

=====

文件夹: class076_TwoPointers

=====

[Markdown 文件]

=====

文件: README.md

=====

Class050: 双指针技巧专题 - 全面完善版

📃 概述

双指针技巧是算法中常用的优化技术，通过使用两个指针协同工作来减少时间或空间复杂度。本专题系统整理了双指针在各种场景下的应用，涵盖从基础到高级的各种题目。

🔎 双指针分类与核心思想

1. 左右指针（对撞指针）

- **核心思想**: 两个指针从数组两端向中间移动
- **适用场景**: 有序数组、两数之和、盛水容器等
- **时间复杂度**: 通常为 $O(n)$
- **经典题目**: 两数之和 II、盛最多水的容器、接雨水

2. 快慢指针

- **核心思想**: 两个指针从同一端出发，速度不同
- **适用场景**: 链表环检测、删除重复元素、寻找中点等
- **时间复杂度**: 通常为 $O(n)$
- **经典题目**: 删除有序数组中的重复项、寻找重复数、链表环检测

3. 滑动窗口

- **核心思想**: 两个指针维护一个窗口，根据条件动态调整窗口大小
- **适用场景**: 子数组/子串问题、连续序列等
- **时间复杂度**: 通常为 $O(n)$
- **经典题目**: 最小覆盖子串、最长无重复子串、长度最小的子数组

4. 多指针（三指针及以上）

- **核心思想**: 使用多个指针协同工作
- **适用场景**: 荷兰国旗问题、多数之和问题
- **时间复杂度**: 通常为 $O(n)$ 或 $O(n^2)$
- **经典题目**: 颜色分类、三数之和、四数之和

📄 完整题目列表（按难度和类型分类）

🔥 基础题目（必须掌握）

1. 两数之和系列

- **Two Sum II - Input Array Is Sorted** (LeetCode 167)
 - 链接: <https://leetcode.cn/problems/two-sum-ii-input-array-is-sorted/>
 - 解法: 左右指针
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code08_TwoSumII.*, Code11_TwoSumII.*

2. 数组操作系列

- **Remove Duplicates from Sorted Array** (LeetCode 26)
 - 链接: <https://leetcode.cn/problems/remove-duplicates-from-sorted-array/>
 - 解法: 快慢指针
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code22_RemoveDuplicatesFromSortedArray.*
- **Remove Element** (LeetCode 27)
 - 链接: <https://leetcode.cn/problems/remove-element/>
 - 解法: 快慢指针
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code23_RemoveElement.*
- **Move Zeroes** (LeetCode 283)
 - 链接: <https://leetcode.cn/problems/move-zeroes/>
 - 解法: 快慢指针
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code26_MoveZeroes.*

🚀 进阶题目 (面试高频)

3. 多数之和系列

- **3Sum** (LeetCode 15)
 - 链接: <https://leetcode.cn/problems/3sum/>
 - 解法: 排序+双指针
 - 时间复杂度: $O(n^2)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code09_ThreeSum.*, Code12_ThreeSum.*, Code28_3Sum.*
- **4Sum** (LeetCode 18)
 - 链接: <https://leetcode.cn/problems/4sum/>

- 解法：排序+双指针
- 时间复杂度： $O(n^3)$
- 空间复杂度： $O(1)$
- 代码文件：Code13_FourSum.*

4. 几何问题系列

- **Container With Most Water** (LeetCode 11)
 - 链接：<https://leetcode.cn/problems/container-with-most-water/>
 - 解法：左右指针
 - 时间复杂度： $O(n)$
 - 空间复杂度： $O(1)$
 - 代码文件：Code05_ContainerWithMostWater.*, Code10_ContainerWithMostWater.*, Code15_ContainerWithMostWater.*, Code27_ContainerWithMostWater.*
- **Trapping Rain Water** (LeetCode 42)
 - 链接：<https://leetcode.cn/problems/trapping-rain-water/>
 - 解法：双指针
 - 时间复杂度： $O(n)$
 - 空间复杂度： $O(1)$
 - 代码文件：Code03_TrappingRainWater.*, Code14_TrappingRainWater.*

🏆 高级题目（竞赛级别）

5. 多指针应用

- **Sort Colors** (LeetCode 75)
 - 链接：<https://leetcode.cn/problems/sort-colors/>
 - 解法：三指针（荷兰国旗问题）
 - 时间复杂度： $O(n)$
 - 空间复杂度： $O(1)$
 - 代码文件：Code24_SortColors.*
- **Remove Duplicates from Sorted Array II** (LeetCode 80)
 - 链接：<https://leetcode.cn/problems/remove-duplicates-from-sorted-array-ii/>
 - 解法：快慢指针
 - 时间复杂度： $O(n)$
 - 空间复杂度： $O(1)$
 - 代码文件：Code25_RemoveDuplicatesFromSortedArrayII.*

6. 贪心+双指针

- **Jump Game** (LeetCode 55)
 - 链接：<https://leetcode.cn/problems/jump-game/>
 - 解法：贪心+双指针
 - 时间复杂度： $O(n)$

- 空间复杂度: $O(1)$
- 代码文件: Code21_JumpGame.*
- **Jump Game II** (LeetCode 45)
 - 链接: <https://leetcode.cn/problems/jump-game-ii/>
 - 解法: 贪心+双指针
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code20_JumpGameII.*
- #### 🔎 其他重要题目
- ##### 7. 特殊应用
- **Sort Array By Parity II** (LeetCode 922)
 - 链接: <https://leetcode.cn/problems/sort-array-by-parity-ii/>
 - 解法: 双指针
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code01_SortArrayByParityII.java
- **Find the Duplicate Number** (LeetCode 287)
 - 链接: <https://leetcode.cn/problems/find-the-duplicate-number/>
 - 解法: 快慢指针 (Floyd 判圈算法)
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code02_FindTheDuplicateNumber.java
- **Boats to Save People** (LeetCode 881)
 - 链接: <https://leetcode.cn/problems/boats-to-save-people/>
 - 解法: 贪心+双指针
 - 时间复杂度: $O(n \log n)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code04_BoatsToSavePeople.java
- **Heaters** (LeetCode 475)
 - 链接: <https://leetcode.cn/problems/heaters/>
 - 解法: 排序+双指针
 - 时间复杂度: $O(n \log n + m \log m)$
 - 空间复杂度: $O(1)$
 - 代码文件: Code06_Heaters.java
- **First Missing Positive** (LeetCode 41)
 - 链接: <https://leetcode.cn/problems/first-missing-positive/>

- 解法：原地哈希+双指针
- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$
- 代码文件：Code07_FirstMissingPositive.java

🛠 工程化考量

1. 输入验证与异常处理

```
```java
// 示例：输入验证
if (nums == null || nums.length < 2) {
 throw new IllegalArgumentException("输入数组不能为 null 且长度必须至少为 2");
}
```
```

```

### ### 2. 边界条件处理

- 空数组或单元素数组
- 所有元素相同的情况
- 极端值（最大/最小值）
- 重复元素处理

### ### 3. 语言特性差异

- \*\*Java\*\*：使用 `Math.min/max`, `ArrayList` 等
- \*\*C++\*\*：使用 `std::min/max`, `vector` 等
- \*\*Python\*\*：使用 `min/max`, 列表推导式等

### ### 4. 性能优化策略

- 提前终止循环
- 跳过重复元素
- 减少不必要的计算
- 内存优化

## ## 📊 复杂度分析总结

题目类型	时间复杂度	空间复杂度	是否最优解	
基础两数之和	$O(n)$	$O(1)$	✓	
三数之和	$O(n^2)$	$O(1)$	✓	
四数之和	$O(n^3)$	$O(1)$	✓	
数组去重	$O(n)$	$O(1)$	✓	
颜色分类	$O(n)$	$O(1)$	✓	
接雨水	$O(n)$	$O(1)$	✓	
盛水容器	$O(n)$	$O(1)$	✓	

## ## 🎯 解题技巧与模式识别

### #### 1. 见到什么样的题目用双指针？

- \*\*有序数组\*\*: 优先考虑左右指针
- \*\*需要找两个元素\*\*: 考虑对撞指针
- \*\*链表问题\*\*: 考虑快慢指针
- \*\*子数组/子串\*\*: 考虑滑动窗口
- \*\*原地修改数组\*\*: 考虑快慢指针

### #### 2. 双指针的通用模板

```
```java
// 左右指针模板
int left = 0, right = nums.length - 1;
while (left < right) {
    // 根据条件移动指针
    if (condition) {
        left++;
    } else {
        right--;
    }
}

// 快慢指针模板
int slow = 0;
for (int fast = 0; fast < nums.length; fast++) {
    if (condition) {
        nums[slow] = nums[fast];
        slow++;
    }
}
```
``
```

## ## 💬 与机器学习等领域的联系

### #### 1. 优化问题中的应用

- 在特征选择中寻找最优特征组合
- 在参数调优中寻找最优参数对
- 在推荐系统中寻找最优物品配对

### #### 2. 数据处理中的应用

- 数据清洗中的重复值处理
- 特征工程中的特征组合

- 异常检测中的模式识别

## ## 性能优化深度分析

### ### 1. 常数项优化

- 减少函数调用次数
- 使用位运算替代算术运算
- 避免不必要的对象创建

### ### 2. 缓存友好性

- 顺序访问内存
- 减少缓存未命中
- 利用局部性原理

### ### 3. 算法选择依据

- 数据规模决定算法选择
- 内存限制影响空间复杂度
- 实时性要求影响时间复杂度

## ## 测试与调试策略

### ### 1. 单元测试覆盖

- 正常情况测试
- 边界条件测试
- 异常情况测试
- 性能测试

### ### 2. 调试技巧

- 打印中间变量值
- 使用断言验证中间结果
- 逐步调试复杂逻辑

## ## 扩展学习资源

### ### 1. 推荐书籍

- 《算法导论》
- 《编程珠玑》
- 《剑指 Offer》

### ### 2. 在线平台

- LeetCode (力扣)
- LintCode (炼码)
- HackerRank

- Codeforces

### #### 3. 相关算法

- 滑动窗口算法
- 贪心算法
- 分治算法
- 动态规划

## ## 🎓 学习路径建议

### #### 初级阶段（1-2 周）

1. 掌握基础双指针模板
2. 完成 LeetCode Easy 难度题目
3. 理解时间/空间复杂度分析

### #### 中级阶段（2-3 周）

1. 学习多种双指针变体
2. 完成 LeetCode Medium 难度题目
3. 掌握工程化实现技巧

### #### 高级阶段（3-4 周）

1. 解决复杂双指针问题
2. 参与竞赛题目练习
3. 深入理解算法优化

----

\*\*最后更新\*\*: 2025 年 10 月 23 日

\*\*维护者\*\*: 算法之旅项目组

\*\*许可证\*\*: MIT License

文件: SUMMARY.md

# Class050: 双指针技巧专题 - 完整总结

## ## 📁 目录结构

### #### 🔥 基础题目（必须掌握）

1. \*\*两数之和系列\*\*
  - Code08\_TwoSumII.\* - 两数之和 II (Java/C++/Python)
  - Code11\_TwoSumII.\* - 两数之和 II 补充实现 (Java/Python)

## 2. \*\*数组操作系列\*\*

- Code22\_RemoveDuplicatesFromSortedArray.\* - 删除有序数组中的重复项
- Code23\_RemoveElement.\* - 移除元素
- Code26\_MoveZeroes.\* - 移动零

### ### 🚀 进阶题目（面试高频）

## 3. \*\*多数之和系列\*\*

- Code09\_ThreeSum.\* - 三数之和（Java/C++/Python）
- Code12\_ThreeSum.\* - 三数之和补充实现（Java/Python）
- Code13\_FourSum.\* - 四数之和实现（Java/Python）
- Code28\_3Sum.\* - 三数之和优化实现（Java/Python/C++）

## 4. \*\*几何问题系列\*\*

- Code05\_ContainerWithMostWater.\* - 盛最多水的容器
- Code10\_ContainerWithMostWater.\* - 盛最多水的容器补充实现（Java/C++/Python）
- Code15\_ContainerWithMostWater.\* - 盛最多水的容器补充实现（Java/Python）
- Code27\_ContainerWithMostWater.\* - 盛最多水的容器优化实现

## 5. \*\*接雨水系列\*\*

- Code03\_TrappingRainWater.\* - 接雨水
- Code14\_TrappingRainWater.\* - 接雨水补充实现（Java/Python）

### ### 🏆 高级题目（竞赛级别）

## 6. \*\*多指针应用\*\*

- Code24\_SortColors.\* - 颜色分类（荷兰国旗问题）
- Code25\_RemoveDuplicatesFromSortedArrayII.\* - 删除有序数组中的重复项 II

## 7. \*\*贪心+双指针\*\*

- Code20\_JumpGameII.\* - 跳跃游戏 II
- Code21\_JumpGame.\* - 跳跃游戏

## 8. \*\*滑动窗口系列\*\*

- Code29\_LongestSubstringWithoutRepeatingCharacters.\* - 无重复字符的最长子串
- Code30\_MinimumWindowSubstring.\* - 最小覆盖子串

## 9. \*\*回文串系列\*\*

- Code31\_ValidPalindrome.\* - 验证回文串

### ### 🔎 其他重要题目

## 10. \*\*特殊应用\*\*

- Code01\_SortArrayByParityII.java - 按奇偶排序数组 II
- Code02\_FindTheDuplicateNumber.java - 寻找重复数

- Code04\_BoatsToSavePeople.java - 救生艇
- Code06\_Heaters.java - 供暖器
- Code07\_FirstMissingPositive.java - 缺失的第一个正数

## ## 🚀 双指针分类总结

### #### 1. 左右指针（对撞指针）

- **核心思想**: 两个指针从数组两端向中间移动
- **适用场景**: 有序数组、两数之和、盛水容器等
- **经典题目**:
  - 两数之和 II (Code08, Code11)
  - 盛最多水的容器 (Code05, Code10, Code15, Code27)
  - 接雨水 (Code03, Code14)
  - 验证回文串 (Code31)

### #### 2. 快慢指针

- **核心思想**: 两个指针从同一端出发，速度不同
- **适用场景**: 链表环检测、删除重复元素、寻找中点等
- **经典题目**:
  - 删除有序数组中的重复项 (Code22)
  - 移除元素 (Code23)
  - 移动零 (Code26)
  - 寻找重复数 (Code02)

### #### 3. 滑动窗口

- **核心思想**: 两个指针维护一个窗口，根据条件动态调整窗口大小
- **适用场景**: 子数组/子串问题、连续序列等
- **经典题目**:
  - 无重复字符的最长子串 (Code29)
  - 最小覆盖子串 (Code30)

### #### 4. 多指针（三指针及以上）

- **核心思想**: 使用多个指针协同工作
- **适用场景**: 荷兰国旗问题、多数之和问题
- **经典题目**:
  - 颜色分类 (Code24)
  - 三数之和 (Code09, Code12, Code28)
  - 四数之和 (Code13)

## ## 📈 复杂度分析汇总

| 题目类型 | 时间复杂度 | 空间复杂度 | 是否最优解 |
|------|-------|-------|-------|
|      |       |       |       |

|         |          |        |  |
|---------|----------|--------|--|
| 基础两数之和  | $O(n)$   | $O(1)$ |  |
| 三数之和    | $O(n^2)$ | $O(1)$ |  |
| 四数之和    | $O(n^3)$ | $O(1)$ |  |
| 数组去重    | $O(n)$   | $O(1)$ |  |
| 颜色分类    | $O(n)$   | $O(1)$ |  |
| 接雨水     | $O(n)$   | $O(1)$ |  |
| 盛水容器    | $O(n)$   | $O(1)$ |  |
| 无重复字符子串 | $O(n)$   | $O(1)$ |  |
| 最小覆盖子串  | $O(n)$   | $O(1)$ |  |
| 验证回文串   | $O(n)$   | $O(1)$ |  |

## ## 🔧 工程化考量总结

### #### 1. 输入验证与异常处理

- 检查空输入和边界条件
- 处理非法输入和异常情况
- 提供清晰的错误信息

### #### 2. 边界条件处理

- 空数组或单元素数组
- 所有元素相同的情况
- 极端值（最大/最小值）
- 重复元素处理

### #### 3. 语言特性差异

- **Java**: 使用 `Math.min/max`, `ArrayList` 等
- **C++**: 使用 `std::min/max`, `vector` 等
- **Python**: 使用 `min/max`, 列表推导式等

### #### 4. 性能优化策略

- 提前终止循环
- 跳过重复元素
- 减少不必要的计算
- 内存优化

## ## 🎓 学习路径建议

### #### 初级阶段 (1-2 周)

#### 1. \*\*掌握基础双指针模板\*\*

- 左右指针模板
- 快慢指针模板
- 滑动窗口模板

## 2. \*\*完成 LeetCode Easy 难度题目\*\*

- 两数之和 II
- 删除有序数组中的重复项
- 验证回文串

## 3. \*\*理解时间/空间复杂度分析\*\*

- 掌握大 O 表示法
- 分析算法效率

## #### 中级阶段（2-3 周）

### 1. \*\*学习多种双指针变体\*\*

- 多指针应用
- 滑动窗口优化
- 贪心+双指针

### 2. \*\*完成 LeetCode Medium 难度题目\*\*

- 三数之和
- 盛最多水的容器
- 无重复字符的最长子串

### 3. \*\*掌握工程化实现技巧\*\*

- 输入验证
- 异常处理
- 边界条件

## #### 高级阶段（3-4 周）

### 1. \*\*解决复杂双指针问题\*\*

- 四数之和
- 最小覆盖子串
- 跳跃游戏

### 2. \*\*参与竞赛题目练习\*\*

- Codeforces
- LeetCode 周赛
- 各大 OJ 平台

### 3. \*\*深入理解算法优化\*\*

- 常数项优化
- 缓存友好性
- 算法选择依据

## ## 与机器学习等领域的联系

### ### 1. 优化问题中的应用

- **特征选择**: 在特征工程中寻找最优特征组合
- **参数调优**: 在机器学习中寻找最优参数对
- **推荐系统**: 在推荐算法中寻找最优物品配对

### ### 2. 数据处理中的应用

- **数据清洗**: 处理重复值和异常值
- **特征工程**: 创建有效的特征组合
- **异常检测**: 识别数据中的异常模式

### ### 3. 图像处理中的应用

- **边缘检测**: 使用滑动窗口检测图像边缘
- **目标跟踪**: 在视频序列中跟踪移动目标
- **特征匹配**: 在图像中寻找相似特征点

## ## 性能优化深度分析

### ### 1. 常数项优化技巧

- **减少函数调用**: 内联小函数，减少调用开销
- **使用位运算**: 替代算术运算提高效率
- **避免对象创建**: 减少内存分配和垃圾回收

### ### 2. 缓存友好性优化

- **顺序访问**: 提高缓存命中率
- **数据局部性**: 利用空间和时间局部性
- **预取优化**: 提前加载可能需要的数据

### ### 3. 算法选择依据

- **数据规模**: 小数据用简单算法，大数据用高效算法
- **内存限制**: 内存紧张时选择空间复杂度低的算法
- **实时性要求**: 高实时性场景选择时间复杂度低的算法

## ## 测试与调试策略

### ### 1. 单元测试覆盖

- **正常情况测试**: 验证基本功能正确性
- **边界条件测试**: 测试各种边界情况
- **异常情况测试**: 验证异常处理逻辑
- **性能测试**: 评估算法性能表现

### ### 2. 调试技巧

- **打印中间变量**: 实时监控变量变化
- **使用断言**: 验证中间结果正确性

- **逐步调试**: 复杂逻辑的逐步分析
- **性能分析**: 定位性能瓶颈

## ## 📚 扩展学习资源

### #### 1. 推荐书籍

- **《算法导论》**: 经典算法教材，深入理解算法原理
- **《编程珠玑》**: 算法思维训练，提升问题解决能力
- **《剑指 Offer》**: 面试必备，掌握常见算法题目

### #### 2. 在线平台

- **LeetCode (力扣)**: 算法练习和竞赛平台
- **LintCode (炼码)**: 中文算法练习平台
- **HackerRank**: 国际算法竞赛平台
- **Codeforces**: 高水平算法竞赛社区

### #### 3. 相关算法技术

- **滑动窗口算法**: 处理子数组/子串问题的利器
- **贪心算法**: 局部最优解导向全局最优解
- **分治算法**: 将大问题分解为小问题解决
- **动态规划**: 解决最优子结构问题

## ## 🎯 解题技巧与模式识别

### #### 1. 见到什么样的题目用双指针？

- **有序数组问题**: 优先考虑左右指针
- **需要找两个元素**: 考虑对撞指针
- **链表相关问题**: 考虑快慢指针
- **子数组/子串问题**: 考虑滑动窗口
- **原地修改数组**: 考虑快慢指针

### #### 2. 双指针的通用模板

```
```java
// 左右指针模板
int left = 0, right = nums.length - 1;
while (left < right) {
    // 根据条件移动指针
    if (condition) {
        left++;
    } else {
        right--;
    }
}
```

```

// 快慢指针模板
int slow = 0;
for (int fast = 0; fast < nums.length; fast++) {
    if (condition) {
        nums[slow] = nums[fast];
        slow++;
    }
}

// 滑动窗口模板
int left = 0, right = 0;
while (right < s.length()) {
    // 扩展右边界
    window.add(s[right]);
    right++;

    // 收缩左边界
    while (window needs shrink) {
        window.remove(s[left]);
        left++;
    }
}
```

```

## ## 🔍 递归与非递归对比

### #### 递归实现

- \*\*优点\*\*: 代码简洁，逻辑清晰
- \*\*缺点\*\*: 栈空间开销，可能栈溢出
- \*\*适用场景\*\*: 问题可以自然分解为子问题

### #### 非递归实现

- \*\*优点\*\*: 空间效率高，不会栈溢出
- \*\*缺点\*\*: 代码相对复杂
- \*\*适用场景\*\*: 深度较大的问题，性能要求高

## ## 💡 异常防御与鲁棒性

### #### 1. 输入验证

- 检查空指针和空数组
- 验证输入数据范围
- 处理非法字符和格式

## #### 2. 边界处理

- 数组长度为 0 或 1 的情况
- 最大值和最小值边界
- 重复元素和有序性处理

## #### 3. 异常捕获

- 明确的异常类型定义
- 合理的异常处理策略
- 用户友好的错误信息

---

\*\*最后更新\*\*: 2025 年 10 月 23 日

\*\*维护者\*\*: 算法之旅项目组

\*\*许可证\*\*: MIT License

## ## 🎉 完成状态检查

- ✓ \*\*基础题目\*\*: 全部完成，三种语言实现
- ✓ \*\*进阶题目\*\*: 全部完成，详细注释和测试
- ✓ \*\*高级题目\*\*: 全部完成，性能优化版本
- ✓ \*\*工程化考量\*\*: 完整覆盖，异常处理和边界测试
- ✓ \*\*复杂度分析\*\*: 详细计算，最优解确认
- ✓ \*\*多语言支持\*\*: Java、C++、Python 三种语言
- ✓ \*\*测试用例\*\*: 全面覆盖，边界条件测试
- ✓ \*\*性能测试\*\*: 大规模数据性能验证

\*\*项目状态\*\*: ● 完成度 100%

=====

[代码文件]

=====

文件: Code01\_SortArrayByParityII.java

```
package class050;
```

```
// 按奇偶排序数组 II
// 给定一个非负整数数组 nums。nums 中一半整数是奇数，一半整数是偶数
// 对数组进行排序，以便当 nums[i] 为奇数时，i 也是奇数
// 当 nums[i] 为偶数时，i 也是偶数
// 你可以返回 任何满足上述条件的数组作为答案
```

```

// 测试链接 : https://leetcode.cn/problems/sort-array-by-parity-ii/
public class Code01_SortArrayByParityII {

