5) cout << num << " ";
cout << endl;
cout << "k = " << k5 << endl;
cout << "最大连续 1 的个数: " << result5 << endl;
cout << "预期: 5" << endl;
cout << endl;
```

```
    // 测试用例6：边界情况，单个元素
    vector<int> nums6 = {0};
    int k6 = 1;
    int result6 = solution.longestOnes(nums6, k6);
    cout << "输入数组: ";
    for (int num : nums6) cout << num << " ";
    cout << endl;
    cout << "k = " << k6 << endl;
    cout << "最大连续1的个数: " << result6 << endl;
    cout << "预期: 1" << endl;
}

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

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

文件: Code22_MaxConsecutiveOnesIII.java
==========================================
package class049;

/**
 * 1004. 最大连续1的个数 III
 * 给定一个二进制数组 nums 和一个整数 k，如果可以翻转最多 k 个 0 ，则返回 数组中连续 1 的最大个数 。
 *
 * 解题思路：
 * 使用滑动窗口维护一个最多包含k个0的窗口
 * 当窗口内0的个数超过k时，收缩左边界
 * 在滑动过程中记录最大窗口大小
 *
 * 时间复杂度：O(n)，其中n是数组长度
 * 空间复杂度：O(1)
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/max-consecutive-ones-iii/
 */
public class Code22_MaxConsecutiveOnesIII {
```

```
/**
 * 计算最大连续 1 的个数（最多翻转 k 个 0）
 *
 * @param nums 二进制数组
 * @param k 最多可以翻转的 0 的个数
 * @return 最大连续 1 的个数
 */
public static int longestOnes(int[] nums, int k) {
    int n = nums.length;
    int maxLength = 0; // 最大长度
    int left = 0; // 窗口左边界
    int zeroCount = 0; // 窗口内 0 的个数

    // 滑动窗口右边界
    for (int right = 0; right < n; right++) {
        // 如果当前元素是 0，增加 0 的计数
        if (nums[right] == 0) {
            zeroCount++;
        }

        // 如果窗口内 0 的个数超过 k，收缩左边界
        while (zeroCount > k) {
            if (nums[left] == 0) {
                zeroCount--;
            }
            left++;
        }

        // 更新最大长度
        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * 优化版本：使用更简洁的写法
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
public static int longestOnesOptimized(int[] nums, int k) {
    int n = nums.length;
    int maxLength = 0;
    int left = 0;
```

```
    int zeroCount = 0;

    for (int right = 0; right < n; right++) {
        zeroCount += 1 - nums[right]; // 如果 nums[right]是 0，则 zeroCount 加 1

        if (zeroCount > k) {
            zeroCount -= 1 - nums[left]; // 如果 nums[left]是 0，则 zeroCount 减 1
            left++;
        }

        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * 另一种思路：使用双指针，不显式维护 zeroCount
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
public static int longestOnesAlternative(int[] nums, int k) {
    int n = nums.length;
    int maxLength = 0;
    int left = 0;
    int right = 0;
    int zeros = 0;

    while (right < n) {
        // 扩展右边界
        if (nums[right] == 0) {
            zeros++;
        }
        right++;

        // 如果 0 的个数超过 k，收缩左边界
        while (zeros > k) {
            if (nums[left] == 0) {
                zeros--;
            }
            left++;
        }

        // 更新最大长度
```

```
            maxLength = Math.max(maxLength, right - left);
        }

        return maxLength;
    }

    /**
     * 使用前缀和思想（当 k 较大时效率更高）
     * 时间复杂度：O(n)，空间复杂度：O(1)
     */
    public static int longestOnesWithPrefixSum(int[] nums, int k) {
        int n = nums.length;
        int maxLength = 0;
        int left = 0;
        int zeroCount = 0;

        for (int right = 0; right < n; right++) {
            if (nums[right] == 0) {
                zeroCount++;
            }

            // 如果 0 的个数超过 k，移动左边界
            if (zeroCount > k) {
                if (nums[left] == 0) {
                    zeroCount--;
                }
                left++;
            }

            // 更新最大长度
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }

    // 测试用例
    public static void main(String[] args) {
        // 测试用例 1
        int[] nums1 = {1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0};
        int k1 = 2;
        int result1 = longestOnes(nums1, k1);
        System.out.println("输入数组: " + java.util.Arrays.toString(nums1));
```

```
System.out.println("k = " + k1);
System.out.println("最大连续 1 的个数: " + result1);
System.out.println("预期: 6");
System.out.println();

// 测试用例 2
int[] nums2 = {0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1};
int k2 = 3;
int result2 = longestOnes(nums2, k2);
System.out.println("输入数组: " + java.util.Arrays.toString(nums2));
System.out.println("k = " + k2);
System.out.println("最大连续 1 的个数: " + result2);
System.out.println("预期: 10");
System.out.println();

// 测试用例 3：k=0
int[] nums3 = {1, 1, 0, 1, 1, 1};
int k3 = 0;
int result3 = longestOnes(nums3, k3);
System.out.println("输入数组: " + java.util.Arrays.toString(nums3));
System.out.println("k = " + k3);
System.out.println("最大连续 1 的个数: " + result3);
System.out.println("预期: 3");
System.out.println();

// 测试用例 4：k 大于 0 的个数
int[] nums4 = {0, 0, 0, 0};
int k4 = 2;
int result4 = longestOnes(nums4, k4);
System.out.println("输入数组: " + java.util.Arrays.toString(nums4));
System.out.println("k = " + k4);
System.out.println("最大连续 1 的个数: " + result4);
System.out.println("预期: 2");
System.out.println();

// 测试用例 5：全是 1
int[] nums5 = {1, 1, 1, 1, 1};
int k5 = 2;
int result5 = longestOnes(nums5, k5);
System.out.println("输入数组: " + java.util.Arrays.toString(nums5));
System.out.println("k = " + k5);
System.out.println("最大连续 1 的个数: " + result5);
System.out.println("预期: 5");
```

```
        System.out.println();

        // 测试用例 6：边界情况，单个元素
        int[] nums6 = {0};
        int k6 = 1;
        int result6 = longestOnes(nums6, k6);
        System.out.println("输入数组: " + java.util.Arrays.toString(nums6));
        System.out.println("k = " + k6);
        System.out.println("最大连续 1 的个数: " + result6);
        System.out.println("预期: 1");
    }
}
```

==========

文件: Code22_MaxConsecutiveOnesIII.py

==========

```
from typing import List

class Solution:
    """
    1004. 最大连续 1 的个数 III
    给定一个二进制数组 nums 和一个整数 k，如果可以翻转最多 k 个 0 ，则返回 数组中连续 1 的最大个数 。

    解题思路：
    使用滑动窗口维护一个最多包含 k 个 0 的窗口
    当窗口内 0 的个数超过 k 时，收缩左边界
    在滑动过程中记录最大窗口大小

    时间复杂度：O(n)，其中 n 是数组长度
    空间复杂度：O(1)

    是否最优解：是

    测试链接：https://leetcode.cn/problems/max-consecutive-ones-iii/
    """

    def longestOnes(self, nums: List[int], k: int) -> int:
        """
        计算最大连续 1 的个数（最多翻转 k 个 0）

        Args:
```

```
        nums: 二进制数组
        k: 最多可以翻转的 0 的个数

    Returns:
        最大连续 1 的个数
    """
    n = len(nums)
    max_length = 0  # 最大长度
    left = 0        # 窗口左边界
    zero_count = 0  # 窗口内 0 的个数

    # 滑动窗口右边界
    for right in range(n):
        # 如果当前元素是 0，增加 0 的计数
        if nums[right] == 0:
            zero_count += 1

        # 如果窗口内 0 的个数超过 k，收缩左边界
        while zero_count > k:
            if nums[left] == 0:
                zero_count -= 1
            left += 1