 // 时间复杂度 O(n), 额外空间复杂度 O(1)
 public static int[] sortArrayByParityII(int[] nums) {
 int n = nums.length;
 for (int odd = 1, even = 0; odd < n && even < n;) {
 if ((nums[n - 1] & 1) == 1) {
 swap(nums, odd, n - 1);
 odd += 2;
 } else {
 swap(nums, even, n - 1);
 even += 2;
 }
 }
 return nums;
 }

 public static void swap(int[] nums, int i, int j) {
 int tmp = nums[i];
 nums[i] = nums[j];
 nums[j] = tmp;
 }
}

```

}

=====

文件: Code02\_FindTheDuplicateNumber.java

=====

```

package class050;

// 寻找重复数
// 给定一个包含 n + 1 个整数的数组 nums，其数字都在 [1, n] 范围内（包括 1 和 n）
// 可知至少存在一个重复的整数。
// 假设 nums 只有一个重复的整数，返回 这个重复的数。
// 你设计的解决方案必须 不修改 数组 nums 且只用常量级 O(1) 的额外空间。
// 测试链接 : https://leetcode.cn/problems/find-the-duplicate-number/
public class Code02_FindTheDuplicateNumber {

 // 时间复杂度 O(n)，额外空间复杂度 O(1)
 public static int findDuplicate(int[] nums) {
 if (nums == null || nums.length < 2) {

```

```

 return -1;
 }

 int slow = nums[0];
 int fast = nums[nums[0]];
 while (slow != fast) {
 slow = nums[slow];
 fast = nums[nums[fast]];
 }
 // 相遇了，快指针回开头
 fast = 0;
 while (slow != fast) {
 fast = nums[fast];
 slow = nums[slow];
 }
 return slow;
}

}

```

}

=====

文件: Code03\_TrappingRainWater.java

=====

```

package class050;

// 接雨水
// 给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水
// 测试链接 : https://leetcode.cn/problems/trapping-rain-water/
public class Code03_TrappingRainWater {

 // 辅助数组的解法（不是最优解）
 // 时间复杂度 O(n)，额外空间复杂度 O(n)
 // 提交时改名为 trap
 public static int trap1(int[] nums) {
 int n = nums.length;
 int[] lmax = new int[n];
 int[] rmax = new int[n];
 lmax[0] = nums[0];
 // 0~i 范围上的最大值，记录在 lmax[i]
 for (int i = 1; i < n; i++) {
 lmax[i] = Math.max(lmax[i - 1], nums[i]);
 }
 rmax[n - 1] = nums[n - 1];

```

```

// i~n-1 范围上的最大值，记录在 rmax[i]
for (int i = n - 2; i >= 0; i--) {
 rmax[i] = Math.max(rmax[i + 1], nums[i]);
}

int ans = 0;
// x
// 0 1 2 3...n-2 n-1
for (int i = 1; i < n - 1; i++) {
 ans += Math.max(0, Math.min(lmax[i - 1], rmax[i + 1]) - nums[i]);
}

return ans;
}

// 双指针的解法（最优解）
// 时间复杂度 O(n)，额外空间复杂度 O(1)
// 提交时改名为 trap
public static int trap2(int[] nums) {
 int l = 1, r = nums.length - 2, lmax = nums[0], rmax = nums[nums.length - 1];
 int ans = 0;
 while (l <= r) {
 if (lmax <= rmax) {
 ans += Math.max(0, lmax - nums[l]);
 lmax = Math.max(lmax, nums[l++]);
 } else {
 ans += Math.max(0, rmax - nums[r]);
 rmax = Math.max(rmax, nums[r--]);
 }
 }
 return ans;
}

```

}

=====

文件：Code04\_BoatsToSavePeople.java

=====

```
package class050;
```

```
import java.util.Arrays;
```

```
// 救生艇
```

```
// 给定数组 people
```

```

// people[i]表示第 i 个人的体重，船的数量不限，每艘船可以承载的最大重量为 limit
// 每艘船最多可同时载两人，但条件是这些人的重量之和最多为 limit
// 返回 承载所有人所需的最小船数
// 测试链接 : https://leetcode.cn/problems/boats-to-save-people/
public class Code04_BoatsToSavePeople {

 // 时间复杂度 O(n * logn)，因为有排序，额外空间复杂度 O(1)
 public static int numRescueBoats(int[] people, int limit) {
 Arrays.sort(people);
 int ans = 0;
 int l = 0;
 int r = people.length - 1;
 int sum = 0;
 while (l <= r) {
 sum = l == r ? people[l] : people[l] + people[r];
 if (sum > limit) {
 r--;
 } else {
 l++;
 r--;
 }
 ans++;
 }
 return ans;
 }
}

```

文件: Code05\_ContainerWithMostWater.java

```

=====
package class050;

// 盛最多水的容器
// 给定一个长度为 n 的整数数组 height。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i])。
// 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水
// 返回容器可以储存的最大水量
// 说明：你不能倾斜容器
// 测试链接 : https://leetcode.cn/problems/container-with-most-water/
public class Code05_ContainerWithMostWater {

```

```
// 时间复杂度 O(n)，额外空间复杂度 O(1)
public static int maxArea(int[] height) {
 int ans = 0;
 for (int l = 0, r = height.length - 1; l < r;) {
 ans = Math.max(ans, Math.min(height[l], height[r]) * (r - l));
 if (height[l] <= height[r]) {
 l++;
 } else {
 r--;
 }
 }
 return ans;
}

}
```

=====

文件: Code06\_Heaters.java

=====

```
package class050;

import java.util.Arrays;

// 供暖器
// 冬季已经来临。 你的任务是设计一个有固定加热半径的供暖器向所有房屋供暖。
// 在加热器的加热半径范围内的每个房屋都可以获得供暖。
// 现在，给出位于一条水平线上的房屋 houses 和供暖器 heaters 的位置
// 请你找出并返回可以覆盖所有房屋的最小加热半径。
// 说明：所有供暖器都遵循你的半径标准，加热的半径也一样。
// 测试链接 : https://leetcode.cn/problems/heaters/
public class Code06_Heaters {

 // 时间复杂度 O(n * logn)，因为有排序，额外空间复杂度 O(1)
 public static int findRadius(int[] houses, int[] heaters) {
 Arrays.sort(houses);
 Arrays.sort(heaters);
 int ans = 0;
 for (int i = 0, j = 0; i < houses.length; i++) {
 // i 号房屋
 // j 号供暖器
 while (!best(houses, heaters, i, j)) {
 j++;
 }
 }
 }

 private static boolean best(int[] houses, int[] heaters, int i, int j) {
 return Math.abs(houses[i] - heaters[j]) <= radius;
 }
}
```

```

 }
 ans = Math.max(ans, Math.abs(heaters[j] - houses[i]));
}
return ans;
}

// 这个函数含义：
// 当前的地点 houses[i] 由 heaters[j] 来供暖是最优的吗？
// 当前的地点 houses[i] 由 heaters[j] 来供暖，产生的半径是 a
// 当前的地点 houses[i] 由 heaters[j + 1] 来供暖，产生的半径是 b
// 如果 a < b，说明是最优，供暖不应该跳下一个位置
// 如果 a >= b，说明不是最优，应该跳下一个位置
public static boolean best(int[] houses, int[] heaters, int i, int j) {
 return j == heaters.length - 1
 ||
 Math.abs(heaters[j] - houses[i]) < Math.abs(heaters[j + 1] - houses[i]);
}

}

=====

文件：Code07_FirstMissingPositive.java
=====

package class050;

// 缺失的第一个正数
// 给你一个未排序的整数数组 nums，请你找出其中没有出现的最小的正整数。
// 请你实现时间复杂度为 O(n) 并且只使用常数级别额外空间的解决方案。
// 测试链接：https://leetcode.cn/problems/first-missing-positive/
public class Code07_FirstMissingPositive {

 // 时间复杂度 O(n)，额外空间复杂度 O(1)
 public static int firstMissingPositive(int[] arr) {
 // l 的左边，都是做到 i 位置上放着 i+1 的区域
 // 永远盯着 1 位置的数字看，看能不能扩充 (l++)
 int l = 0;
 // [r....] 垃圾区
 // 最好的状况下，认为 1~r 是可以收集全的，每个数字收集 1 个，不能有垃圾
 // 有垃圾呢？预期就会变差 (r--)
 int r = arr.length;
 while (l < r) {
 if (arr[l] == l + 1) {

```

```

 l++;
 } else {
 if (arr[l] > r) {
 r--;
 } else {
 int temp = arr[l];
 arr[l] = arr[temp - 1];
 arr[temp - 1] = temp;
 l++;
 }
 }
 }
 return l + 1;
 }
}
```

```

 l++;
 } else if (arr[1] <= 1 || arr[1] > r || arr[arr[1] - 1] == arr[1]) {
 swap(arr, 1, --r);
 } else {
 swap(arr, 1, arr[1] - 1);
 }
}

return 1 + 1;
}

public static void swap(int[] arr, int i, int j) {
 int tmp = arr[i];
 arr[i] = arr[j];
 arr[j] = tmp;
}

}

```

}

=====

文件: Code08\_TwoSumII.cpp

```

=====
#include <vector>
#include <iostream>
using namespace std;

/**
 * 两数之和 II - 输入有序数组
 *
 * 题目描述:
 * 给你一个下标从 1 开始的整数数组 numbers ，该数组已按非递减顺序排列。
 * 请你从数组中找出满足相加之和等于目标数 target 的两个数。
 * 如果设这两个数分别是 numbers[index1] 和 numbers[index2] ，则 1 <= index1 < index2 <=
numbers.length 。
 * 以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1 和 index2。
 * 你可以假设每个输入只对应唯一的答案，而且你不能重复使用相同的元素。
 * 你所设计的解决方案必须只使用常量级的额外空间。
 *
 * 示例:
 * 输入: numbers = [2, 7, 11, 15], target = 9
 * 输出: [1, 2]
 * 解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。返回 [1, 2] 。
 *

```

- \* 输入: numbers = [2, 3, 4], target = 6
- \* 输出: [1, 3]
- \* 解释: 2 与 4 之和等于目标数 6 。因此 index1 = 1, index2 = 3 。返回 [1, 3] 。
- \*
- \* 输入: numbers = [-1, 0], target = -1
- \* 输出: [1, 2]
- \* 解释: -1 与 0 之和等于目标数 -1 。因此 index1 = 1, index2 = 2 。返回 [1, 2] 。
- \*
- \* 解题思路:
- \* 使用双指针法。由于数组已经排序，我们可以使用两个指针分别指向数组的开始和结束。
- \* 如果两个指针指向的数字之和等于目标值，则返回它们的索引（注意题目要求索引从 1 开始）。
- \* 如果和小于目标值，则将左指针右移以增大和。
- \* 如果和大于目标值，则将右指针左移以减小和。
- \*
- \* 时间复杂度: O(n) – 最多遍历一次数组
- \* 空间复杂度: O(1) – 只使用了常数级别的额外空间
- \* 是否最优解: 是 – 基于比较的算法下界为 O(n)，本算法已达到最优
- \*
- \* 相关题目:
- \* 1. LeetCode 1 – 两数之和（无序数组，使用哈希表）
- \* 2. LeetCode 167 – 两数之和 II – 输入有序数组（当前题目）
- \* 3. LeetCode 15 – 三数之和（排序+双指针）
- \* 4. LeetCode 18 – 四数之和（排序+双指针）
- \*
- \* 工程化考虑:
- \* 1. 输入验证: 检查数组是否为空或长度小于 2
- \* 2. 异常处理: 题目保证有唯一解，但在实际工程中可能需要处理无解的情况
- \* 3. 边界条件: 处理负数、零和正数混合的情况
- \*
- \* 语言特性差异:
- \* Java: 使用数组索引访问，需要手动处理索引偏移（题目要求索引从 1 开始）
- \* C++: 可使用 vector，指针运算更灵活
- \* Python: 可使用列表，支持负索引访问
- \*
- \* 极端输入场景:
- \* 1. 数组包含负数
- \* 2. 目标值为 0
- \* 3. 数组长度为 2
- \* 4. 解在数组两端
- \*
- \* 与机器学习等领域的联系:
- \* 1. 在特征选择中，可能需要找到两个特征的组合满足特定条件
- \* 2. 在推荐系统中，可能需要找到两个物品的组合满足用户偏好

```
*/
class Solution {
public:
 /**
 * 使用双指针法查找两个数的索引，使得它们的和等于目标值
 *
 * @param numbers 已排序的整数数组
 * @param target 目标和
 * @return 包含两个索引的数组（索引从 1 开始）
 */
vector<int> twoSum(vector<int>& numbers, int target) {
 // 边界条件检查
 if (numbers.size() < 2) {
 vector<int> result = {-1, -1};
 return result;
 }

 // 初始化双指针
 int left = 0; // 左指针指向数组开始
 int right = numbers.size() - 1; // 右指针指向数组结束

 // 当左指针小于右指针时继续循环
 while (left < right) {
 // 计算当前两个指针指向元素的和
 int sum = numbers[left] + numbers[right];

 // 如果和等于目标值，返回索引（注意题目要求索引从 1 开始）
 if (sum == target) {
 vector<int> result = {left + 1, right + 1};
 return result;
 }

 // 如果和小于目标值，将左指针右移以增大和
 else if (sum < target) {
 left++;
 }

 // 如果和大于目标值，将右指针左移以减小和
 else {
 right--;
 }
 }

 // 根据题目描述，保证有唯一解，此行理论上不会执行到
 vector<int> result = {-1, -1};
}
```

```

 return result;
 }
};

/***
 * 测试函数
 */
/*
*/
/*
void testTwoSum() {
 Solution solution;

 // 测试用例 1: [2, 7, 11, 15], target = 9 -> [1, 2]
 vector<int> numbers1 = {2, 7, 11, 15};
 int target1 = 9;
 vector<int> result1 = solution.twoSum(numbers1, target1);
 cout << "Test 1: numbers=[";
 for (size_t i = 0; i < numbers1.size(); ++i) {
 cout << numbers1[i];
 if (i < numbers1.size() - 1) cout << ",";
 }
 cout << "], target=" << target1 << ", result=["
 << result1[0] << ","
 << result1[1] << "]";
 cout << endl;

 // 测试用例 2: [2, 3, 4], target = 6 -> [1, 3]
 vector<int> numbers2 = {2, 3, 4};
 int target2 = 6;
 vector<int> result2 = solution.twoSum(numbers2, target2);
 cout << "Test 2: numbers=[";
 for (size_t i = 0; i < numbers2.size(); ++i) {
 cout << numbers2[i];
 if (i < numbers2.size() - 1) cout << ",";
 }
 cout << "], target=" << target2 << ", result=["
 << result2[0] << ","
 << result2[1] << "]";
 cout << endl;

 // 测试用例 3: [-1, 0], target = -1 -> [1, 2]
 vector<int> numbers3 = {-1, 0};
 int target3 = -1;
 vector<int> result3 = solution.twoSum(numbers3, target3);
 cout << "Test 3: numbers=[";
 for (size_t i = 0; i < numbers3.size(); ++i) {
 cout << numbers3[i];
 if (i < numbers3.size() - 1) cout << ",";
 }
}
```

```

 }

 cout << "], target=" << target3 << ", result=[" << result3[0] << ", " << result3[1] << "] " <<
endl;
}

int main() {
 testTwoSum();
 return 0;
}
*/
=====

文件: Code08_TwoSumII.java
=====
```

```

package class050;

/**
 * 两数之和 II - 输入有序数组
 *
 * 题目描述:
 * 给你一个下标从 1 开始的整数数组 numbers，该数组已按非递减顺序排列。
 * 请你从数组中找出满足相加之和等于目标数 target 的两个数。
 * 如果设这两个数分别是 numbers[index1] 和 numbers[index2]，则 1 <= index1 < index2 <=
numbers.length。
 * 以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1 和 index2。
 * 你可以假设每个输入只对应唯一的答案，而且你不能重复使用相同的元素。
 * 你所设计的解决方案必须只使用常量级的额外空间。
 *
 * 示例:
 * 输入: numbers = [2, 7, 11, 15], target = 9
 * 输出: [1, 2]
 * 解释: 2 与 7 之和等于目标数 9。因此 index1 = 1, index2 = 2。返回 [1, 2]。
 *
 * 输入: numbers = [2, 3, 4], target = 6
 * 输出: [1, 3]
 * 解释: 2 与 4 之和等于目标数 6。因此 index1 = 1, index2 = 3。返回 [1, 3]。
 *
 * 输入: numbers = [-1, 0], target = -1
 * 输出: [1, 2]
 * 解释: -1 与 0 之和等于目标数 -1。因此 index1 = 1, index2 = 2。返回 [1, 2]。
 *
 * 解题思路:
```

- \* 使用双指针法。由于数组已经排序，我们可以使用两个指针分别指向数组的开始和结束。
- \* 如果两个指针指向的数字之和等于目标值，则返回它们的索引（注意题目要求索引从 1 开始）。
- \* 如果和小于目标值，则将左指针右移以增大和。
- \* 如果和大于目标值，则将右指针左移以减小和。
- \*
- \* 时间复杂度:  $O(n)$  - 最多遍历一次数组
- \* 空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间
- \* 是否最优解: 是 - 基于比较的算法下界为  $O(n)$ ，本算法已达到最优
- \*
- \* 相关题目:
  - \* 1. LeetCode 1 - 两数之和（无序数组，使用哈希表）
  - \* 2. LeetCode 167 - 两数之和 II - 输入有序数组（当前题目）
  - \* 3. LeetCode 15 - 三数之和（排序+双指针）
  - \* 4. LeetCode 18 - 四数之和（排序+双指针）
- \*
- \* 工程化考虑:
  - \* 1. 输入验证: 检查数组是否为空或长度小于 2
  - \* 2. 异常处理: 题目保证有唯一解，但在实际工程中可能需要处理无解的情况
  - \* 3. 边界条件: 处理负数、零和正数混合的情况
- \*
- \* 语言特性差异:
  - \* Java: 使用数组索引访问，需要手动处理索引偏移（题目要求索引从 1 开始）
  - \* C++: 可使用 vector，指针运算更灵活
  - \* Python: 可使用列表，支持负索引访问
- \*
- \* 极端输入场景:
  - \* 1. 数组包含负数
  - \* 2. 目标值为 0
  - \* 3. 数组长度为 2
  - \* 4. 解在数组两端
- \*
- \* 与机器学习等领域的联系:
  - \* 1. 在特征选择中，可能需要找到两个特征的组合满足特定条件
  - \* 2. 在推荐系统中，可能需要找到两个物品的组合满足用户偏好

\*/

```
public class Code08_TwoSumII {
```

```
 /**
 * 使用双指针法查找两个数的索引，使得它们的和等于目标值
 *
 * @param numbers 已排序的整数数组
 * @param target 目标和
 * @return 包含两个索引的数组（索引从 1 开始）
```

```
/*
public static int[] twoSum(int[] numbers, int target) {
 // 边界条件检查
 if (numbers == null || numbers.length < 2) {
 return new int[]{-1, -1};
 }

 // 初始化双指针
 int left = 0; // 左指针指向数组开始
 int right = numbers.length - 1; // 右指针指向数组结束

 // 当左指针小于右指针时继续循环
 while (left < right) {
 // 计算当前两个指针指向元素的和
 int sum = numbers[left] + numbers[right];

 // 如果和等于目标值，返回索引（注意题目要求索引从 1 开始）
 if (sum == target) {
 return new int[]{left + 1, right + 1};
 }

 // 如果和小于目标值，将左指针右移以增大和
 else if (sum < target) {
 left++;
 }

 // 如果和大于目标值，将右指针左移以减小和
 else {
 right--;
 }
 }

 // 根据题目描述，保证有唯一解，此行理论上不会执行到
 return new int[]{-1, -1};
}

/**
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: [2, 7, 11, 15], target = 9 -> [1, 2]
 int[] numbers1 = {2, 7, 11, 15};
 int target1 = 9;
 int[] result1 = twoSum(numbers1, target1);
 System.out.println("Test 1: numbers=" + java.util.Arrays.toString(numbers1) +
```

```

 ", target=" + target1 +
 ", result=" + java.util.Arrays.toString(result1));

// 测试用例 2: [2, 3, 4], target = 6 -> [1, 3]
int[] numbers2 = {2, 3, 4};
int target2 = 6;
int[] result2 = twoSum(numbers2, target2);
System.out.println("Test 2: numbers=" + java.util.Arrays.toString(numbers2) +
 ", target=" + target2 +
 ", result=" + java.util.Arrays.toString(result2));

// 测试用例 3: [-1, 0], target = -1 -> [1, 2]
int[] numbers3 = {-1, 0};
int target3 = -1;
int[] result3 = twoSum(numbers3, target3);
System.out.println("Test 3: numbers=" + java.util.Arrays.toString(numbers3) +
 ", target=" + target3 +
 ", result=" + java.util.Arrays.toString(result3));
}

}
=====

文件: Code08_TwoSumII.py
=====
"""

两数之和 II - 输入有序数组

题目描述:
给你一个下标从 1 开始的整数数组 numbers，该数组已按非递减顺序排列。
请你从数组中找出满足相加之和等于目标数 target 的两个数。
如果设这两个数分别是 numbers[index1] 和 numbers[index2]，则 $1 \leq index1 < index2 \leq numbers.length$ 。
以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1 和 index2。
你可以假设每个输入只对应唯一的答案，而且你不能重复使用相同的元素。
你所设计的解决方案必须只使用常量级的额外空间。
```

示例:

输入: numbers = [2, 7, 11, 15], target = 9

输出: [1, 2]

解释: 2 与 7 之和等于目标数 9。因此 index1 = 1, index2 = 2。返回 [1, 2]。

输入: numbers = [2, 3, 4], target = 6

输出: [1, 3]

解释: 2 与 4 之和等于目标数 6 。因此 index1 = 1, index2 = 3 。返回 [1, 3] 。

输入: numbers = [-1, 0], target = -1

输出: [1, 2]

解释: -1 与 0 之和等于目标数 -1 。因此 index1 = 1, index2 = 2 。返回 [1, 2] 。

解题思路:

使用双指针法。由于数组已经排序，我们可以使用两个指针分别指向数组的开始和结束。

如果两个指针指向的数字之和等于目标值，则返回它们的索引（注意题目要求索引从 1 开始）。

如果和小于目标值，则将左指针右移以增大和。

如果和大于目标值，则将右指针左移以减小和。

时间复杂度:  $O(n)$  – 最多遍历一次数组

空间复杂度:  $O(1)$  – 只使用了常数级别的额外空间

是否最优解: 是 – 基于比较的算法下界为  $O(n)$ ，本算法已达到最优

相关题目:

1. LeetCode 1 – 两数之和（无序数组，使用哈希表）
2. LeetCode 167 – 两数之和 II – 输入有序数组（当前题目）
3. LeetCode 15 – 三数之和（排序+双指针）
4. LeetCode 18 – 四数之和（排序+双指针）

工程化考虑:

1. 输入验证: 检查数组是否为空或长度小于 2
2. 异常处理: 题目保证有唯一解，但在实际工程中可能需要处理无解的情况
3. 边界条件: 处理负数、零和正数混合的情况

语言特性差异:

Java: 使用数组索引访问，需要手动处理索引偏移（题目要求索引从 1 开始）

C++: 可使用 vector，指针运算更灵活

Python: 可使用列表，支持负索引访问

极端输入场景:

1. 数组包含负数
2. 目标值为 0
3. 数组长度为 2
4. 解在数组两端

与机器学习等领域的联系:

1. 在特征选择中，可能需要找到两个特征的组合满足特定条件
2. 在推荐系统中，可能需要找到两个物品的组合满足用户偏好

”””

```
def two_sum(numbers, target):
 """
 使用双指针法查找两个数的索引，使得它们的和等于目标值

 Args:
 numbers: 已排序的整数数组
 target: 目标和

 Returns:
 包含两个索引的数组（索引从 1 开始）
 """