        # 更新最大长度
        max_length = max(max_length, right - left + 1)

    return max_length

def longestOnesOptimized(self, nums: List[int], k: int) -> int:
    """
    优化版本：使用更简洁的写法
    时间复杂度：O(n)，空间复杂度：O(1)
    """
    n = len(nums)
    max_length = 0
    left = 0
    zero_count = 0

    for right in range(n):
        zero_count += 1 - nums[right]  # 如果 nums[right]是 0，则 zero_count 加 1

        if zero_count > k:
            zero_count -= 1 - nums[left]  # 如果 nums[left]是 0，则 zero_count 减 1
```

```
            left += 1

        max_length = max(max_length, right - left + 1)

    return max_length

def longestOnesAlternative(self, nums: List[int], k: int) -> int:
    """
    另一种思路：使用双指针，不显式维护 zero_count
    时间复杂度：O(n)，空间复杂度：O(1)
    """
    n = len(nums)
    max_length = 0
    left = 0
    right = 0
    zeros = 0

    while right < n:
        # 扩展右边界
        if nums[right] == 0:
            zeros += 1
        right += 1

        # 如果 0 的个数超过 k，收缩左边界
        while zeros > k:
            if nums[left] == 0:
                zeros -= 1
            left += 1

        # 更新最大长度
        max_length = max(max_length, right - left)

    return max_length

def longestOnesWithPrefixSum(self, nums: List[int], k: int) -> int:
    """
    使用前缀和思想（当 k 较大时效率更高）
    时间复杂度：O(n)，空间复杂度：O(1)
    """
    n = len(nums)
    max_length = 0
    left = 0
    zero_count = 0
```

```
        for right in range(n):
            if nums[right] == 0:
                zero_count += 1

            # 如果0的个数超过k，移动左边界
            if zero_count > k:
                if nums[left] == 0:
                    zero_count -= 1
                left += 1

            # 更新最大长度
            max_length = max(max_length, right - left + 1)

        return max_length


def test_longest_ones():
    """
    测试函数
    """
    solution = Solution()

    # 测试用例1
    nums1 = [1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0]
    k1 = 2
    result1 = solution.longestOnes(nums1, k1)
    print(f"输入数组: {nums1}")
    print(f"k = {k1}")
    print(f"最大连续1的个数: {result1}")
    print("预期: 6")
    print()

    # 测试用例2
    nums2 = [0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1]
    k2 = 3
    result2 = solution.longestOnes(nums2, k2)
    print(f"输入数组: {nums2}")
    print(f"k = {k2}")
    print(f"最大连续1的个数: {result2}")
    print("预期: 10")
    print()
```

```
    # 测试用例 3：k=0
    nums3 = [1, 1, 0, 1, 1, 1]
    k3 = 0
    result3 = solution.longestOnes(nums3, k3)
    print(f"输入数组: {nums3}")
    print(f"k = {k3}")
    print(f"最大连续 1 的个数: {result3}")
    print("预期: 3")
    print()

    # 测试用例 4：k 大于 0 的个数
    nums4 = [0, 0, 0, 0]
    k4 = 2
    result4 = solution.longestOnes(nums4, k4)
    print(f"输入数组: {nums4}")
    print(f"k = {k4}")
    print(f"最大连续 1 的个数: {result4}")
    print("预期: 2")
    print()

    # 测试用例 5：全是 1
    nums5 = [1, 1, 1, 1, 1]
    k5 = 2
    result5 = solution.longestOnes(nums5, k5)
    print(f"输入数组: {nums5}")
    print(f"k = {k5}")
    print(f"最大连续 1 的个数: {result5}")
    print("预期: 5")
    print()

    # 测试用例 6：边界情况，单个元素
    nums6 = [0]
    k6 = 1
    result6 = solution.longestOnes(nums6, k6)
    print(f"输入数组: {nums6}")
    print(f"k = {k6}")
    print(f"最大连续 1 的个数: {result6}")
    print("预期: 1")


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

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

文件: Code23_GetEqualSubstringsWithinBudget.cpp
==========================================
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

/**
 * 1208. 尽可能使字符串相等
 * 给你两个长度相同的字符串，s 和 t。
 * 将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。
 * 用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，这也意味着字符串的转化可能是不完全的。
 * 如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。
 * 如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。
 *
 * 解题思路：
 * 使用滑动窗口维护一个子数组，使得子数组内字符转换的开销总和不超过 maxCost
 * 当开销超过 maxCost 时，收缩左边界
 * 在滑动过程中记录最大窗口大小
 *
 * 时间复杂度：O(n)，其中 n 是字符串长度
 * 空间复杂度：O(1)
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/get-equal-substrings-within-budget/
 */
class Solution {
public:
    /**
     * 计算可以转化的最大子字符串长度
     *
     * @param s 源字符串
     * @param t 目标字符串
     * @param maxCost 最大预算
     * @return 可以转化的最大长度
     */
```

```
int equalSubstring(string s, string t, int maxCost) {
    int n = s.length();
    int maxLength = 0; // 最大长度
    int currentCost = 0; // 当前窗口的开销
    int left = 0; // 窗口左边界

    // 滑动窗口右边界
    for (int right = 0; right < n; right++) {
        // 计算当前字符的转换开销
        int cost = abs(s[right] - t[right]);
        currentCost += cost;

        // 如果当前开销超过最大预算，收缩左边界
        while (currentCost > maxCost) {
            int leftCost = abs(s[left] - t[left]);
            currentCost -= leftCost;
            left++;
        }

        // 更新最大长度
        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * 优化版本：使用数组预先计算开销
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */
int equalSubstringOptimized(string s, string t, int maxCost) {
    int n = s.length();
    if (n == 0) return 0;

    // 预先计算每个位置的转换开销
    vector<int> costs(n);
    for (int i = 0; i < n; i++) {
        costs[i] = abs(s[i] - t[i]);
    }

    int maxLength = 0;
    int currentCost = 0;
    int left = 0;
```

```
    for (int right = 0; right < n; right++) {
        currentCost += costs[right];

        // 如果当前开销超过最大预算，收缩左边界
        while (currentCost > maxCost) {
            currentCost -= costs[left];
            left++;
        }

        maxLength = max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * 另一种思路：使用双指针，不显式维护 currentCost
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
int equalSubstringAlternative(string s, string t, int maxCost) {
    int n = s.length();
    int maxLength = 0;
    int left = 0;
    int right = 0;
    int currentCost = 0;

    while (right < n) {
        // 扩展右边界
        int cost = abs(s[right] - t[right]);
        currentCost += cost;
        right++;

        // 如果开销超过最大预算，收缩左边界
        while (currentCost > maxCost) {
            int leftCost = abs(s[left] - t[left]);
            currentCost -= leftCost;
            left++;
        }

        // 更新最大长度
        maxLength = max(maxLength, right - left);
    }
```

```
        return maxLength;
    }

    /**
     * 使用前缀和思想（当 maxCost 较大时效率更高）
     * 时间复杂度：O(n)，空间复杂度：O(n)
     */
    int equalSubstringWithPrefixSum(string s, string t, int maxCost) {
        int n = s.length();
        if (n == 0) return 0;

        // 计算前缀和数组
        vector<int> prefixSum(n + 1, 0);
        for (int i = 0; i < n; i++) {
            int cost = abs(s[i] - t[i]);
            prefixSum[i + 1] = prefixSum[i] + cost;
        }

        int maxLength = 0;
        int left = 0;

        for (int right = 0; right < n; right++) {
            // 计算从 left 到 right 的开销
            int currentCost = prefixSum[right + 1] - prefixSum[left];

            // 如果开销不超过最大预算，更新最大长度
            if (currentCost <= maxCost) {
                maxLength = max(maxLength, right - left + 1);
            } else {
                // 开销超过预算，移动左边界
                left++;
            }
        }

        return maxLength;
    }
};