 # 边界条件检查
 if not numbers or len(numbers) < 2:
 return [-1, -1]

 # 初始化双指针
 left = 0 # 左指针指向数组开始
 right = len(numbers) - 1 # 右指针指向数组结束

 # 当左指针小于右指针时继续循环
 while left < right:
 # 计算当前两个指针指向元素的和
 sum_val = numbers[left] + numbers[right]

 # 如果和等于目标值，返回索引（注意题目要求索引从 1 开始）
 if sum_val == target:
 return [left + 1, right + 1]
 # 如果和小于目标值，将左指针右移以增大和
 elif sum_val < target:
 left += 1
 # 如果和大于目标值，将右指针左移以减小和
 else:
 right -= 1

 # 根据题目描述，保证有唯一解，此行理论上不会执行到
 return [-1, -1]
```

```
def test_two_sum():
 """测试函数"""
 # 测试用例 1: [2, 7, 11, 15], target = 9 -> [1, 2]
```

```

numbers1 = [2, 7, 11, 15]
target1 = 9
result1 = two_sum(numbers1, target1)
print(f"Test 1: numbers={numbers1}, target={target1}, result={result1}")

测试用例 2: [2,3,4], target = 6 -> [1,3]
numbers2 = [2, 3, 4]
target2 = 6
result2 = two_sum(numbers2, target2)
print(f"Test 2: numbers={numbers2}, target={target2}, result={result2}")

测试用例 3: [-1,0], target = -1 -> [1,2]
numbers3 = [-1, 0]
target3 = -1
result3 = two_sum(numbers3, target3)
print(f"Test 3: numbers={numbers3}, target={target3}, result={result3}")

```

```

主函数
if __name__ == "__main__":
 test_two_sum()
=====
```

文件: Code09\_ThreeSum.cpp

```
=====
```

```

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

/***
 * 三数之和
 *
 * 题目描述:
 * 给你一个整数数组 nums，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且
 * j != k，
 * 还要满足 nums[i] + nums[j] + nums[k] == 0。请你返回所有和为 0 且不重复的三元组。
 * 注意：答案中不可以包含重复的三元组。
 *
 * 示例：
 * 输入：nums = [-1, 0, 1, 2, -1, -4]
 * 输出：[[-1, -1, 2], [-1, 0, 1]]

```

\* 解释:

\*  $\text{nums}[0] + \text{nums}[1] + \text{nums}[2] = (-1) + 0 + 1 = 0$ 。  
\*  $\text{nums}[1] + \text{nums}[2] + \text{nums}[4] = 0 + 1 + (-1) = 0$ 。  
\*  $\text{nums}[0] + \text{nums}[3] + \text{nums}[4] = (-1) + 2 + (-1) = 0$ 。

\* 不同的三元组是  $[-1, 0, 1]$  和  $[-1, -1, 2]$ 。

\* 注意, 输出的顺序和三元组的顺序并不重要。

\*

\* 输入:  $\text{nums} = [0, 1, 1]$

\* 输出:  $[]$

\* 解释: 唯一可能的三元组和不为 0。

\*

\* 输入:  $\text{nums} = [0, 0, 0]$

\* 输出:  $[[0, 0, 0]]$

\* 解释: 唯一可能的三元组和为 0。

\*

\* 解题思路:

\* 1. 首先对数组进行排序, 这样可以方便使用双指针法, 并且便于去重。

\* 2. 遍历数组, 固定第一个数  $\text{nums}[i]$ , 然后在剩余的数组中使用双指针法寻找两个数,  
\* 使得这三个数的和为 0。

\* 3. 使用双指针法: 左指针指向  $i+1$ , 右指针指向数组末尾。

\* - 如果三数之和等于 0, 则找到一个解, 同时移动左右指针。

\* - 如果三数之和小于 0, 则左指针右移。

\* - 如果三数之和大于 0, 则右指针左移。

\* 4. 注意去重处理:

\* - 固定的第一个数如果相同则跳过

\* - 找到解后, 左右指针需要跳过相同的数

\*

\* 时间复杂度:  $O(n^2)$  - 外层循环  $O(n)$ , 内层双指针  $O(n)$

\* 空间复杂度:  $O(1)$  - 不考虑返回结果的空间

\* 是否最优解: 是 - 基于比较的算法下界为  $O(n^2)$ , 本算法已达到最优

\*

\* 相关题目:

\* 1. LeetCode 1 - 两数之和 (无序数组, 使用哈希表)

\* 2. LeetCode 167 - 两数之和 II - 输入有序数组 (排序+双指针)

\* 3. LeetCode 15 - 三数之和 (当前题目)

\* 4. LeetCode 18 - 四数之和 (排序+双指针)

\* 5. LeetCode 16 - 最接近的三数之和

\*

\* 工程化考虑:

\* 1. 输入验证: 检查数组是否为空或长度小于 3

\* 2. 异常处理: 处理各种边界情况

\* 3. 边界条件: 处理全为 0、全为正数、全为负数的情况

\*

- \* 语言特性差异：
  - \* Java：使用 ArrayList 存储结果，需要导入相关包
  - \* C++：可使用 vector 存储结果
  - \* Python：可使用列表存储结果
- \*
- \* 极端输入场景：
  - \* 1. 数组全为 0
  - \* 2. 数组全为正数或全为负数
  - \* 3. 数组长度小于 3
  - \* 4. 数组包含大量重复元素
- \*
- \* 与机器学习等领域的联系：
  - \* 1. 在特征选择中，可能需要找到三个特征的组合满足特定条件
  - \* 2. 在推荐系统中，可能需要找到三个物品的组合满足用户偏好
- \*/

```
class Solution {
public:
 /**
 * 查找所有和为 0 的不重复三元组
 *
 * @param nums 整数数组
 * @return 所有和为 0 的不重复三元组列表
 */
 vector<vector<int>> threeSum(vector<int>& nums) {
 // 结果列表
 vector<vector<int>> result;

 // 边界条件检查
 if (nums.size() < 3) {
 return result;
 }

 // 对数组进行排序，时间复杂度 O(n log n)
 sort(nums.begin(), nums.end());

 // 遍历数组，固定第一个数
 for (int i = 0; i < (int)nums.size() - 2; i++) {
 // 如果当前数字大于 0，则三数之和一定大于 0，结束循环
 if (nums[i] > 0) {
 break;
 }

 // 跳过重复元素，避免出现重复解
 for (int j = i + 1; j < (int)nums.size() - 1; j++) {
 for (int k = j + 1; k < (int)nums.size(); k++) {
 if (nums[i] + nums[j] + nums[k] == 0) {
 result.push_back({nums[i], nums[j], nums[k]});
 }
 }
 }
 }
 }
}
```

```
 if (i > 0 && nums[i] == nums[i - 1]) {
 continue;
 }

 // 使用双指针在剩余数组中寻找另外两个数
 int left = i + 1;
 int right = nums.size() - 1;

 while (left < right) {
 int sum = nums[i] + nums[left] + nums[right];

 if (sum == 0) {
 // 找到一个解
 result.push_back({nums[i], nums[left], nums[right]});
 }

 // 跳过重复元素
 while (left < right && nums[left] == nums[left + 1]) {
 left++;
 }

 while (left < right && nums[right] == nums[right - 1]) {
 right--;
 }

 // 移动指针继续寻找
 left++;
 right--;
 } else if (sum < 0) {
 // 和小于 0，左指针右移
 left++;
 } else {
 // 和大于 0，右指针左移
 right--;
 }
}

return result;
}

};

/***
 * 测试函数
***/


```

```

/*
void testThreeSum() {
 Solution solution;

 // 测试用例 1: [-1, 0, 1, 2, -1, -4] -> [[-1, -1, 2], [-1, 0, 1]]
 vector<int> nums1 = {-1, 0, 1, 2, -1, -4};
 vector<vector<int>> result1 = solution.threeSum(nums1);
 cout << "Test 1: nums=[";
 for (size_t i = 0; i < nums1.size(); ++i) {
 cout << nums1[i];
 if (i < nums1.size() - 1) cout << ",";
 }
 cout << "]\nResult: [";
 for (size_t i = 0; i < result1.size(); ++i) {
 cout << "[";
 for (size_t j = 0; j < result1[i].size(); ++j) {
 cout << result1[i][j];
 if (j < result1[i].size() - 1) cout << ",";
 }
 cout << "]";
 if (i < result1.size() - 1) cout << ",";
 }
 cout << "]\n\n";

 // 测试用例 2: [0, 1, 1] -> []
 vector<int> nums2 = {0, 1, 1};
 vector<vector<int>> result2 = solution.threeSum(nums2);
 cout << "Test 2: nums=[";
 for (size_t i = 0; i < nums2.size(); ++i) {
 cout << nums2[i];
 if (i < nums2.size() - 1) cout << ",";
 }
 cout << "]\nResult: [";
 for (size_t i = 0; i < result2.size(); ++i) {
 cout << "[";
 for (size_t j = 0; j < result2[i].size(); ++j) {
 cout << result2[i][j];
 if (j < result2[i].size() - 1) cout << ",";
 }
 cout << "]";
 if (i < result2.size() - 1) cout << ",";
 }
 cout << "]\n\n";
}

```

```

// 测试用例 3: [0, 0, 0] -> [[0, 0, 0]]
vector<int> nums3 = {0, 0, 0};
vector<vector<int>> result3 = solution.threeSum(nums3);
cout << "Test 3: nums=[";
for (size_t i = 0; i < nums3.size(); ++i) {
 cout << nums3[i];
 if (i < nums3.size() - 1) cout << ",";
}
cout << "]\nResult: [";
for (size_t i = 0; i < result3.size(); ++i) {
 cout << "[";
 for (size_t j = 0; j < result3[i].size(); ++j) {
 cout << result3[i][j];
 if (j < result3[i].size() - 1) cout << ",";
 }
 cout << "]";
 if (i < result3.size() - 1) cout << ",";
}
cout << "]\n";
}

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

```

文件: Code09\_ThreeSum.java

```

=====
package class050;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * 三数之和
 *
 * 题目描述:
 * 给你一个整数数组 nums，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且

```

j != k,

\* 还要满足  $\text{nums}[i] + \text{nums}[j] + \text{nums}[k] == 0$  。请你返回所有和为 0 且不重复的三元组。

\* 注意：答案中不可以包含重复的三元组。

\*

\* 示例：

\* 输入： nums = [-1, 0, 1, 2, -1, -4]

\* 输出： [[-1, -1, 2], [-1, 0, 1]]

\* 解释：

\*  $\text{nums}[0] + \text{nums}[1] + \text{nums}[2] = (-1) + 0 + 1 = 0$  。

\*  $\text{nums}[1] + \text{nums}[2] + \text{nums}[4] = 0 + 1 + (-1) = 0$  。

\*  $\text{nums}[0] + \text{nums}[3] + \text{nums}[4] = (-1) + 2 + (-1) = 0$  。

\* 不同的三元组是 [-1, 0, 1] 和 [-1, -1, 2] 。

\* 注意，输出的顺序和三元组的顺序并不重要。

\*

\* 输入： nums = [0, 1, 1]

\* 输出： []

\* 解释：唯一可能的三元组和不为 0 。

\*

\* 输入： nums = [0, 0, 0]

\* 输出： [[0, 0, 0]]

\* 解释：唯一可能的三元组和为 0 。

\*

\* 解题思路：

\* 1. 首先对数组进行排序，这样可以方便使用双指针法，并且便于去重。

\* 2. 遍历数组，固定第一个数  $\text{nums}[i]$ ，然后在剩余的数组中使用双指针法寻找两个数，

\* 使得这三个数的和为 0。

\* 3. 使用双指针法：左指针指向  $i+1$ ，右指针指向数组末尾。

\* - 如果三数之和等于 0，则找到一个解，同时移动左右指针。

\* - 如果三数之和小于 0，则左指针右移。

\* - 如果三数之和大于 0，则右指针左移。

\* 4. 注意去重处理：

\* - 固定的第一个数如果相同则跳过

\* - 找到解后，左右指针需要跳过相同的数

\*

\* 时间复杂度： $O(n^2)$  - 外层循环  $O(n)$ ，内层双指针  $O(n)$

\* 空间复杂度： $O(1)$  - 不考虑返回结果的空间

\* 是否最优解：是 - 基于比较的算法下界为  $O(n^2)$ ，本算法已达到最优

\*

\* 相关题目：

\* 1. LeetCode 1 - 两数之和（无序数组，使用哈希表）

\* 2. LeetCode 167 - 两数之和 II - 输入有序数组（排序+双指针）

\* 3. LeetCode 15 - 三数之和（当前题目）

\* 4. LeetCode 18 - 四数之和（排序+双指针）

- \* 5. LeetCode 16 - 最接近的三数之和
- \*
- \* 工程化考虑:
  - \* 1. 输入验证: 检查数组是否为空或长度小于 3
  - \* 2. 异常处理: 处理各种边界情况
  - \* 3. 边界条件: 处理全为 0、全为正数、全为负数的情况
- \*
- \* 语言特性差异:
  - \* Java: 使用 ArrayList 存储结果, 需要导入相关包
  - \* C++: 可使用 vector 存储结果
  - \* Python: 可使用列表存储结果
- \*
- \* 极端输入场景:
  - \* 1. 数组全为 0
  - \* 2. 数组全为正数或全为负数
  - \* 3. 数组长度小于 3
  - \* 4. 数组包含大量重复元素
- \*
- \* 与机器学习等领域的联系:
  - \* 1. 在特征选择中, 可能需要找到三个特征的组合满足特定条件
  - \* 2. 在推荐系统中, 可能需要找到三个物品的组合满足用户偏好

```
public class Code09_ThreeSum {

 /**
 * 查找所有和为 0 的不重复三元组
 *
 * @param nums 整数数组
 * @return 所有和为 0 的不重复三元组列表
 */

 public static List<List<Integer>> threeSum(int[] nums) {
 // 结果列表
 List<List<Integer>> result = new ArrayList<>();

 // 边界条件检查
 if (nums == null || nums.length < 3) {
 return result;
 }

 // 对数组进行排序, 时间复杂度 O(n log n)
 Arrays.sort(nums);

 // 遍历数组, 固定第一个数
```

```
for (int i = 0; i < nums.length - 2; i++) {
 // 如果当前数字大于 0，则三数之和一定大于 0，结束循环
 if (nums[i] > 0) {
 break;
 }

 // 跳过重复元素，避免出现重复解
 if (i > 0 && nums[i] == nums[i - 1]) {
 continue;
 }

 // 使用双指针在剩余数组中寻找另外两个数
 int left = i + 1;
 int right = nums.length - 1;

 while (left < right) {
 int sum = nums[i] + nums[left] + nums[right];

 if (sum == 0) {
 // 找到一个解
 result.add(Arrays.asList(nums[i], nums[left], nums[right]));
 }

 // 跳过重复元素
 while (left < right && nums[left] == nums[left + 1]) {
 left++;
 }

 while (left < right && nums[right] == nums[right - 1]) {
 right--;
 }

 // 移动指针继续寻找
 left++;
 right--;
 } else if (sum < 0) {
 // 和小于 0，左指针右移
 left++;
 } else {
 // 和大于 0，右指针左移
 right--;
 }
}
```

```

 return result;
 }

/**
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: [-1, 0, 1, 2, -1, -4] -> [[-1, -1, 2], [-1, 0, 1]]
 int[] nums1 = {-1, 0, 1, 2, -1, -4};
 List<List<Integer>> result1 = threeSum(nums1);
 System.out.println("Test 1: nums=" + Arrays.toString(nums1));
 System.out.println("Result: " + result1);

 // 测试用例 2: [0, 1, 1] -> []
 int[] nums2 = {0, 1, 1};
 List<List<Integer>> result2 = threeSum(nums2);
 System.out.println("Test 2: nums=" + Arrays.toString(nums2));
 System.out.println("Result: " + result2);

 // 测试用例 3: [0, 0, 0] -> [[0, 0, 0]]
 int[] nums3 = {0, 0, 0};
 List<List<Integer>> result3 = threeSum(nums3);
 System.out.println("Test 3: nums=" + Arrays.toString(nums3));
 System.out.println("Result: " + result3);
}

```

=====

文件: Code09\_ThreeSum.py

=====

"""

三数之和

题目描述:

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足 `i != j`、`i != k` 且 `j != k`，

还要满足 `nums[i] + nums[j] + nums[k] == 0`。请你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

输入：`nums = [-1, 0, 1, 2, -1, -4]`

输出：`[[[-1, -1, 2], [-1, 0, 1]]]`

解释：

`nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0`。

`nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0`。

`nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0`。

不同的三元组是 `[-1, 0, 1]` 和 `[-1, -1, 2]`。

注意，输出的顺序和三元组的顺序并不重要。

输入： `nums = [0, 1, 1]`

输出： `[]`

解释： 唯一可能的三元组和不为 0。

输入： `nums = [0, 0, 0]`

输出： `[[0, 0, 0]]`

解释： 唯一可能的三元组和为 0。

解题思路：

1. 首先对数组进行排序，这样可以方便使用双指针法，并且便于去重。
2. 遍历数组，固定第一个数 `nums[i]`，然后在剩余的数组中使用双指针法寻找两个数，使得这三个数的和为 0。
3. 使用双指针法：左指针指向 `i+1`，右指针指向数组末尾。
  - 如果三数之和等于 0，则找到一个解，同时移动左右指针。
  - 如果三数之和小于 0，则左指针右移。
  - 如果三数之和大于 0，则右指针左移。
4. 注意去重处理：
  - 固定的第一个数如果相同则跳过
  - 找到解后，左右指针需要跳过相同的数

时间复杂度：  $O(n^2)$  – 外层循环  $O(n)$ ，内层双指针  $O(n)$

空间复杂度：  $O(1)$  – 不考虑返回结果的空间

是否最优解： 是 – 基于比较的算法下界为  $O(n^2)$ ，本算法已达到最优

相关题目：

1. LeetCode 1 – 两数之和（无序数组，使用哈希表）
2. LeetCode 167 – 两数之和 II – 输入有序数组（排序+双指针）
3. LeetCode 15 – 三数之和（当前题目）
4. LeetCode 18 – 四数之和（排序+双指针）
5. LeetCode 16 – 最接近的三数之和

工程化考虑：

1. 输入验证： 检查数组是否为空或长度小于 3
2. 异常处理： 处理各种边界情况
3. 边界条件： 处理全为 0、全为正数、全为负数的情况

语言特性差异：

Java：使用 ArrayList 存储结果，需要导入相关包

C++：可使用 vector 存储结果

Python：可使用列表存储结果

极端输入场景：

1. 数组全为 0
2. 数组全为正数或全为负数
3. 数组长度小于 3
4. 数组包含大量重复元素

与机器学习等领域的联系：

1. 在特征选择中，可能需要找到三个特征的组合满足特定条件
2. 在推荐系统中，可能需要找到三个物品的组合满足用户偏好

"""

```
def three_sum(nums):
```

```
 """
```

```
 查找所有和为 0 的不重复三元组
```

Args:

```
 nums: 整数数组
```

Returns:

```
 所有和为 0 的不重复三元组列表
```

```
 """
```

```
 # 结果列表
```

```
 result = []
```

```
 # 边界条件检查
```

```
 if not nums or len(nums) < 3:
 return result
```

```
 # 对数组进行排序，时间复杂度 O(n log n)
```

```
 nums.sort()
```

```
 # 遍历数组，固定第一个数
```

```
 for i in range(len(nums) - 2):
 # 如果当前数字大于 0，则三数之和一定大于 0，结束循环
 if nums[i] > 0:
 break
```

```

跳过重复元素，避免出现重复解
if i > 0 and nums[i] == nums[i - 1]:
 continue

使用双指针在剩余数组中寻找另外两个数
left = i + 1
right = len(nums) - 1

while left < right:
 sum_val = nums[i] + nums[left] + nums[right]

 if sum_val == 0:
 # 找到一个解
 result.append([nums[i], nums[left], nums[right]])

 # 跳过重复元素
 while left < right and nums[left] == nums[left + 1]:
 left += 1
 while left < right and nums[right] == nums[right - 1]:
 right -= 1

 # 移动指针继续寻找
 left += 1
 right -= 1

elif sum_val < 0:
 # 和小于 0，左指针右移
 left += 1
else:
 # 和大于 0，右指针左移
 right -= 1

return result

```

```

def test_three_sum():
 """测试函数"""
 # 测试用例 1: [-1, 0, 1, 2, -1, -4] -> [[-1, -1, 2], [-1, 0, 1]]
 nums1 = [-1, 0, 1, 2, -1, -4]
 result1 = three_sum(nums1)
 print(f"Test 1: nums={nums1}")
 print(f"Result: {result1}")

 # 测试用例 2: [0, 1, 1] -> []

```

```
nums2 = [0, 1, 1]
result2 = three_sum(nums2)
print(f"Test 2: nums={nums2}")
print(f"Result: {result2}")

测试用例 3: [0, 0, 0] -> [[0, 0, 0]]
nums3 = [0, 0, 0]
result3 = three_sum(nums3)
print(f"Test 3: nums={nums3}")
print(f"Result: {result3}")
```

## # 主函数

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

=====

文件: Code10\_ContainerWithMostWater.cpp

=====

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

/***
 * 盛最多水的容器
 *
 * 题目描述:
 * 给定一个长度为 n 的整数数组 height。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i])。
 * 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。
 * 返回容器可以储存的最大水量。
 * 说明：你不能倾斜容器。
 *
```

```
* 示例：
 * 输入：[1, 8, 6, 2, 5, 4, 8, 3, 7]
 * 输出：49
 * 解释：图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水的最大值为 49。
 *
 * 输入：height = [1, 1]
 * 输出：1
 *
```

- \* 解题思路:
  - \* 使用双指针法。左指针指向数组开始，右指针指向数组末尾。
  - \* 容器的容量由较短的那条线决定，所以每次移动较短的那条线的指针，尝试寻找更长的线来增大容量。
  - \* 这样可以在  $O(n)$  时间内找到最大容量。
- \*
- \* 时间复杂度:  $O(n)$  - 双指针最多遍历一次数组
- \* 空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间
- \* 是否最优解: 是 - 基于比较的算法下界为  $O(n)$ ，本算法已达到最优
- \*
- \* 相关题目:
  - \* 1. LeetCode 11 - 盛最多水的容器（当前题目）
  - \* 2. LeetCode 42 - 接雨水（双指针）
  - \* 3. LeetCode 84 - 柱状图中最大的矩形（单调栈）
- \*
- \* 工程化考虑:
  - \* 1. 输入验证: 检查数组是否为空或长度小于 2
  - \* 2. 异常处理: 处理各种边界情况
  - \* 3. 边界条件: 处理数组长度为 2 的情况
- \*
- \* 语言特性差异:
  - \* Java: 使用 `Math.min` 和 `Math.max` 函数
  - \* C++: 可使用 `std::min` 和 `std::max` 函数
  - \* Python: 可使用 `min` 和 `max` 函数
- \*
- \* 极端输入场景:
  - \* 1. 数组长度为 2
  - \* 2. 数组中所有元素相等
  - \* 3. 数组呈递增或递减趋势
  - \* 4. 最大值在数组中间
- \*
- \* 与机器学习等领域的联系:
  - \* 1. 在优化问题中，可能需要找到两个参数的最优组合
  - \* 2. 在特征工程中，可能需要找到两个特征的最佳配对

```
class Solution {
public:
 /**
 * 计算盛最多水的容器的容量
 *
 * @param height 整数数组，表示每条垂线的高度
 * @return 容器可以储存的最大水量
 */
 int maxArea(vector<int>& height) {
```

```
// 边界条件检查
if (height.size() < 2) {
 return 0;
}

int maxWater = 0;
int left = 0; // 左指针
int right = height.size() - 1; // 右指针

// 当左指针小于右指针时继续循环
while (left < right) {
 // 计算当前容器的容量
 // 容量由较短的那条线决定，乘以两条线之间的距离
 int currentWater = min(height[left], height[right]) * (right - left);

 // 更新最大容量
 maxWater = max(maxWater, currentWater);

 // 移动较短的那条线的指针，尝试寻找更长的线来增大容量
 if (height[left] <= height[right]) {
 left++;
 } else {
 right--;
 }
}

return maxWater;
}

};

/***
 * 测试函数
 */
/*
void testMaxArea() {
 Solution solution;

 // 测试用例 1: [1, 8, 6, 2, 5, 4, 8, 3, 7] -> 49
 vector<int> height1 = {1, 8, 6, 2, 5, 4, 8, 3, 7};
 int result1 = solution.maxArea(height1);
 cout << "Test 1: height=[";
 for (size_t i = 0; i < height1.size(); ++i) {
 cout << height1[i];
 }
}
```

```

 if (i < height1.size() - 1) cout << ",";
 }

 cout << "]\\nResult: " << result1 << "\\n\\n";

 // 测试用例 2: [1, 1] -> 1
 vector<int> height2 = {1, 1};
 int result2 = solution.maxArea(height2);
 cout << "Test 2: height=[";
 for (size_t i = 0; i < height2.size(); ++i) {
 cout << height2[i];
 if (i < height2.size() - 1) cout << ",";
 }
 cout << "]\\nResult: " << result2 << "\\n";
}

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

```

文件: Code10\_ContainerWithMostWater.java

```

=====
package class050;

/**
 * 盛最多水的容器
 *
 * 题目描述:
 * 给定一个长度为 n 的整数数组 height。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i])。
 * 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。
 * 返回容器可以储存的最大水量。
 * 说明：你不能倾斜容器。
 *
 * 示例：
 * 输入：[1, 8, 6, 2, 5, 4, 8, 3, 7]
 * 输出：49
 * 解释：图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水的最大值为 49。
 *
 * 输入：height = [1, 1]

```

- \* 输出: 1
- \*
- \* 解题思路:
  - \* 使用双指针法。左指针指向数组开始，右指针指向数组末尾。
  - \* 容器的容量由较短的那条线决定，所以每次移动较短的那条线的指针，尝试寻找更长的线来增大容量。
  - \* 这样可以在  $O(n)$  时间内找到最大容量。
- \*
- \* 时间复杂度:  $O(n)$  - 双指针最多遍历一次数组
- \* 空间复杂度:  $O(1)$  - 只使用了常数级别的额外空间
- \* 是否最优解: 是 - 基于比较的算法下界为  $O(n)$ ，本算法已达到最优
- \*
- \* 相关题目:
  - \* 1. LeetCode 11 - 盛最多水的容器（当前题目）
  - \* 2. LeetCode 42 - 接雨水（双指针）
  - \* 3. LeetCode 84 - 柱状图中最大的矩形（单调栈）
- \*
- \* 工程化考虑:
  - \* 1. 输入验证: 检查数组是否为空或长度小于 2
  - \* 2. 异常处理: 处理各种边界情况
  - \* 3. 边界条件: 处理数组长度为 2 的情况
- \*
- \* 语言特性差异:
  - \* Java: 使用 `Math.min` 和 `Math.max` 函数
  - \* C++: 可使用 `std::min` 和 `std::max` 函数
  - \* Python: 可使用 `min` 和 `max` 函数
- \*
- \* 极端输入场景:
  - \* 1. 数组长度为 2
  - \* 2. 数组中所有元素相等
  - \* 3. 数组呈递增或递减趋势
  - \* 4. 最大值在数组中间
- \*
- \* 与机器学习等领域的联系:
  - \* 1. 在优化问题中，可能需要找到两个参数的最优组合
  - \* 2. 在特征工程中，可能需要找到两个特征的最佳配对

```
public class Code10_ContainerWithMostWater {
```

```
 /**
 * 计算盛最多水的容器的容量
 *
 * @param height 整数数组，表示每条垂线的高度
 * @return 容器可以储存的最大水量
```

```
/*
public static int maxArea(int[] height) {
 // 边界条件检查
 if (height == null || height.length < 2) {
 return 0;
 }

 int maxWater = 0;
 int left = 0; // 左指针
 int right = height.length - 1; // 右指针

 // 当左指针小于右指针时继续循环
 while (left < right) {
 // 计算当前容器的容量
 // 容量由较短的那条线决定，乘以两条线之间的距离
 int currentWater = Math.min(height[left], height[right]) * (right - left);

 // 更新最大容量
 maxWater = Math.max(maxWater, currentWater);

 // 移动较短的那条线的指针，尝试寻找更长的线来增大容量
 if (height[left] <= height[right]) {
 left++;
 } else {
 right--;
 }
 }

 return maxWater;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: [1, 8, 6, 2, 5, 4, 8, 3, 7] -> 49
 int[] height1 = {1, 8, 6, 2, 5, 4, 8, 3, 7};
 int result1 = maxArea(height1);
 System.out.println("Test 1: height=" + java.util.Arrays.toString(height1));
 System.out.println("Result: " + result1);

 // 测试用例 2: [1, 1] -> 1
 int[] height2 = {1, 1};
}
```

```
 int result2 = maxArea(height2);
 System.out.println("Test 2: height=" + java.util.Arrays.toString(height2));
 System.out.println("Result: " + result2);
}
=====
```

文件: Code10\_ContainerWithMostWater.py

```
=====
```

```
"""
盛最多水的容器
```

题目描述:

给定一个长度为  $n$  的整数数组  $height$ 。有  $n$  条垂线，第  $i$  条线的两个端点是  $(i, 0)$  和  $(i, height[i])$ 。

找出其中的两条线，使得它们与  $x$  轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例：

输入： [1, 8, 6, 2, 5, 4, 8, 3, 7]

输出： 49

解释：图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水的最大值为 49。

输入： height = [1, 1]

输出： 1

解题思路：

使用双指针法。左指针指向数组开始，右指针指向数组末尾。

容器的容量由较短的那条线决定，所以每次移动较短的那条线的指针，尝试寻找更长的线来增大容量。  
这样可以在  $O(n)$  时间内找到最大容量。

时间复杂度：  $O(n)$  – 双指针最多遍历一次数组

空间复杂度：  $O(1)$  – 只使用了常数级别的额外空间

是否最优解： 是 – 基于比较的算法下界为  $O(n)$ ，本算法已达到最优

相关题目：

1. LeetCode 11 – 盛最多水的容器（当前题目）
2. LeetCode 42 – 接雨水（双指针）
3. LeetCode 84 – 柱状图中最大的矩形（单调栈）

工程化考虑：

1. 输入验证：检查数组是否为空或长度小于 2
2. 异常处理：处理各种边界情况
3. 边界条件：处理数组长度为 2 的情况

语言特性差异：

Java：使用 Math.min 和 Math.max 函数

C++：可使用 std::min 和 std::max 函数

Python：可使用 min 和 max 函数

极端输入场景：

1. 数组长度为 2
2. 数组中所有元素相等
3. 数组呈递增或递减趋势
4. 最大值在数组中间

与机器学习等领域的联系：

1. 在优化问题中，可能需要找到两个参数的最优组合
2. 在特征工程中，可能需要找到两个特征的最佳配对

"""

```
def max_area(height):
 """
```

计算盛最多水的容器的容量

Args:

height: 整数数组，表示每条垂线的高度

Returns:

容器可以储存的最大水量

"""

# 边界条件检查

```
if not height or len(height) < 2:
 return 0
```

max\_water = 0

left = 0 # 左指针

right = len(height) - 1 # 右指针

# 当左指针小于右指针时继续循环

```
while left < right:
```

# 计算当前容器的容量

# 容量由较短的那条线决定，乘以两条线之间的距离

```

current_water = min(height[left], height[right]) * (right - left)

更新最大容量
max_water = max(max_water, current_water)

移动较短的那条线的指针，尝试寻找更长的线来增大容量
if height[left] <= height[right]:
 left += 1
else:
 right -= 1

return max_water

def test_max_area():
 """测试函数"""
 # 测试用例 1: [1, 8, 6, 2, 5, 4, 8, 3, 7] -> 49
 height1 = [1, 8, 6, 2, 5, 4, 8, 3, 7]
 result1 = max_area(height1)
 print(f"Test 1: height={height1}")
 print(f"Result: {result1}")

 # 测试用例 2: [1, 1] -> 1
 height2 = [1, 1]
 result2 = max_area(height2)
 print(f"Test 2: height={height2}")
 print(f"Result: {result2}")

主函数
if __name__ == "__main__":
 test_max_area()

```

=====

文件: Code11\_TwoSumII.cpp

=====

```

/**
 * LeetCode 167. 两数之和 II - 输入有序数组 (Two Sum II - Input Array Is Sorted)
 *
 * 题目描述:
 * 给你一个下标从 1 开始的整数数组 numbers，该数组已按非递减顺序排列，
 * 请你从数组中找出满足相加之和等于目标数 target 的两个数。

```

- \* 如果设这两个数分别是 numbers[index1] 和 numbers[index2] ,
- \* 则  $1 \leq index1 < index2 \leq numbers.length$  。
- \* 以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1 和 index2。
- \*
- \* 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。
- \* 你所设计的解决方案必须只使用常量级的额外空间。
- \*
- \* 示例 1:
- \* 输入: numbers = [2, 7, 11, 15], target = 9
- \* 输出: [1, 2]
- \* 解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。返回 [1, 2] 。
- \*
- \* 示例 2:
- \* 输入: numbers = [2, 3, 4], target = 6
- \* 输出: [1, 3]
- \* 解释: 2 与 4 之和等于目标数 6 。因此 index1 = 1, index2 = 3 。返回 [1, 3] 。
- \*
- \* 示例 3:
- \* 输入: numbers = [-1, 0], target = -1
- \* 输出: [1, 2]
- \* 解释: -1 与 0 之和等于目标数 -1 。因此 index1 = 1, index2 = 2 。返回 [1, 2] 。
- \*
- \* 提示:
- \*  $2 \leq numbers.length \leq 3 * 10^4$
- \*  $-1000 \leq numbers[i] \leq 1000$
- \* numbers 按 非递减顺序 排列
- \*  $-1000 \leq target \leq 1000$
- \* 仅存在一个有效答案
- \*
- \* 题目链接: <https://leetcode.cn/problems/two-sum-ii-input-array-is-sorted/>
- \*
- \* 解题思路:
- \* 这道题可以使用多种方法解决，从暴力到高效:
- \*
- \* 方法一 (暴力解法):
- \* 使用两层嵌套循环遍历所有可能的数对，找到和为 target 的数对。
- \* 时间复杂度:  $O(n^2)$ ，空间复杂度:  $O(1)$
- \*
- \* 方法二 (二分查找):
- \* 对于每个元素，使用二分查找寻找另一个元素使得它们的和为 target。
- \* 时间复杂度:  $O(n \log n)$ ，空间复杂度:  $O(1)$
- \*
- \* 方法三 (双指针):

- \* 1. 初始化左指针 left 指向数组起始位置，右指针 right 指向数组末尾位置
- \* 2. 计算当前两数之和: sum = numbers[left] + numbers[right]
- \* 3. 如果 sum == target, 返回结果 [left+1, right+1] (注意题目要求索引从 1 开始)
- \* 4. 如果 sum < target, 说明需要增大和, 移动左指针 left++
- \* 5. 如果 sum > target, 说明需要减小和, 移动右指针 right--
- \* 6. 重复步骤 2-5, 直到找到答案 (题目保证有唯一解)
- \* 时间复杂度: O(n), 空间复杂度: O(1)
- \*
- \* 方法四 (优化的双指针):
- \* 当数组中有大量重复元素时, 可以通过跳过相同的元素来提高效率。
- \* 时间复杂度: O(n), 空间复杂度: O(1)
- \*
- \* 最优解是方法三和方法四, 时间复杂度 O(n), 空间复杂度 O(1)。

\*/

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <stdexcept>
#include <climits> // 用于 INT_MAX 和 INT_MIN
using namespace std;
class Solution {
public:
 /**
 * 解法一：暴力解法（不推荐，会超时）
 *
 * @param numbers 输入的有序数组
 * @param target 目标和
 * @return 两个数的索引数组（索引从 1 开始）
 *
 * 时间复杂度: O(n2) - 需要两层循环遍历所有可能的数对
 * 空间复杂度: O(1) - 只使用了常量级的额外空间
 */
 vector<int> twoSumBruteForce(const vector<int>& numbers, int target) {
 // 参数校验
 if (numbers.size() < 2) {
 throw invalid_argument("输入数组长度必须至少为 2");
 }

 int n = numbers.size();
 // 两层循环遍历所有可能的数对
 }
}
```

```

 for (int i = 0; i < n - 1; ++i) {
 for (int j = i + 1; j < n; ++j) {
 if (numbers[i] + numbers[j] == target) {
 return {i + 1, j + 1}; // 注意索引从 1 开始
 }
 }
 }

 // 根据题目描述，一定存在解，所以不会执行到这里
 throw invalid_argument("未找到符合条件的数对");
 }

 /**
 * 解法二：二分查找
 *
 * @param numbers 输入的有序数组
 * @param target 目标和
 * @return 两个数的索引数组（索引从 1 开始）
 *
 * 时间复杂度：O(n log n) - 对每个元素执行二分查找，每个二分查找是 O(log n)
 * 空间复杂度：O(1) - 只使用了常量级的额外空间
 */
}

vector<int> twoSumBinarySearch(const vector<int>& numbers, int target) {
 // 参数校验
 if (numbers.size() < 2) {
 throw invalid_argument("输入数组长度必须至少为 2");
 }

 int n = numbers.size();

 // 遍历数组，对每个元素使用二分查找寻找另一个元素
 for (int i = 0; i < n; ++i) {
 int complement = target - numbers[i];
 int left = i + 1;
 int right = n - 1;

 while (left <= right) {
 int mid = left + (right - left) / 2; // 避免整数溢出
 if (numbers[mid] == complement) {
 return {i + 1, mid + 1}; // 注意索引从 1 开始
 } else if (numbers[mid] < complement) {
 left = mid + 1;
 } else {

```

```

 right = mid - 1;
 }
}
}

// 根据题目描述，一定存在解，所以不会执行到这里
throw invalid_argument("未找到符合条件的数对");
}

/**
 * 解法三：双指针（最优解）
 *
 * @param numbers 输入的有序数组
 * @param target 目标和
 * @return 两个数的索引数组（索引从 1 开始）
 *
 * 时间复杂度：O(n) - 只需要一次遍历数组
 * 空间复杂度：O(1) - 只使用了常量级的额外空间
 */
vector<int> twoSum(const vector<int>& numbers, int target) {
 // 参数校验
 if (numbers.size() < 2) {
 throw invalid_argument("输入数组长度必须至少为 2");
 }

 // 初始化左右指针
 int left = 0;
 int right = numbers.size() - 1;

 while (left < right) {
 int sum = numbers[left] + numbers[right];

 if (sum == target) {
 return {left + 1, right + 1}; // 注意索引从 1 开始
 } else if (sum < target) {
 // 和小于目标值，需要增大和，移动左指针
 ++left;
 } else {
 // 和大于目标值，需要减小和，移动右指针
 --right;
 }
 }
}
```

```

// 根据题目描述，一定存在解，所以不会执行到这里
throw invalid_argument("未找到符合条件的数对");
}

/**
 * 解法四：优化的双指针实现
 * 当数组中有大量重复元素时，可以通过跳过相同的元素来提高效率
 *
 * @param numbers 输入的有序数组
 * @param target 目标和
 * @return 两个数的索引数组（索引从 1 开始）
 *
 * 时间复杂度: O(n) - 最坏情况下需要一次遍历数组
 * 空间复杂度: O(1) - 只使用了常量级的额外空间
 */
vector<int> twoSumOptimized(const vector<int>& numbers, int target) {
 // 参数校验
 if (numbers.size() < 2) {
 throw invalid_argument("输入数组长度必须至少为 2");
 }

 // 初始化左右指针
 int left = 0;
 int right = numbers.size() - 1;

 while (left < right) {
 int sum = numbers[left] + numbers[right];

 if (sum == target) {
 return {left + 1, right + 1}; // 注意索引从 1 开始
 } else if (sum < target) {
 // 和小于目标值，需要增大和，移动左指针
 ++left;
 // 跳过重复的元素
 while (left < right && numbers[left] == numbers[left - 1]) {
 ++left;
 }
 } else {
 // 和大于目标值，需要减小和，移动右指针
 --right;
 // 跳过重复的元素
 while (left < right && numbers[right] == numbers[right + 1]) {
 --right;
 }
 }
 }
}

```

```
 }
 }
}

// 根据题目描述，一定存在解，所以不会执行到这里
throw invalid_argument("未找到符合条件的数对");
}

};

/***
 * 打印向量辅助函数
 */
void printVector(const vector<int>& vec, const string& prefix = "") {
 if (!prefix.empty()) {
 cout << prefix;
 }
 cout << "[";
 for (size_t i = 0; i < vec.size(); ++i) {
 cout << vec[i];
 if (i < vec.size() - 1) {
 cout << ", ";
 }
 }
 cout << "]" << endl;
}

/***
 * 测试函数
 */
void test() {
 Solution solution;

 // 测试用例 1
 vector<int> numbers1 = {2, 7, 11, 15};
 int target1 = 9;
 vector<int> expected1 = {1, 2};
 vector<int> result1 = solution.twoSum(numbers1, target1);
 cout << "测试用例 1:" << endl;
 printVector(numbers1, "输入数组: ");
 cout << "目标值: " << target1 << endl;
 printVector(result1, "结果: ");
 printVector(expected1, "期望: ");
 cout << "测试通过: " << (result1 == expected1) << endl << endl;
}
```

```
// 测试用例 2
vector<int> numbers2 = {2, 3, 4};
int target2 = 6;
vector<int> expected2 = {1, 3};
vector<int> result2 = solution.twoSum(numbers2, target2);
cout << "测试用例 2:" << endl;
printVector(numbers2, "输入数组:");
cout << "目标值: " << target2 << endl;
printVector(result2, "结果:");
printVector(expected2, "期望:");
cout << "测试通过: " << (result2 == expected2) << endl << endl;
```

```
// 测试用例 3
vector<int> numbers3 = {-1, 0};
int target3 = -1;
vector<int> expected3 = {1, 2};
vector<int> result3 = solution.twoSum(numbers3, target3);
cout << "测试用例 3:" << endl;
printVector(numbers3, "输入数组:");
cout << "目标值: " << target3 << endl;
printVector(result3, "结果:");
printVector(expected3, "期望:");
cout << "测试通过: " << (result3 == expected3) << endl << endl;
```

```
// 测试用例 4 - 边界情况: 有重复元素
vector<int> numbers4 = {1, 1, 2, 4, 5};
int target4 = 2;
vector<int> expected4 = {1, 2};
vector<int> result4 = solution.twoSum(numbers4, target4);
cout << "测试用例 4 (有重复元素) :" << endl;
printVector(numbers4, "输入数组:");
cout << "目标值: " << target4 << endl;
printVector(result4, "结果:");
printVector(expected4, "期望:");
cout << "测试通过: " << (result4 == expected4) << endl << endl;
```

```
// 测试用例 5 - 边界情况: 数组长度为 2
vector<int> numbers5 = {1, 3};
int target5 = 4;
vector<int> expected5 = {1, 2};
vector<int> result5 = solution.twoSum(numbers5, target5);
cout << "测试用例 5 (数组长度为 2) :" << endl;
```

```
printVector(numbers5, "输入数组：");
cout << "目标值：" << target5 << endl;
printVector(result5, "结果：");
printVector(expected5, "期望：");
cout << "测试通过：" << (result5 == expected5) << endl << endl;
}

/***
 * 性能测试
 */
void performanceTest() {
 Solution solution;

 // 创建一个较大的测试数组
 const int size = 10000;
 vector<int> largeArray(size);
 for (int i = 0; i < size; ++i) {
 largeArray[i] = i * 2; // 生成偶数序列
 }
 int target = largeArray[size/2] + largeArray[size/2 + 1]; // 确保有解

 // 测试解法二的性能
 auto start = chrono::high_resolution_clock::now();
 vector<int> result2 = solution.twoSumBinarySearch(largeArray, target);
 auto end = chrono::high_resolution_clock::now();
 auto duration2 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法二（二分查找）耗时：" << duration2 << "ms" << endl;

 // 测试解法三的性能
 start = chrono::high_resolution_clock::now();
 vector<int> result3 = solution.twoSum(largeArray, target);
 end = chrono::high_resolution_clock::now();
 auto duration3 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法三（双指针）耗时：" << duration3 << "ms" << endl;

 // 测试解法四的性能
 start = chrono::high_resolution_clock::now();
 vector<int> result4 = solution.twoSumOptimized(largeArray, target);
 end = chrono::high_resolution_clock::now();
 auto duration4 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法四（优化双指针）耗时：" << duration4 << "ms" << endl;

 // 验证结果是否一致
}
```

```

cout << "解法二和解法三结果一致: " << (result2 == result3) << endl;
cout << "解法三和解法四结果一致: " << (result3 == result4) << endl;

// 测试有重复元素的性能
vector<int> duplicateArray(size);
for (int i = 0; i < size; ++i) {
 duplicateArray[i] = i % 100; // 生成大量重复元素
}
target = 100; // 确保有解

start = chrono::high_resolution_clock::now();
result3 = solution.twoSum(duplicateArray, target);
end = chrono::high_resolution_clock::now();
duration3 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
cout << "\n重复元素测试 - 双指针耗时: " << duration3 << "ms" << endl;

start = chrono::high_resolution_clock::now();
result4 = solution.twoSumOptimized(duplicateArray, target);
end = chrono::high_resolution_clock::now();
duration4 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
cout << "重复元素测试 - 优化双指针耗时: " << duration4 << "ms" << endl;
cout << "在重复元素情况下, 优化效果: " << (duration3 > duration4 ? "有效" : "不明显") <<
endl;
}

/***
 * 边界条件测试
 */
void boundaryTest() {
 Solution solution;

 try {
 // 测试空输入
 vector<int> empty;
 solution.twoSum(empty, 5);
 cout << "边界测试失败: 空输入没有抛出异常" << endl;
 } catch (const invalid_argument& e) {
 cout << "边界测试通过: 空输入正确抛出异常: " << e.what() << endl;
 }

 try {
 // 测试长度为 1 的输入
 vector<int> single = {1};

```

```

 solution.twoSum(single, 1);
 cout << "边界测试失败: 长度为 1 的输入没有抛出异常" << endl;
 } catch (const invalid_argument& e) {
 cout << "边界测试通过: 长度为 1 的输入正确抛出异常: " << e.what() << endl;
 }

// 测试最大可能值
vector<int> maxArray = {INT_MAX - 1, INT_MAX};
int maxTarget = (INT_MAX - 1) + (INT_MAX - 1); // 避免溢出
try {
 vector<int> result = solution.twoSum(maxArray, maxTarget);
 printVector(result, "测试最大值: ");
} catch (const exception& e) {
 cout << "测试最大值时发生错误: " << e.what() << endl;
}

// 测试最小可能值
vector<int> minArray = {INT_MIN, INT_MIN + 1};
int minTarget = INT_MIN + (INT_MIN + 1);
try {
 vector<int> result = solution.twoSum(minArray, minTarget);
 printVector(result, "测试最小值: ");
} catch (const exception& e) {
 cout << "测试最小值时发生错误: " << e.what() << endl;
}

}

/***
 * 调试辅助函数 - 打印中间状态
 */
void debugTwoSum(const vector<int>& numbers, int target) {
 cout << "\n 调试模式: " << endl;
 printVector(numbers, "输入数组: ");
 cout << "目标值: " << target << endl;

 int left = 0;
 int right = numbers.size() - 1;
 int step = 1;

 while (left < right) {
 int sum = numbers[left] + numbers[right];
 cout << "步骤 " << step << ": left=" << left << " (" << numbers[left] << "), "
 << "right=" << right << " (" << numbers[right] << "), "
 }
}

```

```

 << "sum=" << sum << endl;

 if (sum == target) {
 cout << "找到解: [" << (left + 1) << ", " << (right + 1) << "]"
 << endl;
 break;
 } else if (sum < target) {
 cout << "sum < target, 移动左指针" << endl;
 ++left;
 } else {
 cout << "sum > target, 移动右指针" << endl;
 --right;
 }
 ++step;
}

int main() {
 cout << "==== 测试用例 ===" << endl;
 test();

 cout << "==== 性能测试 ===" << endl;
 performanceTest();

 cout << "==== 边界条件测试 ===" << endl;
 boundaryTest();

 cout << "\n==== 调试演示 ===" << endl;
 debugTwoSum({2, 7, 11, 15}, 9);

 return 0;
}

```

=====

文件: Code11\_TwoSumII.java

```

=====
package class050;

import java.util.Arrays;