// 测试函数
void testEqualSubstring() {
    Solution solution;
```

```
// 测试用例 1
string s1 = "abcd";
string t1 = "bcdf";
int maxCost1 = 3;
int result1 = solution.equalSubstring(s1, t1, maxCost1);
cout << "s = \"" << s1 << "\", t = \"" << t1 << "\", maxCost = " << maxCost1 << endl;
cout << "最大长度: " << result1 << endl;
cout << "预期: 3" << endl;
cout << endl;

// 测试用例 2
string s2 = "abcd";
string t2 = "cdef";
int maxCost2 = 3;
int result2 = solution.equalSubstring(s2, t2, maxCost2);
cout << "s = \"" << s2 << "\", t = \"" << t2 << "\", maxCost = " << maxCost2 << endl;
cout << "最大长度: " << result2 << endl;
cout << "预期: 1" << endl;
cout << endl;

// 测试用例 3
string s3 = "abcd";
string t3 = "acde";
int maxCost3 = 0;
int result3 = solution.equalSubstring(s3, t3, maxCost3);
cout << "s = \"" << s3 << "\", t = \"" << t3 << "\", maxCost = " << maxCost3 << endl;
cout << "最大长度: " << result3 << endl;
cout << "预期: 1" << endl;
cout << endl;

// 测试用例 4：相同字符串
string s4 = "abcd";
string t4 = "abcd";
int maxCost4 = 10;
int result4 = solution.equalSubstring(s4, t4, maxCost4);
cout << "s = \"" << s4 << "\", t = \"" << t4 << "\", maxCost = " << maxCost4 << endl;
cout << "最大长度: " << result4 << endl;
cout << "预期: 4" << endl;
cout << endl;

// 测试用例 5：空字符串
string s5 = "";
string t5 = "";
```

```
    int maxCost5 = 10;
    int result5 = solution.equalSubstring(s5, t5, maxCost5);
    cout << "s = \"" << s5 << "\", t = \"" << t5 << "\", maxCost = " << maxCost5 << endl;
    cout << "最大长度: " << result5 << endl;
    cout << "预期: 0" << endl;
    cout << endl;

    // 测试用例 6：边界情况，单个字符
    string s6 = "a";
    string t6 = "b";
    int maxCost6 = 1;
    int result6 = solution.equalSubstring(s6, t6, maxCost6);
    cout << "s = \"" << s6 << "\", t = \"" << t6 << "\", maxCost = " << maxCost6 << endl;
    cout << "最大长度: " << result6 << endl;
    cout << "预期: 1" << endl;
}

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

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

文件: Code23_GetEqualSubstringsWithinBudget.java

```
==========================================
package class049;

/**
 * 1208. 尽可能使字符串相等
 * 给你两个长度相同的字符串，s 和 t。
 * 将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。
 * 用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，这也意味着字符串的转化可能是不完全的。
 * 如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。
 * 如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。
 *
 * 解题思路：
 * 使用滑动窗口维护一个子数组，使得子数组内字符转换的开销总和不超过 maxCost
 * 当开销超过 maxCost 时，收缩左边界
 * 在滑动过程中记录最大窗口大小
 *
```

```
 * 时间复杂度：O(n)，其中 n 是字符串长度
 * 空间复杂度：O(1)
 *
 * 是否最优解：是
 *
 * 测试链接：https://leetcode.cn/problems/get-equal-substrings-within-budget/
 */
public class Code23_GetEqualSubstringsWithinBudget {

    /**
     * 计算可以转化的最大子字符串长度
     *
     * @param s 源字符串
     * @param t 目标字符串
     * @param maxCost 最大预算
     * @return 可以转化的最大长度
     */
    public static int equalSubstring(String s, String t, int maxCost) {
        int n = s.length();
        int maxLength = 0; // 最大长度
        int currentCost = 0; // 当前窗口的开销
        int left = 0; // 窗口左边界

        // 滑动窗口右边界
        for (int right = 0; right < n; right++) {
            // 计算当前字符的转换开销
            int cost = Math.abs(s.charAt(right) - t.charAt(right));
            currentCost += cost;

            // 如果当前开销超过最大预算，收缩左边界
            while (currentCost > maxCost) {
                int leftCost = Math.abs(s.charAt(left) - t.charAt(left));
                currentCost -= leftCost;
                left++;
            }

            // 更新最大长度
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }
```

```java
/**
 * 优化版本：使用数组预先计算开销
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */
public static int equalSubstringOptimized(String s, String t, int maxCost) {
    int n = s.length();
    if (n == 0) return 0;

    // 预先计算每个位置的转换开销
    int[] costs = new int[n];
    for (int i = 0; i < n; i++) {
        costs[i] = Math.abs(s.charAt(i) - t.charAt(i));
    }

    int maxLength = 0;
    int currentCost = 0;
    int left = 0;

    for (int right = 0; right < n; right++) {
        currentCost += costs[right];

        // 如果当前开销超过最大预算，收缩左边界
        while (currentCost > maxCost) {
            currentCost -= costs[left];
            left++;
        }

        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}

/**
 * 另一种思路：使用双指针，不显式维护 currentCost
 * 时间复杂度：O(n)，空间复杂度：O(1)
 */
public static int equalSubstringAlternative(String s, String t, int maxCost) {
    int n = s.length();
    int maxLength = 0;
    int left = 0;
    int right = 0;
    int currentCost = 0;
```

```
    while (right < n) {
        // 扩展右边界
        int cost = Math.abs(s.charAt(right) - t.charAt(right));
        currentCost += cost;
        right++;

        // 如果开销超过最大预算，收缩左边界
        while (currentCost > maxCost) {
            int leftCost = Math.abs(s.charAt(left) - t.charAt(left));
            currentCost -= leftCost;
            left++;
        }

        // 更新最大长度
        maxLength = Math.max(maxLength, right - left);
    }

    return maxLength;