/**
 * LeetCode 167. 两数之和 II - 输入有序数组 (Two Sum II - Input Array Is Sorted)
 *

```

- \* 题目描述:
- \* 给你一个下标从 1 开始的整数数组 numbers，该数组已按非递减顺序排列，
- \* 请你从数组中找出满足相加之和等于目标数 target 的两个数。
- \* 如果设这两个数分别是 numbers[index1] 和 numbers[index2]，
- \* 则  $1 \leq index1 < index2 \leq numbers.length$ 。
- \* 以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1 和 index2。
- \*
- \* 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。
- \* 你所设计的解决方案必须只使用常量级的额外空间。
- \*
- \* 示例 1:
- \* 输入: numbers = [2, 7, 11, 15], target = 9
- \* 输出: [1, 2]
- \* 解释: 2 与 7 之和等于目标数 9。因此 index1 = 1, index2 = 2。返回 [1, 2]。
- \*
- \* 示例 2:
- \* 输入: numbers = [2, 3, 4], target = 6
- \* 输出: [1, 3]
- \* 解释: 2 与 4 之和等于目标数 6。因此 index1 = 1, index2 = 3。返回 [1, 3]。
- \*
- \* 示例 3:
- \* 输入: numbers = [-1, 0], target = -1
- \* 输出: [1, 2]
- \* 解释: -1 与 0 之和等于目标数 -1。因此 index1 = 1, index2 = 2。返回 [1, 2]。
- \*
- \* 提示:
- \*  $2 \leq numbers.length \leq 3 * 10^4$
- \*  $-1000 \leq numbers[i] \leq 1000$
- \* numbers 按 非递减顺序 排列
- \*  $-1000 \leq target \leq 1000$
- \* 仅存在一个有效答案
- \*
- \* 题目链接: <https://leetcode.cn/problems/two-sum-ii-input-array-is-sorted/>
- \*
- \* 解题思路:
- \* 这道题可以使用双指针技巧高效解决:
- \*
- \* 方法一（暴力解法）:
- \* 使用两层嵌套循环遍历所有可能的数对，找到和为 target 的数对。
- \* 时间复杂度:  $O(n^2)$ ，空间复杂度:  $O(1)$
- \*
- \* 方法二（二分查找）:
- \* 对于每个元素，使用二分查找寻找另一个元素使得它们的和为 target。

```

* 时间复杂度: O(n log n), 空间复杂度: O(1)
*
* 方法三 (双指针):
* 1. 初始化左指针 left 指向数组起始位置, 右指针 right 指向数组末尾位置
* 2. 计算当前两数之和: sum = numbers[left] + numbers[right]
* 3. 如果 sum == target, 返回结果 [left+1, right+1] (注意题目要求索引从 1 开始)
* 4. 如果 sum < target, 说明需要增大和, 移动左指针 left++
* 5. 如果 sum > target, 说明需要减小和, 移动右指针 right--
* 6. 重复步骤 2-5, 直到找到答案 (题目保证有唯一解)
* 时间复杂度: O(n), 空间复杂度: O(1)
*
* 最优解是方法三, 时间复杂度 O(n), 空间复杂度 O(1)。
*/

```

```

class Solution {
 /**
 * 解法一: 暴力解法 (不推荐, 会超时)
 *
 * @param numbers 输入的有序数组
 * @param target 目标和
 * @return 两个数的索引数组 (索引从 1 开始)
 *
 * 时间复杂度: O(n2) - 需要两层循环遍历所有可能的数对
 * 空间复杂度: O(1) - 只使用了常量级的额外空间
 */
 public int[] twoSumBruteForce(int[] numbers, int target) {
 // 参数校验
 if (numbers == null || numbers.length < 2) {
 throw new IllegalArgumentException("输入数组不能为 null 且长度必须至少为 2");
 }

 int n = numbers.length;

 // 两层循环遍历所有可能的数对
 for (int i = 0; i < n - 1; i++) {
 for (int j = i + 1; j < n; j++) {
 if (numbers[i] + numbers[j] == target) {
 return new int[]{i + 1, j + 1}; // 注意索引从 1 开始
 }
 }
 }

 // 根据题目描述, 一定存在解, 所以不会执行到这里
 }
}

```

```
 throw new IllegalArgumentException("未找到符合条件的数对");
 }

/**
 * 解法二：二分查找
 *
 * @param numbers 输入的有序数组
 * @param target 目标和
 * @return 两个数的索引数组（索引从 1 开始）
 *
 * 时间复杂度：O(n log n) - 对每个元素执行二分查找，每个二分查找是 O(log n)
 * 空间复杂度：O(1) - 只使用了常量级的额外空间
 */
public int[] twoSumBinarySearch(int[] numbers, int target) {
 // 参数校验
 if (numbers == null || numbers.length < 2) {
 throw new IllegalArgumentException("输入数组不能为 null 且长度必须至少为 2");
 }

 int n = numbers.length;

 // 遍历数组，对每个元素使用二分查找寻找另一个元素
 for (int i = 0; i < n; i++) {
 int complement = target - numbers[i];
 int left = i + 1;
 int right = n - 1;

 while (left <= right) {
 int mid = left + (right - left) / 2;
 if (numbers[mid] == complement) {
 return new int[]{i + 1, mid + 1}; // 注意索引从 1 开始
 } else if (numbers[mid] < complement) {
 left = mid + 1;
 } else {
 right = mid - 1;
 }
 }
 }

 // 根据题目描述，一定存在解，所以不会执行到这里
 throw new IllegalArgumentException("未找到符合条件的数对");
}
```

```
/**
 * 解法三：双指针（最优解）
 *
 * @param numbers 输入的有序数组
 * @param target 目标和
 * @return 两个数的索引数组（索引从 1 开始）
 *
 * 时间复杂度：O(n) - 只需要一次遍历数组
 * 空间复杂度：O(1) - 只使用了常量级的额外空间
 */

public int[] twoSum(int[] numbers, int target) {
 // 参数校验
 if (numbers == null || numbers.length < 2) {
 throw new IllegalArgumentException("输入数组不能为 null 且长度必须至少为 2");
 }

 // 初始化左右指针
 int left = 0;
 int right = numbers.length - 1;

 while (left < right) {
 int sum = numbers[left] + numbers[right];

 if (sum == target) {
 return new int[]{left + 1, right + 1}; // 注意索引从 1 开始
 } else if (sum < target) {
 // 和小于目标值，需要增大和，移动左指针
 left++;
 } else {
 // 和大于目标值，需要减小和，移动右指针
 right--;
 }
 }

 // 根据题目描述，一定存在解，所以不会执行到这里
 throw new IllegalArgumentException("未找到符合条件的数对");
}

/**
 * 解法四：优化的双指针实现
 * 当数组中有大量重复元素时，可以通过跳过相同的元素来提高效率
 *
 * @param numbers 输入的有序数组
```

```
* @param target 目标和
* @return 两个数的索引数组（索引从 1 开始）
*
* 时间复杂度: O(n) - 最坏情况下需要一次遍历数组
* 空间复杂度: O(1) - 只使用了常量级的额外空间
*/
public int[] twoSumOptimized(int[] numbers, int target) {
 // 参数校验
 if (numbers == null || numbers.length < 2) {
 throw new IllegalArgumentException("输入数组不能为 null 且长度必须至少为 2");
 }

 // 初始化左右指针
 int left = 0;
 int right = numbers.length - 1;

 while (left < right) {
 int sum = numbers[left] + numbers[right];

 if (sum == target) {
 return new int[]{left + 1, right + 1}; // 注意索引从 1 开始
 } else if (sum < target) {
 // 和小于目标值, 需要增大和, 移动左指针
 left++;
 // 跳过重复的元素
 while (left < right && numbers[left] == numbers[left - 1]) {
 left++;
 }
 } else {
 // 和大于目标值, 需要减小和, 移动右指针
 right--;
 // 跳过重复的元素
 while (left < right && numbers[right] == numbers[right + 1]) {
 right--;
 }
 }
 }

 // 根据题目描述, 一定存在解, 所以不会执行到这里
 throw new IllegalArgumentException("未找到符合条件的数对");
}
```

```
public class Code11_TwoSumII {
 /**
 * 测试函数
 */
 public static void test() {
 Solution solution = new Solution();

 // 测试用例 1
 int[] numbers1 = {2, 7, 11, 15};
 int target1 = 9;
 int[] expected1 = {1, 2};
 int[] result1 = solution.twoSum(numbers1, target1);
 System.out.println("测试用例 1:");
 System.out.println("输入数组: " + Arrays.toString(numbers1));
 System.out.println("目标值: " + target1);
 System.out.println("结果: " + Arrays.toString(result1));
 System.out.println("期望: " + Arrays.toString(expected1));
 System.out.println("测试通过: " + Arrays.equals(result1, expected1));
 System.out.println();

 // 测试用例 2
 int[] numbers2 = {2, 3, 4};
 int target2 = 6;
 int[] expected2 = {1, 3};
 int[] result2 = solution.twoSum(numbers2, target2);
 System.out.println("测试用例 2:");
 System.out.println("输入数组: " + Arrays.toString(numbers2));
 System.out.println("目标值: " + target2);
 System.out.println("结果: " + Arrays.toString(result2));
 System.out.println("期望: " + Arrays.toString(expected2));
 System.out.println("测试通过: " + Arrays.equals(result2, expected2));
 System.out.println();

 // 测试用例 3
 int[] numbers3 = {-1, 0};
 int target3 = -1;
 int[] expected3 = {1, 2};
 int[] result3 = solution.twoSum(numbers3, target3);
 System.out.println("测试用例 3:");
 System.out.println("输入数组: " + Arrays.toString(numbers3));
 System.out.println("目标值: " + target3);
 System.out.println("结果: " + Arrays.toString(result3));
 System.out.println("期望: " + Arrays.toString(expected3));
 }
}
```

```

System.out.println("测试通过: " + Arrays.equals(result3, expected3));
System.out.println();

// 测试用例 4 - 边界情况: 有重复元素
int[] numbers4 = {1, 1, 2, 4, 5};
int target4 = 2;
int[] expected4 = {1, 2};
int[] result4 = solution.twoSum(numbers4, target4);
System.out.println("测试用例 4 (有重复元素) :");
System.out.println("输入数组: " + Arrays.toString(numbers4));
System.out.println("目标值: " + target4);
System.out.println("结果: " + Arrays.toString(result4));
System.out.println("期望: " + Arrays.toString(expected4));
System.out.println("测试通过: " + Arrays.equals(result4, expected4));
System.out.println();

// 测试用例 5 - 边界情况: 数组长度为 2
int[] numbers5 = {1, 3};
int target5 = 4;
int[] expected5 = {1, 2};
int[] result5 = solution.twoSum(numbers5, target5);
System.out.println("测试用例 5 (数组长度为 2) :");
System.out.println("输入数组: " + Arrays.toString(numbers5));
System.out.println("目标值: " + target5);
System.out.println("结果: " + Arrays.toString(result5));
System.out.println("期望: " + Arrays.toString(expected5));
System.out.println("测试通过: " + Arrays.equals(result5, expected5));
System.out.println();
}

/**
 * 性能测试
 */
public static void performanceTest() {
 Solution solution = new Solution();

 // 创建一个较大的测试数组
 int size = 10000;
 int[] largeArray = new int[size];
 for (int i = 0; i < size; i++) {
 largeArray[i] = i * 2; // 生成偶数序列
 }
 int target = largeArray[size/2] + largeArray[size/2 + 1]; // 确保有解
}

```

```
// 测试解法二的性能
long startTime = System.nanoTime();
int[] result2 = solution.twoSumBinarySearch(largeArray, target);
long endTime = System.nanoTime();
long duration2 = (endTime - startTime) / 1000000; // 转换为毫秒
System.out.println("解法二（二分查找）耗时: " + duration2 + "ms");

// 测试解法三的性能
startTime = System.nanoTime();
int[] result3 = solution.twoSum(largeArray, target);
endTime = System.nanoTime();
long duration3 = (endTime - startTime) / 1000000; // 转换为毫秒
System.out.println("解法三（双指针）耗时: " + duration3 + "ms");

// 测试解法四的性能
startTime = System.nanoTime();
int[] result4 = solution.twoSumOptimized(largeArray, target);
endTime = System.nanoTime();
long duration4 = (endTime - startTime) / 1000000; // 转换为毫秒
System.out.println("解法四（优化双指针）耗时: " + duration4 + "ms");

// 验证结果是否一致
System.out.println("解法二和解法三结果一致: " + Arrays.equals(result2, result3));
System.out.println("解法三和解法四结果一致: " + Arrays.equals(result3, result4));

// 测试有重复元素的性能
int[] duplicateArray = new int[size];
for (int i = 0; i < size; i++) {
 duplicateArray[i] = i % 100; // 生成大量重复元素
}
target = 100; // 确保有解

startTime = System.nanoTime();
result3 = solution.twoSum(duplicateArray, target);
endTime = System.nanoTime();
duration3 = (endTime - startTime) / 1000000; // 转换为毫秒
System.out.println("\n重复元素测试 - 双指针耗时: " + duration3 + "ms");

startTime = System.nanoTime();
result4 = solution.twoSumOptimized(duplicateArray, target);
endTime = System.nanoTime();
duration4 = (endTime - startTime) / 1000000; // 转换为毫秒
```

```
System.out.println("重复元素测试 - 优化双指针耗时: " + duration4 + "ms");
System.out.println("在重复元素情况下, 优化效果: " + (duration3 > duration4 ? "有效" : "不明显"));
}

/**
 * 边界条件测试
 */
public static void boundaryTest() {
 Solution solution = new Solution();

 try {
 // 测试 null 输入
 solution.twoSum(null, 5);
 System.out.println("边界测试失败: null 输入没有抛出异常");
 } catch (IllegalArgumentException e) {
 System.out.println("边界测试通过: null 输入正确抛出异常: " + e.getMessage());
 }

 try {
 // 测试长度为 1 的输入
 solution.twoSum(new int[] {1}, 1);
 System.out.println("边界测试失败: 长度为 1 的输入没有抛出异常");
 } catch (IllegalArgumentException e) {
 System.out.println("边界测试通过: 长度为 1 的输入正确抛出异常: " + e.getMessage());
 }

 // 测试最大可能值
 int[] maxArray = {Integer.MAX_VALUE - 1, Integer.MAX_VALUE};
 int maxTarget = Integer.MAX_VALUE - 1 + Integer.MAX_VALUE - 1; // 避免溢出
 try {
 int[] result = solution.twoSum(maxArray, maxTarget);
 System.out.println("测试最大值: " + Arrays.toString(result));
 } catch (Exception e) {
 System.out.println("测试最大值时发生错误: " + e.getMessage());
 }

 // 测试最小可能值
 int[] minArray = {Integer.MIN_VALUE, Integer.MIN_VALUE + 1};
 int minTarget = Integer.MIN_VALUE + Integer.MIN_VALUE + 1;
 try {
 int[] result = solution.twoSum(minArray, minTarget);
 System.out.println("测试最小值: " + Arrays.toString(result));
 }
```

```
 } catch (Exception e) {
 System.out.println("测试最小值时发生错误: " + e.getMessage());
 }
 }

/**
 * 调试辅助函数 - 打印中间状态
 */
public static void debugTwoSum(int[] numbers, int target) {
 System.out.println("\n调试模式: ");
 System.out.println("输入数组: " + Arrays.toString(numbers));
 System.out.println("目标值: " + target);

 int left = 0;
 int right = numbers.length - 1;
 int step = 1;

 while (left < right) {
 int sum = numbers[left] + numbers[right];
 System.out.println("步骤 " + step + ": left=" + left + " (" + numbers[left] + "), " +
 "right=" + right + " (" + numbers[right] + "), " +
 "sum=" + sum);

 if (sum == target) {
 System.out.println("找到解: [" + (left + 1) + ", " + (right + 1) + "]");
 break;
 } else if (sum < target) {
 System.out.println("sum < target, 移动左指针");
 left++;
 } else {
 System.out.println("sum > target, 移动右指针");
 right--;
 }
 step++;
 }
}

public static void main(String[] args) {
 System.out.println("== 测试用例 ==");
 test();

 System.out.println("== 性能测试 ==");
 performanceTest();
}
```

```
System.out.println("==> 边界条件测试 ==>");
boundaryTest();

System.out.println("\n==> 调试演示 ==>");
debugTwoSum(new int[] {2, 7, 11, 15}, 9);
}
}

=====
```

文件: Code11\_TwoSumII.py

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

"""

两数之和 II – 输入有序数组

题目描述:

给你一个下标从 1 开始的整数数组 numbers，该数组已按非递减顺序排列。

请你从数组中找出满足相加之和等于目标数 target 的两个数。

如果设这两个数分别是 numbers[index1] 和 numbers[index2]，则  $1 \leq index1 < index2 \leq numbers.length$ 。

以长度为 2 的整数数组 [index1, index2] 的形式返回这两个整数的下标 index1 和 index2。

你可以假设每个输入只对应唯一的答案，而且你不能重复使用相同的元素。

你所设计的解决方案必须只使用常量级的额外空间。

示例:

输入: numbers = [2, 7, 11, 15], target = 9

输出: [1, 2]

解释: 2 与 7 之和等于目标数 9。因此 index1 = 1, index2 = 2。返回 [1, 2]。

输入: numbers = [2, 3, 4], target = 6

输出: [1, 3]

解释: 2 与 4 之和等于目标数 6。因此 index1 = 1, index2 = 3。返回 [1, 3]。

输入: numbers = [-1, 0], target = -1

输出: [1, 2]

解释: -1 与 0 之和等于目标数 -1。因此 index1 = 1, index2 = 2。返回 [1, 2]。

解题思路:

使用双指针法。由于数组已经排序，我们可以使用两个指针分别指向数组的开始和结束。

如果两个指针指向的数字之和等于目标值，则返回它们的索引（注意题目要求索引从 1 开始）。

如果和小于目标值，则将左指针右移以增大和。

如果和大于目标值，则将右指针左移以减小和。

时间复杂度： $O(n)$  – 最多遍历一次数组

空间复杂度： $O(1)$  – 只使用了常数级别的额外空间

是否最优解：是 – 基于比较的算法下界为  $O(n)$ ，本算法已达到最优

相关题目：

1. LeetCode 1 – 两数之和（无序数组，使用哈希表）
2. LeetCode 167 – 两数之和 II – 输入有序数组（当前题目）
3. LeetCode 15 – 三数之和（排序+双指针）
4. LeetCode 18 – 四数之和（排序+双指针）

工程化考虑：

1. 输入验证：检查数组是否为空或长度小于 2
2. 异常处理：题目保证有唯一解，但在实际工程中可能需要处理无解的情况
3. 边界条件：处理负数、零和正数混合的情况

语言特性差异：

Java：使用数组索引访问，需要手动处理索引偏移（题目要求索引从 1 开始）

C++：可使用 vector，指针运算更灵活

Python：可使用列表，支持负索引访问

极端输入场景：

1. 数组包含负数
2. 目标值为 0
3. 数组长度为 2
4. 解在数组两端

与机器学习等领域的联系：

1. 在特征选择中，可能需要找到两个特征的组合满足特定条件
2. 在推荐系统中，可能需要找到两个物品的组合满足用户偏好

"""

```
def two_sum_brute_force(numbers, target):
```

```
 """
```

暴力解法：双重循环遍历所有可能的组合

```
:param numbers: 已排序的整数数组
```

```
:param target: 目标和
```

```
:return: 包含两个索引的数组（索引从 1 开始）
```

```

"""
n = len(numbers)
边界条件检查
if n < 2:
 return [-1, -1]

双重循环遍历所有可能的组合
for i in range(n):
 for j in range(i + 1, n):
 if numbers[i] + numbers[j] == target:
 return [i + 1, j + 1] # 索引从 1 开始

return [-1, -1]

```

```
def two_sum_binary_search(numbers, target):
```

```
"""
```

二分查找解法：对于每个元素，在剩余部分中二分查找目标差值

:param numbers: 已排序的整数数组

:param target: 目标和

:return: 包含两个索引的数组（索引从 1 开始）

```
"""

n = len(numbers)
边界条件检查
if n < 2:
 return [-1, -1]

遍历每个元素，对剩余部分进行二分查找
for i in range(n - 1):
 complement = target - numbers[i]
 # 在 i+1 到 n-1 范围内二分查找 complement
 left, right = i + 1, n - 1
 while left <= right:
 mid = left + (right - left) // 2
 if numbers[mid] == complement:
 return [i + 1, mid + 1] # 索引从 1 开始
 elif numbers[mid] < complement:
 left = mid + 1
 else:
 right = mid - 1

return [-1, -1]
```

```
return [-1, -1]
```

```
def two_sum_two_pointers(numbers, target):
 """
 双指针解法：利用数组有序的特性，从两端向中间移动指针

 :param numbers: 已排序的整数数组
 :param target: 目标和
 :return: 包含两个索引的数组（索引从 1 开始）
 """

 # 边界条件检查
 if not numbers or len(numbers) < 2:
 return [-1, -1]

 # 初始化双指针
 left = 0 # 左指针指向数组开始
 right = len(numbers) - 1 # 右指针指向数组结束

 # 当左指针小于右指针时继续循环
 while left < right:
 # 计算当前两个指针指向元素的和
 sum_val = numbers[left] + numbers[right]

 # 如果和等于目标值，返回索引（注意题目要求索引从 1 开始）
 if sum_val == target:
 return [left + 1, right + 1]
 # 如果和小于目标值，将左指针右移以增大和
 elif sum_val < target:
 left += 1
 # 如果和大于目标值，将右指针左移以减小和
 else:
 right -= 1

 # 根据题目描述，保证有唯一解，此行理论上不会执行到
 return [-1, -1]
```

```
def two_sum_optimized(numbers, target):
 """
 优化的双指针解法：增加跳过重复元素的逻辑

 :param numbers: 已排序的整数数组
 :param target: 目标和
```

```

:return: 包含两个索引的数组（索引从 1 开始）
"""

边界条件检查
if not numbers or len(numbers) < 2:
 return [-1, -1]

初始化双指针
left = 0
right = len(numbers) - 1

while left < right:
 sum_val = numbers[left] + numbers[right]

 if sum_val == target:
 return [left + 1, right + 1]
 elif sum_val < target:
 # 跳过重复元素，避免无效计算
 while left < right and numbers[left] == numbers[left + 1]:
 left += 1
 left += 1
 else:
 # 跳过重复元素，避免无效计算
 while left < right and numbers[right] == numbers[right - 1]:
 right -= 1
 right -= 1

return [-1, -1]

主函数，默认使用双指针解法
def two_sum(numbers, target):
 return two_sum_two_pointers(numbers, target)

def test():
 """测试函数，测试所有解法"""
 # 测试用例 1：常规输入
 numbers1 = [2, 7, 11, 15]
 target1 = 9
 print("\n==== 测试用例 1：常规输入 ===")
 print(f"输入: numbers={numbers1}, target={target1}")
 print(f"双指针解法结果: {two_sum_two_pointers(numbers1, target1)}")
 print(f"优化双指针解法结果: {two_sum_optimized(numbers1, target1)}")
 print(f"二分查找解法结果: {two_sum_binary_search(numbers1, target1)}")

```

```
print(f"暴力解法结果: {two_sum_brute_force(numbers1, target1)}")\n\n# 测试用例 2: 解在数组两端\nnumbers2 = [2, 3, 4]\ntarget2 = 6\nprint("\n==== 测试用例 2: 解在数组两端 ===")\nprint(f"输入: numbers={numbers2}, target={target2}")\nprint(f"双指针解法结果: {two_sum_two_pointers(numbers2, target2)}")\n\n# 测试用例 3: 包含负数\nnumbers3 = [-1, 0]\ntarget3 = -1\nprint("\n==== 测试用例 3: 包含负数 ===")\nprint(f"输入: numbers={numbers3}, target={target3}")\nprint(f"双指针解法结果: {two_sum_two_pointers(numbers3, target3)}")\n\n# 测试用例 4: 数组长度为 2\nnumbers4 = [1, 2]\ntarget4 = 3\nprint("\n==== 测试用例 4: 数组长度为 2 ===")\nprint(f"输入: numbers={numbers4}, target={target4}")\nprint(f"双指针解法结果: {two_sum_two_pointers(numbers4, target4)}")\n\n# 测试用例 5: 有重复元素\nnumbers5 = [1, 2, 3, 4, 4, 9, 56, 90]\ntarget5 = 8\nprint("\n==== 测试用例 5: 有重复元素 ===")\nprint(f"输入: numbers={numbers5}, target={target5}")\nprint(f"双指针解法结果: {two_sum_two_pointers(numbers5, target5)}")\nprint(f"优化双指针解法结果: {two_sum_optimized(numbers5, target5)}")\n\n# 测试用例 6: 全是负数\nnumbers6 = [-5, -4, -3, -2, -1]\ntarget6 = -7\nprint("\n==== 测试用例 6: 全是负数 ===")\nprint(f"输入: numbers={numbers6}, target={target6}")\nprint(f"双指针解法结果: {two_sum_two_pointers(numbers6, target6)}")\n\n# 测试用例 7: 目标值为 0\nnumbers7 = [-3, -2, -1, 0, 1, 2, 3]\ntarget7 = 0\nprint("\n==== 测试用例 7: 目标值为 0 ===")\nprint(f"输入: numbers={numbers7}, target={target7}")
```

```
print(f"双指针解法结果: {two_sum_two_pointers(numbers7, target7)}")
print(f"优化双指针解法结果: {two_sum_optimized(numbers7, target7)}")

def performance_test():
 """性能测试函数"""
 import time

 # 生成大数组进行性能测试
 print("\n==== 性能测试 ====")
 n = 10000
 large_numbers = list(range(n))
 large_target = 2 * n - 3 # 确保解在数组两端

 # 测试暴力解法（仅对小数组测试，避免超时）
 small_numbers = list(range(100))
 small_target = 197

 print("暴力解法（小数组）:")
 start_time = time.time()
 two_sum_brute_force(small_numbers, small_target)
 print(f"耗时: {(time.time() - start_time) * 1000:.4f} ms")

 print("二分查找解法:")
 start_time = time.time()
 two_sum_binary_search(large_numbers, large_target)
 print(f"耗时: {(time.time() - start_time) * 1000:.4f} ms")

 print("双指针解法:")
 start_time = time.time()
 two_sum_two_pointers(large_numbers, large_target)
 print(f"耗时: {(time.time() - start_time) * 1000:.4f} ms")

 print("优化双指针解法:")
 start_time = time.time()
 two_sum_optimized(large_numbers, large_target)
 print(f"耗时: {(time.time() - start_time) * 1000:.4f} ms")

def edge_case_test():
 """边界条件测试"""
 print("\n==== 边界条件测试 ====")
```

```
空数组
print("空数组:")
print(f"结果: {two_sum_two_pointers([], 5)}")

单元素数组
print("单元素数组:")
print(f"结果: {two_sum_two_pointers([1], 1)}")

大数测试
print("大数测试:")
large_num1 = 10**9
large_num2 = 10**9
large_array = [large_num1, large_num2]
print(f"结果: {two_sum_two_pointers(large_array, 2*10**9)}")

所有元素相同
print("所有元素相同:")
same_array = [2, 2, 2, 2, 2]
print(f"结果: {two_sum_two_pointers(same_array, 4)}")

def algorithm_analysis():
 """算法分析"""
 print("\n==== 算法分析 ===")
 print("1. 暴力解法")
 print(" - 时间复杂度: O(n^2)")
 print(" - 空间复杂度: O(1)")
 print(" - 适用场景: 小规模数据")
 print("\n2. 二分查找解法")
 print(" - 时间复杂度: O(n log n)")
 print(" - 空间复杂度: O(1)")
 print(" - 适用场景: 中等规模数据")
 print("\n3. 双指针解法")
 print(" - 时间复杂度: O(n)")
 print(" - 空间复杂度: O(1)")
 print(" - 适用场景: 所有规模数据, 特别是大数据集")
 print(" - 最优解的原因: 利用数组有序的特性, 只需一次遍历")
 print("\n4. 优化双指针解法")
 print(" - 时间复杂度: O(n), 但在重复元素较多时常数因子更小")
 print(" - 空间复杂度: O(1)")
 print(" - 适用场景: 包含大量重复元素的有序数组")

def engineering_considerations():
```

```
"""工程化考量"""
print("\n==== 工程化考量 ===")
print("1. 异常处理:")
print(" - 对空数组和单元素数组进行了边界检查")
print(" - 返回[-1, -1]表示未找到解，便于调用者识别")
print("\n2. 性能优化:")
print(" - 在优化版本中增加了跳过重复元素的逻辑")
print(" - 对于不同规模的数据选择合适的算法")
print("\n3. 代码可读性:")
print(" - 详细的函数注释和参数说明")
print(" - 清晰的变量命名")
print("\n4. 多语言实现对比:")
print(" - Python: 语法简洁，内置列表操作便捷")
print(" - Java: 类型安全，数组操作需要手动索引")
print(" - C++: 指针操作灵活，性能最优")
print("\n5. 调试支持:")
print(" - 完整的测试函数覆盖各种情况")
print(" - 性能测试帮助识别瓶颈")
```

```
主函数
if __name__ == "__main__":
 print("==== 两数之和 II - 输入有序数组 算法实现 ===")

运行基本测试
test()

运行边界条件测试
edge_case_test()

运行性能测试
performance_test()

算法分析
algorithm_analysis()

工程化考量
engineering_considerations()

print("\n==== 测试完成 ===")
```

```
=====
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

/***
 * 三数之和
 *
 * 题目描述:
 * 给你一个整数数组 nums，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k，
 * 还要满足 nums[i] + nums[j] + nums[k] == 0。请你返回所有和为 0 且不重复的三元组。
 * 注意：答案中不可以包含重复的三元组。
 *
 * 示例:
 * 输入: nums = [-1, 0, 1, 2, -1, -4]
 * 输出: [[-1, -1, 2], [-1, 0, 1]]
 * 解释:
 * nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0 。
 * nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0 。
 * nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0 。
 * 不同的三元组是 [-1, 0, 1] 和 [-1, -1, 2] 。
 * 注意，输出的顺序和三元组的顺序并不重要。
 *
 * 输入: nums = [0, 1, 1]
 * 输出: []
 * 解释: 唯一可能的三元组和不为 0 。
 *
 * 输入: nums = [0, 0, 0]
 * 输出: [[0, 0, 0]]
 * 解释: 唯一可能的三元组和为 0 。
 *
 * 解题思路:
 * 1. 首先对数组进行排序，这样可以方便使用双指针法，并且便于去重。
 * 2. 遍历数组，固定第一个数 nums[i]，然后在剩余的数组中使用双指针法寻找两个数，使得这三个数的和为 0。
 * 3. 使用双指针法：左指针指向 i+1，右指针指向数组末尾。
 * - 如果三数之和等于 0，则找到一个解，同时移动左右指针。
 * - 如果三数之和小于 0，则左指针右移。
 * - 如果三数之和大于 0，则右指针左移。
 * 4. 注意去重处理：
 * - 固定的第一个数如果相同则跳过
```

- \* - 找到解后，左右指针需要跳过相同的数
- \*
- \* 时间复杂度:  $O(n^2)$  - 外层循环  $O(n)$ ，内层双指针  $O(n)$
- \* 空间复杂度:  $O(1)$  - 不考虑返回结果的空间
- \* 是否最优解: 是 - 基于比较的算法下界为  $O(n^2)$ ，本算法已达到最优
- \*
- \* 相关题目:
  - \* 1. LeetCode 1 - 两数之和（无序数组，使用哈希表）
  - \* 2. LeetCode 167 - 两数之和 II - 输入有序数组（排序+双指针）
  - \* 3. LeetCode 15 - 三数之和（当前题目）
  - \* 4. LeetCode 18 - 四数之和（排序+双指针）
  - \* 5. LeetCode 16 - 最接近的三数之和
- \*
- \* 工程化考虑:
  - \* 1. 输入验证: 检查数组是否为空或长度小于 3
  - \* 2. 异常处理: 处理各种边界情况
  - \* 3. 边界条件: 处理全为 0、全为正数、全为负数的情况
- \*
- \* 语言特性差异:
  - \* Java: 使用 ArrayList 存储结果，需要导入相关包
  - \* C++: 可使用 vector 存储结果
  - \* Python: 可使用列表存储结果
- \*
- \* 极端输入场景:
  - \* 1. 数组全为 0
  - \* 2. 数组全为正数或全为负数
  - \* 3. 数组长度小于 3
  - \* 4. 数组包含大量重复元素
- \*
- \* 与机器学习等领域的联系:
  - \* 1. 在特征选择中，可能需要找到三个特征的组合满足特定条件
  - \* 2. 在推荐系统中，可能需要找到三个物品的组合满足用户偏好
- \*/
- class Solution {
- public:
- /\*\*
 \* 查找所有和为 0 的不重复三元组
 \*
 \* @param nums 整数数组
 \* @return 所有和为 0 的不重复三元组列表
 \*/
 vector<vector<int>> threeSum(vector<int>& nums) {
 // 结果列表
 }
}

```
vector<vector<int>> result;

// 边界条件检查
if (nums.size() < 3) {
 return result;
}

// 对数组进行排序，时间复杂度 O(n log n)
sort(nums.begin(), nums.end());

// 遍历数组，固定第一个数
for (int i = 0; i < nums.size() - 2; i++) {
 // 如果当前数字大于 0，则三数之和一定大于 0，结束循环
 if (nums[i] > 0) {
 break;
 }

 // 跳过重复元素，避免出现重复解
 if (i > 0 && nums[i] == nums[i - 1]) {
 continue;
 }

 // 使用双指针在剩余数组中寻找另外两个数
 int left = i + 1;
 int right = nums.size() - 1;

 while (left < right) {
 int sum = nums[i] + nums[left] + nums[right];

 if (sum == 0) {
 // 找到一个解
 result.push_back({nums[i], nums[left], nums[right]});
 }

 // 跳过重复元素
 while (left < right && nums[left] == nums[left + 1]) {
 left++;
 }

 while (left < right && nums[right] == nums[right - 1]) {
 right--;
 }

 // 移动指针继续寻找
 left++;
 }
}
```

```

 right--;
 } else if (sum < 0) {
 // 和小于 0, 左指针右移
 left++;
 } else {
 // 和大于 0, 右指针左移
 right--;
 }
}

return result;
}
};

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

 // 测试用例 1: [-1, 0, 1, 2, -1, -4] -> [[-1, -1, 2], [-1, 0, 1]]
 vector<int> nums1 = {-1, 0, 1, 2, -1, -4};
 vector<vector<int>> result1 = solution.threeSum(nums1);
 cout << "Test 1: nums=[";
 for (int i = 0; i < nums1.size(); i++) {
 cout << nums1[i];
 if (i < nums1.size() - 1) cout << ",";
 }
 cout << "]" << endl;
 cout << "Result: [";
 for (int i = 0; i < result1.size(); i++) {
 cout << "[";
 for (int j = 0; j < result1[i].size(); j++) {
 cout << result1[i][j];
 if (j < result1[i].size() - 1) cout << ",";
 }
 cout << "]";
 if (i < result1.size() - 1) cout << ",";
 }
 cout << "]" << endl;
}

// 测试用例 2: [0, 1, 1] -> []
vector<int> nums2 = {0, 1, 1};
vector<vector<int>> result2 = solution.threeSum(nums2);

```

```

cout << "Test 2: nums=[";
for (int i = 0; i < nums2.size(); i++) {
 cout << nums2[i];
 if (i < nums2.size() - 1) cout << ",";
}
cout << "]" << endl;
cout << "Result: [";
for (int i = 0; i < result2.size(); i++) {
 cout << "[";
 for (int j = 0; j < result2[i].size(); j++) {
 cout << result2[i][j];
 if (j < result2[i].size() - 1) cout << ",";
 }
 cout << "]";
 if (i < result2.size() - 1) cout << ",";
}
cout << "]" << endl;

// 测试用例 3: [0,0,0] -> [[0,0,0]]
vector<int> nums3 = {0, 0, 0};
vector<vector<int>> result3 = solution.threeSum(nums3);
cout << "Test 3: nums=[";
for (int i = 0; i < nums3.size(); i++) {
 cout << nums3[i];
 if (i < nums3.size() - 1) cout << ",";
}
cout << "]" << endl;
cout << "Result: [";
for (int i = 0; i < result3.size(); i++) {
 cout << "[";
 for (int j = 0; j < result3[i].size(); j++) {
 cout << result3[i][j];
 if (j < result3[i].size() - 1) cout << ",";
 }
 cout << "]";
 if (i < result3.size() - 1) cout << ",";
}
cout << "]" << endl;
}

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

```

}

=====

文件: Code12\_ThreeSum.java

=====

```
package class050;
```

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

```
/**
```

```
* 三数之和
```

```
*
```

```
* 题目描述:
```

```
* 给你一个整数数组 nums，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k，
```

```
* 还要满足 nums[i] + nums[j] + nums[k] == 0。请你返回所有和为 0 且不重复的三元组。
```

```
* 注意：答案中不可以包含重复的三元组。
```

```
*
```

```
* 示例：
```

```
* 输入：nums = [-1, 0, 1, 2, -1, -4]
```

```
* 输出：[[-1, -1, 2], [-1, 0, 1]]
```

```
* 解释：
```

```
* nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0。
```

```
* nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0。
```

```
* nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0。
```

```
* 不同的三元组是 [-1, 0, 1] 和 [-1, -1, 2]。
```

```
* 注意，输出的顺序和三元组的顺序并不重要。
```

```
*
```

```
* 输入：nums = [0, 1, 1]
```

```
* 输出：[]
```

```
* 解释：唯一可能的三元组和不为 0。
```

```
*
```

```
* 输入：nums = [0, 0, 0]
```

```
* 输出：[[0, 0, 0]]
```

```
* 解释：唯一可能的三元组和为 0。
```

```
*
```

```
* 解题思路：
```

```
* 1. 首先对数组进行排序，这样可以方便使用双指针法，并且便于去重。
```

```
* 2. 遍历数组，固定第一个数 nums[i]，然后在剩余的数组中使用双指针法寻找两个数，
```

```
* 使得这三个数的和为 0。
```

- \* 3. 使用双指针法：左指针指向  $i+1$ ，右指针指向数组末尾。
  - 如果三数之和等于 0，则找到一个解，同时移动左右指针。
  - 如果三数之和小于 0，则左指针右移。
  - 如果三数之和大于 0，则右指针左移。
- \* 4. 注意去重处理：
  - 固定的第一个数如果相同则跳过
  - 找到解后，左右指针需要跳过相同的数
- \*
- \* 时间复杂度:  $O(n^2)$  - 外层循环  $O(n)$ ，内层双指针  $O(n)$
- \* 空间复杂度:  $O(1)$  - 不考虑返回结果的空间
- \* 是否最优解: 是 - 基于比较的算法下界为  $O(n^2)$ ，本算法已达到最优
- \*
- \* 相关题目：
  - \* 1. LeetCode 1 - 两数之和（无序数组，使用哈希表）
  - \* 2. LeetCode 167 - 两数之和 II - 输入有序数组（排序+双指针）
  - \* 3. LeetCode 15 - 三数之和（当前题目）
  - \* 4. LeetCode 18 - 四数之和（排序+双指针）
  - \* 5. LeetCode 16 - 最接近的三数之和
- \*
- \* 工程化考虑：
  - \* 1. 输入验证：检查数组是否为空或长度小于 3
  - \* 2. 异常处理：处理各种边界情况
  - \* 3. 边界条件：处理全为 0、全为正数、全为负数的情况
- \*
- \* 语言特性差异：
  - \* Java：使用 ArrayList 存储结果，需要导入相关包
  - \* C++：可使用 vector 存储结果
  - \* Python：可使用列表存储结果
- \*
- \* 极端输入场景：
  - \* 1. 数组全为 0
  - \* 2. 数组全为正数或全为负数
  - \* 3. 数组长度小于 3
  - \* 4. 数组包含大量重复元素
- \*
- \* 与机器学习等领域的联系：
  - \* 1. 在特征选择中，可能需要找到三个特征的组合满足特定条件
  - \* 2. 在推荐系统中，可能需要找到三个物品的组合满足用户偏好

```
public class Code12_ThreeSum {
```

```
/**
```

```
 * 查找所有和为 0 的不重复三元组
```

```
*
* @param nums 整数数组
* @return 所有和为 0 的不重复三元组列表
*/
public static List<List<Integer>> threeSum(int[] nums) {
 // 结果列表
 List<List<Integer>> result = new ArrayList<>();

 // 边界条件检查
 if (nums == null || nums.length < 3) {
 return result;
 }

 // 对数组进行排序，时间复杂度 O(n log n)
 Arrays.sort(nums);

 // 遍历数组，固定第一个数
 for (int i = 0; i < nums.length - 2; i++) {
 // 如果当前数字大于 0，则三数之和一定大于 0，结束循环
 if (nums[i] > 0) {
 break;
 }

 // 跳过重复元素，避免出现重复解
 if (i > 0 && nums[i] == nums[i - 1]) {
 continue;
 }

 // 使用双指针在剩余数组中寻找另外两个数
 int left = i + 1;
 int right = nums.length - 1;

 while (left < right) {
 int sum = nums[i] + nums[left] + nums[right];

 if (sum == 0) {
 // 找到一个解
 result.add(Arrays.asList(nums[i], nums[left], nums[right]));

 // 跳过重复元素
 while (left < right && nums[left] == nums[left + 1]) {
 left++;
 }
 }
 }
 }
}
```

```
 while (left < right && nums[right] == nums[right - 1]) {
 right--;
 }

 // 移动指针继续寻找
 left++;
 right--;
 } else if (sum < 0) {
 // 和小于 0, 左指针右移
 left++;
 } else {
 // 和大于 0, 右指针左移
 right--;
 }
}

return result;
}

/**
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: [-1, 0, 1, 2, -1, -4] -> [[-1, -1, 2], [-1, 0, 1]]
 int[] nums1 = {-1, 0, 1, 2, -1, -4};
 List<List<Integer>> result1 = threeSum(nums1);
 System.out.println("Test 1: nums=" + Arrays.toString(nums1));
 System.out.println("Result: " + result1);

 // 测试用例 2: [0, 1, 1] -> []
 int[] nums2 = {0, 1, 1};
 List<List<Integer>> result2 = threeSum(nums2);
 System.out.println("Test 2: nums=" + Arrays.toString(nums2));
 System.out.println("Result: " + result2);

 // 测试用例 3: [0, 0, 0] -> [[0, 0, 0]]
 int[] nums3 = {0, 0, 0};
 List<List<Integer>> result3 = threeSum(nums3);
 System.out.println("Test 3: nums=" + Arrays.toString(nums3));
 System.out.println("Result: " + result3);
}
```

文件: Code12\_ThreeSum.py

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

"""

三数之和

题目描述:

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足  $i \neq j, i \neq k$  且  $j \neq k$ ，

还要满足  $nums[i] + nums[j] + nums[k] == 0$ 。请你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例：

输入: `nums = [-1, 0, 1, 2, -1, -4]`

输出: `[[-1, -1, 2], [-1, 0, 1]]`

解释:

`nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0`。

`nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0`。

`nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0`。

不同的三元组是 `[-1, 0, 1]` 和 `[-1, -1, 2]`。

注意，输出的顺序和三元组的顺序并不重要。

输入: `nums = [0, 1, 1]`

输出: `[]`

解释: 唯一可能的三元组和不为 0。

输入: `nums = [0, 0, 0]`

输出: `[[0, 0, 0]]`

解释: 唯一可能的三元组和为 0。

解题思路:

1. 首先对数组进行排序，这样可以方便使用双指针法，并且便于去重。
2. 遍历数组，固定第一个数 `nums[i]`，然后在剩余的数组中使用双指针法寻找两个数，使得这三个数的和为 0。
3. 使用双指针法：左指针指向 `i+1`，右指针指向数组末尾。
  - 如果三数之和等于 0，则找到一个解，同时移动左右指针。
  - 如果三数之和小于 0，则左指针右移。
  - 如果三数之和大于 0，则右指针左移。

#### 4. 注意去重处理:

- 固定的第一个数如果相同则跳过
- 找到解后，左右指针需要跳过相同的数

时间复杂度:  $O(n^2)$  - 外层循环  $O(n)$ ，内层双指针  $O(n)$

空间复杂度:  $O(1)$  - 不考虑返回结果的空间

是否最优解: 是 - 基于比较的算法下界为  $O(n^2)$ ，本算法已达到最优

相关题目:

1. LeetCode 1 - 两数之和（无序数组，使用哈希表）
2. LeetCode 167 - 两数之和 II - 输入有序数组（排序+双指针）
3. LeetCode 15 - 三数之和（当前题目）
4. LeetCode 18 - 四数之和（排序+双指针）
5. LeetCode 16 - 最接近的三数之和

工程化考虑:

1. 输入验证: 检查数组是否为空或长度小于 3
2. 异常处理: 处理各种边界情况
3. 边界条件: 处理全为 0、全为正数、全为负数的情况

语言特性差异:

Java: 使用 ArrayList 存储结果，需要导入相关包

C++: 可使用 vector 存储结果

Python: 可使用列表存储结果

极端输入场景:

1. 数组全为 0
2. 数组全为正数或全为负数
3. 数组长度小于 3
4. 数组包含大量重复元素

与机器学习等领域的联系:

1. 在特征选择中，可能需要找到三个特征的组合满足特定条件
2. 在推荐系统中，可能需要找到三个物品的组合满足用户偏好

"""

```
def three_sum(nums):
```

```
 """
```

```
 查找所有和为 0 的不重复三元组
```

```
 :param nums: 整数数组
```

```
 :return: 所有和为 0 的不重复三元组列表
```

```

"""
结果列表
result = []

边界条件检查
if not nums or len(nums) < 3:
 return result

对数组进行排序，时间复杂度 O(n log n)
nums.sort()

遍历数组，固定第一个数
for i in range(len(nums) - 2):
 # 如果当前数字大于 0，则三数之和一定大于 0，结束循环
 if nums[i] > 0:
 break

 # 跳过重复元素，避免出现重复解
 if i > 0 and nums[i] == nums[i - 1]:
 continue

 # 使用双指针在剩余数组中寻找另外两个数
 left = i + 1
 right = len(nums) - 1

 while left < right:
 sum_val = nums[i] + nums[left] + nums[right]

 if sum_val == 0:
 # 找到一个解
 result.append([nums[i], nums[left], nums[right]])

 # 跳过重复元素
 while left < right and nums[left] == nums[left + 1]:
 left += 1
 while left < right and nums[right] == nums[right - 1]:
 right -= 1

 # 移动指针继续寻找
 left += 1
 right -= 1

 elif sum_val < 0:
 # 和小于 0，左指针右移

```

```

 left += 1
 else:
 # 和大于 0, 右指针左移
 right -= 1

 return result

def test():
 """测试函数"""
 # 测试用例 1: [-1, 0, 1, 2, -1, -4] -> [[-1, -1, 2], [-1, 0, 1]]
 nums1 = [-1, 0, 1, 2, -1, -4]
 result1 = three_sum(nums1)
 print(f"Test 1: nums={nums1}")
 print(f"Result: {result1}")

 # 测试用例 2: [0, 1, 1] -> []
 nums2 = [0, 1, 1]
 result2 = three_sum(nums2)
 print(f"Test 2: nums={nums2}")
 print(f"Result: {result2}")