}

/**
 * 使用前缀和思想（当 maxCost 较大时效率更高）
 * 时间复杂度：O(n)，空间复杂度：O(n)
 */
public static int equalSubstringWithPrefixSum(String s, String t, int maxCost) {
    int n = s.length();
    if (n == 0) return 0;

    // 计算前缀和数组
    int[] prefixSum = new int[n + 1];
    for (int i = 0; i < n; i++) {
        int cost = Math.abs(s.charAt(i) - t.charAt(i));
        prefixSum[i + 1] = prefixSum[i] + cost;
    }

    int maxLength = 0;
    int left = 0;

    for (int right = 0; right < n; right++) {
        // 计算从 left 到 right 的开销
        int currentCost = prefixSum[right + 1] - prefixSum[left];
```

```
            // 如果开销不超过最大预算，更新最大长度
            if (currentCost <= maxCost) {
                maxLength = Math.max(maxLength, right - left + 1);
            } else {
                // 开销超过预算，移动左边界
                left++;
            }
        }

        return maxLength;
    }

    // 测试用例
    public static void main(String[] args) {
        // 测试用例 1
        String s1 = "abcd";
        String t1 = "bcdf";
        int maxCost1 = 3;
        int result1 = equalSubstring(s1, t1, maxCost1);
        System.out.println("s = \"" + s1 + "\", t = \"" + t1 + "\", maxCost = " + maxCost1);
        System.out.println("最大长度: " + result1);
        System.out.println("预期: 3");
        System.out.println();

        // 测试用例 2
        String s2 = "abcd";
        String t2 = "cdef";
        int maxCost2 = 3;
        int result2 = equalSubstring(s2, t2, maxCost2);
        System.out.println("s = \"" + s2 + "\", t = \"" + t2 + "\", maxCost = " + maxCost2);
        System.out.println("最大长度: " + result2);
        System.out.println("预期: 1");
        System.out.println();

        // 测试用例 3
        String s3 = "abcd";
        String t3 = "acde";
        int maxCost3 = 0;
        int result3 = equalSubstring(s3, t3, maxCost3);
        System.out.println("s = \"" + s3 + "\", t = \"" + t3 + "\", maxCost = " + maxCost3);
        System.out.println("最大长度: " + result3);
        System.out.println("预期: 1");
        System.out.println();
```

```
        // 测试用例 4：相同字符串
        String s4 = "abcd";
        String t4 = "abcd";
        int maxCost4 = 10;
        int result4 = equalSubstring(s4, t4, maxCost4);
        System.out.println("s = \"" + s4 + "\", t = \"" + t4 + "\", maxCost = " + maxCost4);
        System.out.println("最大长度: " + result4);
        System.out.println("预期: 4");
        System.out.println();

        // 测试用例 5：空字符串
        String s5 = "";
        String t5 = "";
        int maxCost5 = 10;
        int result5 = equalSubstring(s5, t5, maxCost5);
        System.out.println("s = \"" + s5 + "\", t = \"" + t5 + "\", maxCost = " + maxCost5);
        System.out.println("最大长度: " + result5);
        System.out.println("预期: 0");
        System.out.println();

        // 测试用例 6：边界情况，单个字符
        String s6 = "a";
        String t6 = "b";
        int maxCost6 = 1;
        int result6 = equalSubstring(s6, t6, maxCost6);
        System.out.println("s = \"" + s6 + "\", t = \"" + t6 + "\", maxCost = " + maxCost6);
        System.out.println("最大长度: " + result6);
        System.out.println("预期: 1");
    }
}
```