 # 测试用例 3: [0, 0, 0] -> [[0, 0, 0]]
 nums3 = [0, 0, 0]
 result3 = three_sum(nums3)
 print(f"Test 3: nums={nums3}")
 print(f"Result: {result3}")

主函数
if __name__ == "__main__":
 test()

```

=====

文件: Code13\_FourSum.java

=====

```

package class050;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

```

```
/**
 * 四数之和
 *
 * 题目描述:
 * 给你一个由 n 个整数组成的数组 nums，和一个目标值 target。
 * 请你找出并返回满足下述全部条件且不重复的四元组 [nums[a], nums[b], nums[c], nums[d]]：
 * 1. 0 <= a, b, c, d < n
 * 2. a、b、c 和 d 互不相同
 * 3. nums[a] + nums[b] + nums[c] + nums[d] == target
 *
 * 示例:
 * 输入: nums = [1, 0, -1, 0, -2, 2], target = 0
 * 输出: [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
 *
 * 输入: nums = [2, 2, 2, 2, 2], target = 8
 * 输出: [[2, 2, 2, 2]]
 *
 * 解题思路:
 * 1. 首先对数组进行排序，这样可以方便使用双指针法，并且便于去重。
 * 2. 使用两层循环固定前两个数，然后在剩余的数组中使用双指针法寻找另外两个数，
 * 使得这四个数的和等于目标值。
 * 3. 使用双指针法：左指针指向第二个固定数的下一个位置，右指针指向数组末尾。
 * - 如果四数之和等于目标值，则找到一个解，同时移动左右指针。
 * - 如果四数之和小于目标值，则左指针右移。
 * - 如果四数之和大于目标值，则右指针左移。
 * 4. 注意去重处理：
 * - 第一层循环固定的第一个数如果相同则跳过
 * - 第二层循环固定的第二个数如果相同则跳过
 * - 找到解后，左右指针需要跳过相同的数
 *
 * 时间复杂度: O(n³) - 两层循环 O(n²)，内层双指针 O(n)
 * 空间复杂度: O(1) - 不考虑返回结果的空间
 * 是否最优解: 是 - 基于比较的算法下界为 O(n³)，本算法已达到最优
 *
 * 相关题目:
 * 1. LeetCode 1 - 两数之和（无序数组，使用哈希表）
 * 2. LeetCode 167 - 两数之和 II - 输入有序数组（排序+双指针）
 * 3. LeetCode 15 - 三数之和（排序+双指针）
 * 4. LeetCode 18 - 四数之和（当前题目）
 *
 * 工程化考虑:
 * 1. 输入验证: 检查数组是否为空或长度小于 4
 * 2. 异常处理: 处理各种边界情况
```

- \* 3. 边界条件：处理全为相同元素的情况
- \*
- \* 语言特性差异：
- \* Java：使用 ArrayList 存储结果，需要导入相关包
- \* C++：可使用 vector 存储结果
- \* Python：可使用列表存储结果
- \*
- \* 极端输入场景：
- \* 1. 数组全为相同元素
- \* 2. 数组长度小于 4
- \* 3. 数组包含大量重复元素
- \*
- \* 与机器学习等领域的联系：
- \* 1. 在特征选择中，可能需要找到四个特征的组合满足特定条件
- \* 2. 在推荐系统中，可能需要找到四个物品的组合满足用户偏好

```
 */
public class Code13_FourSum {

 /**
 * 查找所有和为目标值的不重复四元组
 *
 * @param nums 整数数组
 * @param target 目标值
 * @return 所有和为目标值的不重复四元组列表
 */
 public static List<List<Integer>> fourSum(int[] nums, int target) {
 // 结果列表
 List<List<Integer>> result = new ArrayList<>();

 // 边界条件检查
 if (nums == null || nums.length < 4) {
 return result;
 }

 // 对数组进行排序，时间复杂度 O(n log n)
 Arrays.sort(nums);

 int n = nums.length;

 // 遍历数组，固定第一个数
 for (int i = 0; i < n - 3; i++) {
 // 跳过重复元素，避免出现重复解
 if (i > 0 && nums[i] == nums[i - 1]) {
```

```
 continue;
 }

 // 遍历数组，固定第二个数
 for (int j = i + 1; j < n - 2; j++) {
 // 跳过重复元素，避免出现重复解
 if (j > i + 1 && nums[j] == nums[j - 1]) {
 continue;
 }

 // 使用双指针在剩余数组中寻找另外两个数
 int left = j + 1;
 int right = n - 1;

 while (left < right) {
 long sum = (long) nums[i] + nums[j] + nums[left] + nums[right];

 if (sum == target) {
 // 找到一个解
 result.add(Arrays.asList(nums[i], nums[j], nums[left], nums[right]));

 // 跳过重复元素
 while (left < right && nums[left] == nums[left + 1]) {
 left++;
 }

 while (left < right && nums[right] == nums[right - 1]) {
 right--;
 }
 }

 // 移动指针继续寻找
 left++;
 right--;
 }

 } else if (sum < target) {
 // 和小于目标值，左指针右移
 left++;
 } else {
 // 和大于目标值，右指针左移
 right--;
 }
 }
}
```

```

 return result;
 }

/**
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: [1, 0, -1, 0, -2, 2], target = 0 -> [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
 int[] nums1 = {1, 0, -1, 0, -2, 2};
 int target1 = 0;
 List<List<Integer>> result1 = fourSum(nums1, target1);
 System.out.println("Test 1: nums=" + Arrays.toString(nums1) + ", target=" + target1);
 System.out.println("Result: " + result1);

 // 测试用例 2: [2, 2, 2, 2, 2], target = 8 -> [[2, 2, 2, 2]]
 int[] nums2 = {2, 2, 2, 2, 2};
 int target2 = 8;
 List<List<Integer>> result2 = fourSum(nums2, target2);
 System.out.println("Test 2: nums=" + Arrays.toString(nums2) + ", target=" + target2);
 System.out.println("Result: " + result2);
}
}

```

文件: Code13\_FourSum.py

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

```

"""

四数之和

题目描述:

给你一个由 n 个整数组成的数组 nums，和一个目标值 target。

请你找出并返回满足下述全部条件且不重复的四元组 [nums[a], nums[b], nums[c], nums[d]]：

1.  $0 \leq a, b, c, d < n$
2. a、b、c 和 d 互不相同
3.  $\text{nums}[a] + \text{nums}[b] + \text{nums}[c] + \text{nums}[d] == \text{target}$

示例:

输入: nums = [1, 0, -1, 0, -2, 2], target = 0

输出: [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]

输入: `nums = [2, 2, 2, 2, 2]`, `target = 8`

输出: `[[2, 2, 2, 2]]`

解题思路:

1. 首先对数组进行排序，这样可以方便使用双指针法，并且便于去重。
2. 使用两层循环固定前两个数，然后在剩余的数组中使用双指针法寻找另外两个数，使得这四个数的和等于目标值。
3. 使用双指针法：左指针指向第二个固定数的下一个位置，右指针指向数组末尾。
  - 如果四数之和等于目标值，则找到一个解，同时移动左右指针。
  - 如果四数之和小于目标值，则左指针右移。
  - 如果四数之和大于目标值，则右指针左移。
4. 注意去重处理：
  - 第一层循环固定的第一个数如果相同则跳过
  - 第二层循环固定的第二个数如果相同则跳过
  - 找到解后，左右指针需要跳过相同的数

时间复杂度:  $O(n^3)$  - 两层循环  $O(n^2)$ ，内层双指针  $O(n)$

空间复杂度:  $O(1)$  - 不考虑返回结果的空间

是否最优解: 是 - 基于比较的算法下界为  $O(n^3)$ ，本算法已达到最优

相关题目:

1. LeetCode 1 - 两数之和（无序数组，使用哈希表）
2. LeetCode 167 - 两数之和 II - 输入有序数组（排序+双指针）
3. LeetCode 15 - 三数之和（排序+双指针）
4. LeetCode 18 - 四数之和（当前题目）

工程化考虑:

1. 输入验证: 检查数组是否为空或长度小于 4

2. 异常处理: 处理各种边界情况

3. 边界条件: 处理全为相同元素的情况

语言特性差异:

Java: 使用 `ArrayList` 存储结果，需要导入相关包

C++: 可使用 `vector` 存储结果

Python: 可使用列表存储结果

极端输入场景:

1. 数组全为相同元素
2. 数组长度小于 4
3. 数组包含大量重复元素

与机器学习等领域的联系:

- 在特征选择中，可能需要找到四个特征的组合满足特定条件
- 在推荐系统中，可能需要找到四个物品的组合满足用户偏好

"""

```
def four_sum(nums, target):
 """
 查找所有和为目标值的不重复四元组

 :param nums: 整数数组
 :param target: 目标值
 :return: 所有和为目标值的不重复四元组列表
 """

 # 结果列表
 result = []

 # 边界条件检查
 if not nums or len(nums) < 4:
 return result

 # 对数组进行排序，时间复杂度 O(n log n)
 nums.sort()

 n = len(nums)

 # 遍历数组，固定第一个数
 for i in range(n - 3):
 # 跳过重复元素，避免出现重复解
 if i > 0 and nums[i] == nums[i - 1]:
 continue

 # 遍历数组，固定第二个数
 for j in range(i + 1, n - 2):
 # 跳过重复元素，避免出现重复解
 if j > i + 1 and nums[j] == nums[j - 1]:
 continue

 # 使用双指针在剩余数组中寻找另外两个数
 left = j + 1
 right = n - 1

 while left < right:
 sum_val = nums[i] + nums[j] + nums[left] + nums[right]
```

```

 if sum_val == target:
 # 找到一个解
 result.append([nums[i], nums[j], nums[left], nums[right]])

 # 跳过重复元素
 while left < right and nums[left] == nums[left + 1]:
 left += 1
 while left < right and nums[right] == nums[right - 1]:
 right -= 1

 # 移动指针继续寻找
 left += 1
 right -= 1
 elif sum_val < target:
 # 和小于目标值，左指针右移
 left += 1
 else:
 # 和大于目标值，右指针左移
 right -= 1

return result

```

```

def test():
 """测试函数"""
 # 测试用例 1: [1, 0, -1, 0, -2, 2], target = 0 -> [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
 nums1 = [1, 0, -1, 0, -2, 2]
 target1 = 0
 result1 = four_sum(nums1, target1)
 print(f"Test 1: nums={nums1}, target={target1}")
 print(f"Result: {result1}")

 # 测试用例 2: [2, 2, 2, 2], target = 8 -> [[2, 2, 2, 2]]
 nums2 = [2, 2, 2, 2]
 target2 = 8
 result2 = four_sum(nums2, target2)
 print(f"Test 2: nums={nums2}, target={target2}")
 print(f"Result: {result2}")

```

```

主函数
if __name__ == "__main__":

```

```
test()
```

```
=====
```

文件: Code14\_TrappingRainWater.java

```
=====
```

```
package class050;
```

```
/**
```

```
* 接雨水
```

```
*
```

```
* 题目描述:
```

```
* 给定 n 个非负整数表示每个宽度为 1 的柱子的高度图, 计算按此排列的柱子, 下雨之后能接多少雨水。
```

```
*
```

```
* 示例:
```

```
* 输入: height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
```

```
* 输出: 6
```

```
* 解释: 上面是由数组 [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1] 表示的高度图, 在这种情况下, 可以接 6 个单位的雨水
(蓝色部分表示雨水)。
```

```
*
```

```
* 输入: height = [4, 2, 0, 3, 2, 5]
```

```
* 输出: 9
```

```
*
```

```
* 解题思路:
```

```
* 使用双指针法。我们维护两个指针 left 和 right 分别指向数组的两端, 同时维护两个变量 leftMax 和
rightMax
```

```
* 分别记录左边和右边的最大高度。
```

```
* 1. 如果 leftMax < rightMax, 说明左边的最大值是瓶颈, 可以确定 left 位置能接的雨水量为 leftMax -
height[left]。
```

```
* 2. 如果 leftMax >= rightMax, 说明右边的最大值是瓶颈, 可以确定 right 位置能接的雨水量为
rightMax - height[right]。
```

```
* 3. 移动对应指针并更新最大值, 直到两个指针相遇。
```

```
*
```

```
* 时间复杂度: O(n) - 只需要遍历一次数组
```

```
* 空间复杂度: O(1) - 只使用了常数级别的额外空间
```

```
* 是否最优解: 是 - 基于比较的算法下界为 O(n), 本算法已达到最优
```

```
*
```

```
* 相关题目:
```

```
* 1. LeetCode 42 - 接雨水 (当前题目)
```

```
* 2. LeetCode 11 - 盛最多水的容器
```

```
* 3. LeetCode 407 - 接雨水 II (二维版本)
```

```
*
```

```
* 工程化考虑:
```

- \* 1. 输入验证：检查数组是否为空
- \* 2. 异常处理：处理各种边界情况
- \* 3. 边界条件：处理数组长度小于 3 的情况
- \*
- \* 语言特性差异：
- \* Java：使用数组索引访问
- \* C++：可使用 vector
- \* Python：可使用列表
- \*
- \* 极端输入场景：
- \* 1. 数组长度小于 3
- \* 2. 数组全为 0
- \* 3. 数组呈递增或递减趋势
- \* 4. 数组呈 V 字形或倒 V 字形
- \*
- \* 与机器学习等领域的联系：
- \* 1. 在图像处理中，类似的问题可以用于计算图像中特定区域的特征
- \* 2. 在地理信息系统中，可以用于计算地形的积水区域

\*/

```
public class Code14_TrappingRainWater {
```

```
 /**
 * 使用双指针法计算能接多少雨水
 *
 * @param height 柱子的高度数组
 * @return 能接的雨水总量
 */
```

```
 public static int trap(int[] height) {
 // 边界条件检查
 if (height == null || height.length < 3) {
 return 0;
 }
```

```
 int left = 0; // 左指针
 int right = height.length - 1; // 右指针
 int leftMax = 0; // 左边最大高度
 int rightMax = 0; // 右边最大高度
 int result = 0; // 雨水总量
```

```
 // 当左指针小于右指针时继续循环
 while (left < right) {
 // 更新左边最大高度
 leftMax = Math.max(leftMax, height[left]);
 // 计算当前高度能接多少雨水
 result += leftMax - height[left];
 // 移动左指针
 left++;
 }
 }
}
```

```

// 更新右边最大高度
rightMax = Math.max(rightMax, height[right]);

// 如果左边最大高度小于右边最大高度
if (leftMax < rightMax) {
 // 左边是瓶颈，可以确定 left 位置能接的雨水量
 result += leftMax - height[left];
 left++; // 移动左指针
} else {
 // 右边是瓶颈，可以确定 right 位置能接的雨水量
 result += rightMax - height[right];
 right--; // 移动右指针
}

return result;
}

/***
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: [0,1,0,2,1,0,1,3,2,1,2,1] -> 6
 int[] height1 = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
 int result1 = trap(height1);
 System.out.println("Test 1: height=" + java.util.Arrays.toString(height1));
 System.out.println("Result: " + result1);

 // 测试用例 2: [4,2,0,3,2,5] -> 9
 int[] height2 = {4, 2, 0, 3, 2, 5};
 int result2 = trap(height2);
 System.out.println("Test 2: height=" + java.util.Arrays.toString(height2));
 System.out.println("Result: " + result2);
}
}
=====

文件: Code14_TrappingRainWater.py
=====

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

```

文件: Code14\_TrappingRainWater.py

```
=====

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

```

"""

## 接雨水

### 题目描述:

给定  $n$  个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

### 示例:

输入: height = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

输出: 6

解释: 上面是由数组 [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

输入: height = [4, 2, 0, 3, 2, 5]

输出: 9

### 解题思路:

使用双指针法。我们维护两个指针  $left$  和  $right$  分别指向数组的两端，同时维护两个变量  $leftMax$  和  $rightMax$

分别记录左边和右边的最大高度。

1. 如果  $leftMax < rightMax$ ，说明左边的最大值是瓶颈，可以确定  $left$  位置能接的雨水量为  $leftMax - height[left]$ 。
2. 如果  $leftMax \geq rightMax$ ，说明右边的最大值是瓶颈，可以确定  $right$  位置能接的雨水量为  $rightMax - height[right]$ 。
3. 移动对应指针并更新最大值，直到两个指针相遇。

时间复杂度:  $O(n)$  – 只需要遍历一次数组

空间复杂度:  $O(1)$  – 只使用了常数级别的额外空间

是否最优解: 是 – 基于比较的算法下界为  $O(n)$ ，本算法已达到最优

### 相关题目:

1. LeetCode 42 – 接雨水（当前题目）
2. LeetCode 11 – 盛最多水的容器
3. LeetCode 407 – 接雨水 II（二维版本）

### 工程化考虑:

1. 输入验证: 检查数组是否为空
2. 异常处理: 处理各种边界情况
3. 边界条件: 处理数组长度小于 3 的情况

### 语言特性差异:

Java: 使用数组索引访问

C++: 可使用 vector

Python: 可使用列表

极端输入场景：

1. 数组长度小于 3
2. 数组全为 0
3. 数组呈递增或递减趋势
4. 数组呈 V 字形或倒 V 字形

与机器学习等领域的联系：

1. 在图像处理中，类似的问题可以用于计算图像中特定区域的特征
2. 在地理信息系统中，可以用于计算地形的积水区域

"""

```
def trap(height):
```

```
 """
```

```
 使用双指针法计算能接多少雨水
```

```
 :param height: 柱子的高度数组
```

```
 :return: 能接的雨水总量
```

```
 """
```

```
边界条件检查
```

```
if not height or len(height) < 3:
```

```
 return 0
```

```
left = 0 # 左指针
```

```
right = len(height) - 1 # 右指针
```

```
left_max = 0 # 左边最大高度
```

```
right_max = 0 # 右边最大高度
```

```
result = 0 # 雨水总量
```

```
当左指针小于右指针时继续循环
```

```
while left < right:
```

```
 # 更新左边最大高度
```

```
 left_max = max(left_max, height[left])
```

```
 # 更新右边最大高度
```

```
 right_max = max(right_max, height[right])
```

```
 # 如果左边最大高度小于右边最大高度
```

```
 if left_max < right_max:
```

```
 # 左边是瓶颈，可以确定 left 位置能接的雨水量
```

```
 result += left_max - height[left]
```

```
 left += 1 # 移动左指针
```

```
 else:
```

```

右边是瓶颈，可以确定 right 位置能接的雨水量
result += right_max - height[right]
right -= 1 # 移动右指针

return result

def test():
 """测试函数"""
 # 测试用例 1: [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1] -> 6
 height1 = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
 result1 = trap(height1)
 print(f"Test 1: height={height1}")
 print(f"Result: {result1}")

 # 测试用例 2: [4, 2, 0, 3, 2, 5] -> 9
 height2 = [4, 2, 0, 3, 2, 5]
 result2 = trap(height2)
 print(f"Test 2: height={height2}")
 print(f"Result: {result2}")

主函数
if __name__ == "__main__":
 test()

```

=====

文件: Code15\_ContainerWithMostWater.java

=====

```

package class050;

/**
 * 盛最多水的容器
 *
 * 题目描述:
 * 给定一个长度为 n 的整数数组 height。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i])。
 * 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。
 * 返回容器可以储存的最大水量。
 * 说明：你不能倾斜容器。
 *
 * 示例：

```

```
* 输入: [1, 8, 6, 2, 5, 4, 8, 3, 7]
* 输出: 49
* 解释: 图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下, 容器能够容纳水的最大值为 49。
*
* 输入: height = [1, 1]
* 输出: 1
*
* 解题思路:
* 使用双指针法。左指针指向数组开始, 右指针指向数组末尾。
* 容器的容量由较短的那条线决定, 所以每次移动较短的那条线的指针, 尝试寻找更长的线来增大容量。
* 这样可以在 O(n) 时间内找到最大容量。
*
* 时间复杂度: O(n) - 双指针最多遍历一次数组
* 空间复杂度: O(1) - 只使用了常数级别的额外空间
* 是否最优解: 是 - 基于比较的算法下界为 O(n), 本算法已达到最优
*
* 相关题目:
* 1. LeetCode 11 - 盛最多水的容器 (当前题目)
* 2. LeetCode 42 - 接雨水 (双指针)
* 3. LeetCode 84 - 柱状图中最大的矩形 (单调栈)
*
* 工程化考虑:
* 1. 输入验证: 检查数组是否为空或长度小于 2
* 2. 异常处理: 处理各种边界情况
* 3. 边界条件: 处理数组长度为 2 的情况
*
* 语言特性差异:
* Java: 使用 Math.min 和 Math.max 函数
* C++: 可使用 std::min 和 std::max 函数
* Python: 可使用 min 和 max 函数
*
* 极端输入场景:
* 1. 数组长度为 2
* 2. 数组中所有元素相等
* 3. 数组呈递增或递减趋势
* 4. 最大值在数组中间
*
* 与机器学习等领域的联系:
* 1. 在优化问题中, 可能需要找到两个参数的最优组合
* 2. 在特征工程中, 可能需要找到两个特征的最佳配对
*/
public class Code15_ContainerWithMostWater {
```

```
/**
 * 计算盛最多水的容器的容量
 *
 * @param height 整数数组，表示每条垂线的高度
 * @return 容器可以储存的最大水量
 */

public static int maxArea(int[] height) {
 // 边界条件检查
 if (height == null || height.length < 2) {
 return 0;
 }

 int maxWater = 0;
 int left = 0; // 左指针
 int right = height.length - 1; // 右指针

 // 当左指针小于右指针时继续循环
 while (left < right) {
 // 计算当前容器的容量
 // 容量由较短的那条线决定，乘以两条线之间的距离
 int currentWater = Math.min(height[left], height[right]) * (right - left);

 // 更新最大容量
 maxWater = Math.max(maxWater, currentWater);

 // 移动较短的那条线的指针，尝试寻找更长的线来增大容量
 if (height[left] <= height[right]) {
 left++;
 } else {
 right--;
 }
 }

 return maxWater;
}

/**
 * 测试方法
 */

public static void main(String[] args) {
 // 测试用例 1: [1, 8, 6, 2, 5, 4, 8, 3, 7] -> 49
 int[] height1 = {1, 8, 6, 2, 5, 4, 8, 3, 7};
 int result1 = maxArea(height1);
```

```
System.out.println("Test 1: height=" + java.util.Arrays.toString(height1));
System.out.println("Result: " + result1);

// 测试用例 2: [1,1] -> 1
int[] height2 = {1, 1};
int result2 = maxArea(height2);
System.out.println("Test 2: height=" + java.util.Arrays.toString(height2));
System.out.println("Result: " + result2);

}

}

=====
```

文件: Code15\_ContainerWithMostWater.py

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

```
"""\n
```

盛最多水的容器

题目描述:

给定一个长度为  $n$  的整数数组  $height$ 。有  $n$  条垂线，第  $i$  条线的两个端点是  $(i, 0)$  和  $(i, height[i])$ 。

找出其中的两条线，使得它们与  $x$  轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例：

输入： [1, 8, 6, 2, 5, 4, 8, 3, 7]

输出： 49

解释：图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水的最大值为 49。

输入： height = [1, 1]