==========

文件: Code23_GetEqualSubstringsWithinBudget.py

==========

```
class Solution:
    """
    1208. 尽可能使字符串相等
    给你两个长度相同的字符串，s 和 t。
    将 s 中的第 i 个字符变到 t 中的第 i 个字符需要 |s[i] - t[i]| 的开销（开销可能为 0），也就是两个字符的 ASCII 码值的差的绝对值。
    用于变更字符串的最大预算是 maxCost。在转化字符串时，总开销应当小于等于该预算，这也意味着字符
```

串的转化可能是不完全的。
如果你可以将 s 的子字符串转化为它在 t 中对应的子字符串，则返回可以转化的最大长度。
如果 s 中没有子字符串可以转化成 t 中对应的子字符串，则返回 0。

解题思路：
使用滑动窗口维护一个子数组，使得子数组内字符转换的开销总和不超过 maxCost
当开销超过 maxCost 时，收缩左边界
在滑动过程中记录最大窗口大小

时间复杂度：O(n)，其中 n 是字符串长度
空间复杂度：O(1)

是否最优解：是

测试链接：https://leetcode.cn/problems/get-equal-substrings-within-budget/
"""

```
def equalSubstring(self, s: str, t: str, maxCost: int) -> int:
    """
    计算可以转化的最大子字符串长度

    Args:
        s: 源字符串
        t: 目标字符串
        maxCost: 最大预算

    Returns:
        可以转化的最大长度
    """
    n = len(s)
    max_length = 0  # 最大长度
    current_cost = 0  # 当前窗口的开销
    left = 0  # 窗口左边界

    # 滑动窗口右边界
    for right in range(n):
        # 计算当前字符的转换开销
        cost = abs(ord(s[right]) - ord(t[right]))
        current_cost += cost

        # 如果当前开销超过最大预算，收缩左边界
        while current_cost > maxCost:
            left_cost = abs(ord(s[left]) - ord(t[left]))
```

```
            current_cost -= left_cost
            left += 1

        # 更新最大长度
        max_length = max(max_length, right - left + 1)

    return max_length

def equalSubstringOptimized(self, s: str, t: str, maxCost: int) -> int:
    """
    优化版本：使用数组预先计算开销
    时间复杂度：O(n)，空间复杂度：O(n)
    """
    n = len(s)
    if n == 0:
        return 0

    # 预先计算每个位置的转换开销
    costs = [0] * n
    for i in range(n):
        costs[i] = abs(ord(s[i]) - ord(t[i]))

    max_length = 0
    current_cost = 0
    left = 0

    for right in range(n):
        current_cost += costs[right]

        # 如果当前开销超过最大预算，收缩左边界
        while current_cost > maxCost:
            current_cost -= costs[left]
            left += 1

        max_length = max(max_length, right - left + 1)

    return max_length

def equalSubstringAlternative(self, s: str, t: str, maxCost: int) -> int:
    """
    另一种思路：使用双指针，不显式维护 current_cost
    时间复杂度：O(n)，空间复杂度：O(1)
    """
```

```
    n = len(s)
    max_length = 0
    left = 0
    right = 0
    current_cost = 0

    while right < n:
        # 扩展右边界
        cost = abs(ord(s[right]) - ord(t[right]))
        current_cost += cost
        right += 1

        # 如果开销超过最大预算，收缩左边界
        while current_cost > maxCost:
            left_cost = abs(ord(s[left]) - ord(t[left]))
            current_cost -= left_cost
            left += 1

        # 更新最大长度
        max_length = max(max_length, right - left)

    return max_length

def equalSubstringWithPrefixSum(self, s: str, t: str, maxCost: int) -> int:
    """
    使用前缀和思想（当 maxCost 较大时效率更高）
    时间复杂度：O(n)，空间复杂度：O(n)
    """
    n = len(s)
    if n == 0:
        return 0

    # 计算前缀和数组
    prefix_sum = [0] * (n + 1)
    for i in range(n):
        cost = abs(ord(s[i]) - ord(t[i]))
        prefix_sum[i + 1] = prefix_sum[i] + cost

    max_length = 0
    left = 0

    for right in range(n):
        # 计算从 left 到 right 的开销
```

```
            current_cost = prefix_sum[right + 1] - prefix_sum[left]

            # 如果开销不超过最大预算，更新最大长度
            if current_cost <= maxCost:
                max_length = max(max_length, right - left + 1)
            else:
                # 开销超过预算，移动左边界
                left += 1

        return max_length


def test_equal_substring():
    """
    测试函数
    """
    solution = Solution()

    # 测试用例 1
    s1 = "abcd"
    t1 = "bcdf"
    maxCost1 = 3
    result1 = solution.equalSubstring(s1, t1, maxCost1)
    print(f"s = \"{s1}\", t = \"{t1}\", maxCost = {maxCost1}")
    print(f"最大长度: {result1}")
    print("预期: 3")
    print()

    # 测试用例 2
    s2 = "abcd"
    t2 = "cdef"
    maxCost2 = 3
    result2 = solution.equalSubstring(s2, t2, maxCost2)
    print(f"s = \"{s2}\", t = \"{t2}\", maxCost = {maxCost2}")
    print(f"最大长度: {result2}")
    print("预期: 1")
    print()

    # 测试用例 3
    s3 = "abcd"
    t3 = "acde"
    maxCost3 = 0
    result3 = solution.equalSubstring(s3, t3, maxCost3)
```

```
    print(f"s = \"{s3}\", t = \"{t3}\", maxCost = {maxCost3}")
    print(f"最大长度: {result3}")
    print("预期: 1")
    print()

    # 测试用例 4：相同字符串
    s4 = "abcd"
    t4 = "abcd"
    maxCost4 = 10
    result4 = solution.equalSubstring(s4, t4, maxCost4)
    print(f"s = \"{s4}\", t = \"{t4}\", maxCost = {maxCost4}")
    print(f"最大长度: {result4}")
    print("预期: 4")
    print()

    # 测试用例 5：空字符串
    s5 = ""
    t5 = ""
    maxCost5 = 10
    result5 = solution.equalSubstring(s5, t5, maxCost5)
    print(f"s = \"{s5}\", t = \"{t5}\", maxCost = {maxCost5}")
    print(f"最大长度: {result5}")
    print("预期: 0")
    print()

    # 测试用例 6：边界情况，单个字符
    s6 = "a"
    t6 = "b"
    maxCost6 = 1
    result6 = solution.equalSubstring(s6, t6, maxCost6)
    print(f"s = \"{s6}\", t = \"{t6}\", maxCost = {maxCost6}")
    print(f"最大长度: {result6}")
    print("预期: 1")


if __name__ == "__main__":
    test_equal_substring()

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