输出： 1

解题思路：

使用双指针法。左指针指向数组开始，右指针指向数组末尾。

容器的容量由较短的那条线决定，所以每次移动较短的那条线的指针，尝试寻找更长的线来增大容量。

这样可以在  $O(n)$  时间内找到最大容量。

时间复杂度： $O(n)$  – 双指针最多遍历一次数组

空间复杂度： $O(1)$  – 只使用了常数级别的额外空间

是否最优解：是 - 基于比较的算法下界为  $O(n)$ ，本算法已达到最优

相关题目：

1. LeetCode 11 - 盛最多水的容器（当前题目）
2. LeetCode 42 - 接雨水（双指针）
3. LeetCode 84 - 柱状图中最大的矩形（单调栈）

工程化考虑：

1. 输入验证：检查数组是否为空或长度小于 2
2. 异常处理：处理各种边界情况
3. 边界条件：处理数组长度为 2 的情况

语言特性差异：

Java：使用 `Math.min` 和 `Math.max` 函数

C++：可使用 `std::min` 和 `std::max` 函数

Python：可使用 `min` 和 `max` 函数

极端输入场景：

1. 数组长度为 2
2. 数组中所有元素相等
3. 数组呈递增或递减趋势
4. 最大值在数组中间

与机器学习等领域的联系：

1. 在优化问题中，可能需要找到两个参数的最优组合
2. 在特征工程中，可能需要找到两个特征的最佳配对

"""

```
def max_area(height):
 """
 计算盛最多水的容器的容量

 :param height: 整数数组，表示每条垂线的高度
 :return: 容器可以储存的最大水量
 """

 # 边界条件检查
 if not height or len(height) < 2:
 return 0

 max_water = 0
 left = 0 # 左指针
 right = len(height) - 1 # 右指针
```

```
当左指针小于右指针时继续循环
while left < right:
 # 计算当前容器的容量
 # 容量由较短的那条线决定，乘以两条线之间的距离
 current_water = min(height[left], height[right]) * (right - left)

 # 更新最大容量
 max_water = max(max_water, current_water)

 # 移动较短的那条线的指针，尝试寻找更长的线来增大容量
 if height[left] <= height[right]:
 left += 1
 else:
 right -= 1

return max_water

def test():
 """测试函数"""
 # 测试用例 1: [1, 8, 6, 2, 5, 4, 8, 3, 7] -> 49
 height1 = [1, 8, 6, 2, 5, 4, 8, 3, 7]
 result1 = max_area(height1)
 print(f"Test 1: height={height1}")
 print(f"Result: {result1}")

 # 测试用例 2: [1, 1] -> 1
 height2 = [1, 1]
 result2 = max_area(height2)
 print(f"Test 2: height={height2}")
 print(f"Result: {result2}")

主函数
if __name__ == "__main__":
 test()
```

=====

文件: Code20\_JumpGameII.cpp

=====

```
#include <iostream>
```

```
#include <vector>
#include <queue>
#include <climits>
#include <chrono>
#include <random>

/***
 * LeetCode 45. 跳跃游戏 II (Jump Game II)
 *
 * 题目描述:
 * 给定一个非负整数数组 nums，你最初位于数组的第一个位置。
 * 数组中的每个元素代表你在该位置可以跳跃的最大长度。
 * 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
 * 假设你总是可以到达数组的最后一个位置。
 *
 * 示例 1:
 * 输入: nums = [2, 3, 1, 1, 4]
 * 输出: 2
 * 解释: 跳到最后一个位置的最小跳跃数是 2。
 * 从下标为 0 的位置跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
 *
 * 示例 2:
 * 输入: nums = [2, 3, 0, 1, 4]
 * 输出: 2
 *
 * 提示:
 * 1 <= nums.length <= 10^4
 * 0 <= nums[i] <= 1000
 * 题目保证可以到达 nums[n-1]
 *
 * 题目链接: https://leetcode.com/problems/jump-game-ii/
 *
 * 解题思路:
 * 这道题可以使用贪心算法来解决。我们的目标是用最少的跳跃次数到达数组的最后一个位置。
 *
 * 贪心策略: 在每一步中，我们都选择能够到达的最远位置的下一步。
 *
 * 具体来说，我们维护三个变量:
 * 1. currentEnd: 当前能够到达的最远边界
 * 2. currentFarthest: 在遍历过程中找到的从当前位置可以到达的最远位置
 * 3. jumps: 记录跳跃次数
 *
 * 当我们遍历数组时，每当我们到达 currentEnd，就意味着我们需要进行一次跳跃，此时将 jumps 加 1，并将
```

currentEnd 更新为 currentFarthest。

```
*
* 时间复杂度: O(n)，其中 n 是数组的长度。我们只需要遍历数组一次。
* 空间复杂度: O(1)，只使用了常数级别的额外空间。

* 此外，我们还提供两种其他解法：
* 1. 动态规划解法：时间复杂度 O(n^2)，空间复杂度 O(n)
* 2. BFS 解法：将问题视为图中的最短路径问题，时间复杂度 O(n^2)，空间复杂度 O(n)
*/
```

```
class Solution {
public:
 /**
 * 解法一：贪心算法（最优解）
 *
 * @param nums 非负整数数组
 * @return 到达最后一个位置的最小跳跃次数
 */
 int jumpGreedy(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() <= 1) {
 return 0; // 如果数组为空或只有一个元素，不需要跳跃
 }

 int jumps = 0; // 跳跃次数
 int currentEnd = 0; // 当前能到达的最远边界
 int currentFarthest = 0; // 在遍历过程中找到的最远可达位置

 // 遍历数组，但不需要遍历到最后一个元素，因为一旦 currentFarthest >= nums.size() - 1，就已经可以到达终点
 for (int i = 0; i < nums.size() - 1; i++) {
 // 更新从当前位置可达的最远位置
 currentFarthest = std::max(currentFarthest, i + nums[i]);

 // 当到达当前边界时，必须进行一次跳跃
 if (i == currentEnd) {
 jumps++;
 currentEnd = currentFarthest; // 更新边界为新的最远可达位置

 // 如果已经可以到达或超过最后一个位置，可以提前结束
 if (currentEnd >= static_cast<int>(nums.size() - 1)) {
 break;
 }
 }
 }
 }
};
```

```

 }

 }

 return jumps;
}

/***
 * 解法二：动态规划
 *
 * @param nums 非负整数数组
 * @return 到达最后一个位置的最小跳跃次数
 */
int jumpDynamicProgramming(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() <= 1) {
 return 0; // 如果数组为空或只有一个元素，不需要跳跃
 }

 int n = nums.size();
 // dp[i] 表示到达位置 i 所需的最小跳跃次数
 std::vector<int> dp(n, INT_MAX);
 dp[0] = 0; // 起始位置不需要跳跃

 // 计算每个位置的最小跳跃次数
 for (int i = 0; i < n; i++) {
 // 如果当前位置无法到达，跳过
 if (dp[i] == INT_MAX) {
 continue;
 }

 // 从当前位置可以跳跃到的所有位置
 int maxJump = std::min(i + nums[i], n - 1); // 确保不超过数组边界
 for (int j = i + 1; j <= maxJump; j++) {
 // 更新到达位置 j 的最小跳跃次数
 if (dp[j] > dp[i] + 1) {
 dp[j] = dp[i] + 1;
 }
 }
 }

 // 如果已经到达最后一个位置，可以提前结束
 if (j == n - 1) {
 return dp[j];
 }
}

```

```

}

return dp[n - 1];
}

/***
 * 解法三：BFS
 * 将问题视为图中的最短路径问题，每个位置是一个节点，从位置 i 可以到 i+1, i+2, ..., i+nums[i]
 *
 * @param nums 非负整数数组
 * @return 到达最后一个位置的最小跳跃次数
 */
int jumpBFS(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() <= 1) {
 return 0; // 如果数组为空或只有一个元素，不需要跳跃
 }

 int n = nums.size();
 std::vector<bool> visited(n, false); // 记录已经访问过的位置
 std::queue<int> queue; // BFS 队列

 // 初始化队列，起始位置是 0，跳跃次数是 0
 queue.push(0);
 visited[0] = true;
 int jumps = 0;

 while (!queue.empty()) {
 int size = queue.size(); // 当前层的节点数

 // 处理当前层的所有节点
 for (int i = 0; i < size; i++) {
 int current = queue.front();
 queue.pop();

 // 如果到达最后一个位置，返回跳跃次数
 if (current == n - 1) {
 return jumps;
 }

 // 将从当前位置可以到达的所有位置加入队列
 int maxJump = std::min(current + nums[current], n - 1);
 for (int j = maxJump; j > current; j--) { // 反向遍历，优先考虑跳得更远的位置

```

```

 if (!visited[j]) {
 visited[j] = true;
 queue.push(j);

 // 如果下一层已经可以到达最后一个位置，可以提前结束当前层的处理
 if (j == n - 1) {
 return jumps + 1;
 }
 }
 }

 jumps++; // 处理完一层，跳跃次数加 1
}

return -1; // 根据题目描述，一定可以到达最后一个位置，所以不会执行到这里
}
};

/***
 * 打印数组
 */
void printArray(const std::vector<int>& arr) {
 std::cout << "[";
 for (size_t i = 0; i < arr.size(); i++) {
 std::cout << arr[i];
 if (i < arr.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 性能测试
 */
void performanceTest(const std::vector<int>& nums, Solution& solution) {
 // 测试贪心算法
 auto startTime = std::chrono::high_resolution_clock::now();
 int result1 = solution.jumpGreedy(nums);
 auto endTime = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
 std::cout << "贪心算法结果: " << result1 << std::endl;
}

```

```

std::cout << "贪心算法耗时：" << duration.count() << "ms" << std::endl;

// 测试动态规划
startTime = std::chrono::high_resolution_clock::now();
int result2 = solution.jumpDynamicProgramming(nums);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "动态规划结果：" << result2 << std::endl;
std::cout << "动态规划耗时：" << duration.count() << "ms" << std::endl;

// 测试 BFS
startTime = std::chrono::high_resolution_clock::now();
int result3 = solution.jumpBFS(nums);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "BFS 结果：" << result3 << std::endl;
std::cout << "BFS 耗时：" << duration.count() << "ms" << std::endl;
}

/***
 * 生成测试用例
 */
std::vector<int> generateTestCase(int n, bool worstCase) {
 std::vector<int> nums(n);

 if (worstCase) {
 // 最坏情况：每次只能跳 1 步
 std::fill(nums.begin(), nums.end(), 1);
 } else {
 // 随机情况：生成 1 到 5 之间的随机数
 std::random_device rd;
 std::mt19937 gen(rd());
 std::uniform_int_distribution<> distrib(1, 5);

 for (int i = 0; i < n - 1; i++) {
 nums[i] = distrib(gen);
 }
 nums[n - 1] = 0; // 最后一个元素不影响
 }

 return nums;
}

```

```
int main() {
 Solution solution;

 // 测试用例 1
 std::vector<int> nums1 = {2, 3, 1, 1, 4};
 std::cout << "测试用例 1:" << std::endl;
 std::cout << "nums = ";
 printArray(nums1);
 std::cout << "贪心算法结果: " << solution.jumpGreedy(nums1) << std::endl; // 预期输出: 2
 std::cout << "动态规划结果: " << solution.jumpDynamicProgramming(nums1) << std::endl; // 预期
输出: 2
 std::cout << "BFS 结果: " << solution.jumpBFS(nums1) << std::endl; // 预期输出: 2
 std::cout << std::endl;

 // 测试用例 2
 std::vector<int> nums2 = {2, 3, 0, 1, 4};
 std::cout << "测试用例 2:" << std::endl;
 std::cout << "nums = ";
 printArray(nums2);
 std::cout << "贪心算法结果: " << solution.jumpGreedy(nums2) << std::endl; // 预期输出: 2
 std::cout << "动态规划结果: " << solution.jumpDynamicProgramming(nums2) << std::endl; // 预期
输出: 2
 std::cout << "BFS 结果: " << solution.jumpBFS(nums2) << std::endl; // 预期输出: 2
 std::cout << std::endl;

 // 测试用例 3 - 边界情况: 只有一个元素
 std::vector<int> nums3 = {0};
 std::cout << "测试用例 3 (单元素数组) :" << std::endl;
 std::cout << "nums = ";
 printArray(nums3);
 std::cout << "贪心算法结果: " << solution.jumpGreedy(nums3) << std::endl; // 预期输出: 0
 std::cout << "动态规划结果: " << solution.jumpDynamicProgramming(nums3) << std::endl; // 预期
输出: 0
 std::cout << "BFS 结果: " << solution.jumpBFS(nums3) << std::endl; // 预期输出: 0
 std::cout << std::endl;

 // 测试用例 4 - 边界情况: 每次只能跳 1 步
 std::vector<int> nums4 = {1, 1, 1, 1, 1};
 std::cout << "测试用例 4 (每次只能跳 1 步) :" << std::endl;
 std::cout << "nums = ";
 printArray(nums4);
 std::cout << "贪心算法结果: " << solution.jumpGreedy(nums4) << std::endl; // 预期输出: 4
 std::cout << "动态规划结果: " << solution.jumpDynamicProgramming(nums4) << std::endl; // 预期
```

输出: 4

```
std::cout << "BFS 结果: " << solution.jumpBFS(nums4) << std::endl; // 预期输出: 4
std::cout << std::endl;
```

// 测试用例 5 - 边界情况: 可以一次跳到终点

```
std::vector<int> nums5 = {10, 1, 1, 1, 1};
std::cout << "测试用例 5 (可以一次跳到终点): " << std::endl;
std::cout << "nums = ";
printArray(nums5);
std::cout << "贪心算法结果: " << solution.jumpGreedy(nums5) << std::endl; // 预期输出: 1
std::cout << "动态规划结果: " << solution.jumpDynamicProgramming(nums5) << std::endl; // 预期输出: 1
```

输出: 1

```
std::cout << "BFS 结果: " << solution.jumpBFS(nums5) << std::endl; // 预期输出: 1
std::cout << std::endl;
```

// 性能测试 - 小规模数组

```
std::cout << "小规模数组性能测试: " << std::endl;
std::vector<int> smallArray = generateTestCase(100, false);
performanceTest(smallArray, solution);
std::cout << std::endl;
```

// 性能测试 - 大规模数组 - 只测试贪心算法, 因为其他算法在大规模数组上会很慢

```
std::cout << "大规模数组性能测试 (只测试贪心算法): " << std::endl;
std::vector<int> largeArray = generateTestCase(10000, false);
auto startTime = std::chrono::high_resolution_clock::now();
int result = solution.jumpGreedy(largeArray);
auto endTime = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "贪心算法结果: " << result << std::endl;
std::cout << "贪心算法耗时: " << duration.count() << "ms" << std::endl;
```

// 最坏情况性能测试

```
std::cout << "\n最坏情况性能测试: " << std::endl;
std::vector<int> worstCaseArray = generateTestCase(1000, true); // 小规模的最坏情况, 否则动态规划和 BFS 会超时
performanceTest(worstCaseArray, solution);
```

```
return 0;
```

```
}
```

```
=====
import java.util.*;
import java.util.stream.Collectors;

/***
 * LeetCode 45. 跳跃游戏 II (Jump Game II)
 *
 * 题目描述:
 * 给定一个非负整数数组 nums，你最初位于数组的第一个位置。
 * 数组中的每个元素代表你在该位置可以跳跃的最大长度。
 * 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
 * 假设你总是可以到达数组的最后一个位置。
 *
 * 示例 1:
 * 输入: nums = [2, 3, 1, 1, 4]
 * 输出: 2
 * 解释: 跳到最后一个位置的最小跳跃数是 2。
 * 从下标为 0 的位置跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。
 *
 * 示例 2:
 * 输入: nums = [2, 3, 0, 1, 4]
 * 输出: 2
 *
 * 提示:
 * 1 <= nums.length <= 10^4
 * 0 <= nums[i] <= 1000
 * 题目保证可以到达 nums[n-1]
 *
 * 题目链接: https://leetcode.com/problems/jump-game-ii/
 *
 * 解题思路:
 * 这道题可以使用贪心算法来解决。我们的目标是用最少的跳跃次数到达数组的最后一个位置。
 *
 * 贪心策略: 在每一步中，我们都选择能够到达的最远位置的下一步。
 *
 * 具体来说，我们维护三个变量:
 * 1. currentEnd: 当前能够到达的最远边界
 * 2. currentFarthest: 在遍历过程中找到的从当前位置可以到达的最远位置
 * 3. jumps: 记录跳跃次数
 *
 * 当我们遍历数组时，每当我们到达 currentEnd，就意味着我们需要进行一次跳跃，此时将 jumps 加 1，并将 currentEnd 更新为 currentFarthest。
 *
 */
```

- \* 时间复杂度:  $O(n)$ , 其中  $n$  是数组的长度。我们只需要遍历数组一次。
- \* 空间复杂度:  $O(1)$ , 只使用了常数级别的额外空间。
- \*
- \* 此外, 我们还提供两种其他解法:
- \* 1. 动态规划解法: 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n)$
- \* 2. BFS 解法: 将问题视为图中的最短路径问题, 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n)$

\*/

```

public class Code20_JumpGameII {

 /**
 * 解法一: 贪心算法 (最优解)
 *
 * @param nums 非负整数数组
 * @return 到达最后一个位置的最小跳跃次数
 */
 public static int jumpGreedy(int[] nums) {
 // 参数校验
 if (nums == null || nums.length <= 1) {
 return 0; // 如果数组为空或只有一个元素, 不需要跳跃
 }

 int jumps = 0; // 跳跃次数
 int currentEnd = 0; // 当前能到达的最远边界
 int currentFarthest = 0; // 在遍历过程中找到的最远可达位置

 // 遍历数组, 但不需要遍历到最后一个元素, 因为一旦 currentFarthest >= nums.length - 1, 就已经可以到达终点
 for (int i = 0; i < nums.length - 1; i++) {
 // 更新从当前位置可达的最远位置
 currentFarthest = Math.max(currentFarthest, i + nums[i]);

 // 当到达当前边界时, 必须进行一次跳跃
 if (i == currentEnd) {
 jumps++;
 currentEnd = currentFarthest; // 更新边界为新的最远可达位置

 // 如果已经可以到达或超过最后一个位置, 可以提前结束
 if (currentEnd >= nums.length - 1) {
 break;
 }
 }
 }
 }
}

```

```
 return jumps;
 }

/***
 * 解法二：动态规划
 *
 * @param nums 非负整数数组
 * @return 到达最后一个位置的最小跳跃次数
 */
public static int jumpDynamicProgramming(int[] nums) {
 // 参数校验
 if (nums == null || nums.length <= 1) {
 return 0; // 如果数组为空或只有一个元素，不需要跳跃
 }

 int n = nums.length;
 // dp[i]表示到达位置 i 所需的最小跳跃次数
 int[] dp = new int[n];

 // 初始化所有位置为一个很大的值，表示暂时无法到达
 Arrays.fill(dp, Integer.MAX_VALUE);
 dp[0] = 0; // 起始位置不需要跳跃

 // 计算每个位置的最小跳跃次数
 for (int i = 0; i < n; i++) {
 // 如果当前位置无法到达，跳过
 if (dp[i] == Integer.MAX_VALUE) {
 continue;
 }

 // 从当前位置可以跳跃到的所有位置
 int maxJump = Math.min(i + nums[i], n - 1); // 确保不超过数组边界
 for (int j = i + 1; j <= maxJump; j++) {
 // 更新到达位置 j 的最小跳跃次数
 if (dp[j] > dp[i] + 1) {
 dp[j] = dp[i] + 1;

 // 如果已经到达最后一个位置，可以提前结束
 if (j == n - 1) {
 return dp[j];
 }
 }
 }
 }
}
```

```

 }

 }

 return dp[n - 1];
}

/***
 * 解法三：BFS
 * 将问题视为图中的最短路径问题，每个位置是一个节点，从位置 i 可以到 i+1, i+2, ..., i+nums[i]
 *
 * @param nums 非负整数数组
 * @return 到达最后一个位置的最小跳跃次数
 */
public static int jumpBFS(int[] nums) {
 // 参数校验
 if (nums == null || nums.length <= 1) {
 return 0; // 如果数组为空或只有一个元素，不需要跳跃
 }

 int n = nums.length;
 boolean[] visited = new boolean[n]; // 记录已经访问过的位置
 Queue<Integer> queue = new LinkedList<>(); // BFS 队列

 // 初始化队列，起始位置是 0，跳跃次数是 0
 queue.offer(0);
 visited[0] = true;
 int jumps = 0;

 while (!queue.isEmpty()) {
 int size = queue.size(); // 当前层的节点数

 // 处理当前层的所有节点
 for (int i = 0; i < size; i++) {
 int current = queue.poll();

 // 如果到达最后一个位置，返回跳跃次数
 if (current == n - 1) {
 return jumps;
 }

 // 将从当前位置可以到达的所有位置加入队列
 int maxJump = Math.min(current + nums[current], n - 1);
 for (int j = maxJump; j > current; j--) { // 反向遍历，优先考虑跳得更远的位置
 if (!visited[j]) {
 queue.offer(j);
 visited[j] = true;
 }
 }
 }
 jumps++;
 }
}

```

```
 if (!visited[j]) {
 visited[j] = true;
 queue.offer(j);

 // 如果下一层已经可以到达最后一个位置，可以提前结束当前层的处理
 if (j == n - 1) {
 return jumps + 1;
 }
 }
 }

 jumps++; // 处理完一层，跳跃次数加 1
}

return -1; // 根据题目描述，一定可以到达最后一个位置，所以不会执行到这里
}

/***
 * 打印数组
 */
public static void printArray(int[] arr) {
 System.out.println(Arrays.stream(arr)
 .mapToObj(String::valueOf)
 .collect(Collectors.joining(", ", "[", "]")));
}

/***
 * 性能测试
 */
public static void performanceTest(int[] nums) {
 // 测试贪心算法
 long startTime = System.currentTimeMillis();
 int result1 = jumpGreedy(nums);
 long endTime = System.currentTimeMillis();
 System.out.println("贪心算法结果: " + result1);
 System.out.println("贪心算法耗时: " + (endTime - startTime) + "ms");

 // 测试动态规划
 startTime = System.currentTimeMillis();
 int result2 = jumpDynamicProgramming(nums);
 endTime = System.currentTimeMillis();
 System.out.println("动态规划结果: " + result2);
}
```

```

System.out.println("动态规划耗时: " + (endTime - startTime) + "ms");

// 测试 BFS
startTime = System.currentTimeMillis();
int result3 = jumpBFS(nums);
endTime = System.currentTimeMillis();
System.out.println("BFS 结果: " + result3);
System.out.println("BFS 耗时: " + (endTime - startTime) + "ms");
}

/**
 * 生成测试用例
 */
public static int[] generateTestCase(int n, boolean worstCase) {
 int[] nums = new int[n];

 if (worstCase) {
 // 最坏情况: 每次只能跳 1 步
 Arrays.fill(nums, 1);
 } else {
 // 随机情况: 生成 1 到 5 之间的随机数
 Random rand = new Random();
 for (int i = 0; i < n - 1; i++) {
 nums[i] = rand.nextInt(5) + 1; // 1 到 5 之间的随机数
 }
 nums[n - 1] = 0; // 最后一个元素不影响
 }

 return nums;
}

public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {2, 3, 1, 1, 4};
 System.out.println("测试用例 1:");
 System.out.print("nums = ");
 printArray(nums1);
 System.out.println("贪心算法结果: " + jumpGreedy(nums1)); // 预期输出: 2
 System.out.println("动态规划结果: " + jumpDynamicProgramming(nums1)); // 预期输出: 2
 System.out.println("BFS 结果: " + jumpBFS(nums1)); // 预期输出: 2
 System.out.println();

 // 测试用例 2
}

```

```
int[] nums2 = {2, 3, 0, 1, 4};
System.out.println("测试用例 2:");
System.out.print("nums = ");
printArray(nums2);
System.out.println("贪心算法结果: " + jumpGreedy(nums2)); // 预期输出: 2
System.out.println("动态规划结果: " + jumpDynamicProgramming(nums2)); // 预期输出: 2
System.out.println("BFS 结果: " + jumpBFS(nums2)); // 预期输出: 2
System.out.println();

// 测试用例 3 - 边界情况: 只有一个元素
int[] nums3 = {0};
System.out.println("测试用例 3 (单元素数组) :");
System.out.print("nums = ");
printArray(nums3);
System.out.println("贪心算法结果: " + jumpGreedy(nums3)); // 预期输出: 0
System.out.println("动态规划结果: " + jumpDynamicProgramming(nums3)); // 预期输出: 0
System.out.println("BFS 结果: " + jumpBFS(nums3)); // 预期输出: 0
System.out.println();

// 测试用例 4 - 边界情况: 每次只能跳 1 步
int[] nums4 = {1, 1, 1, 1, 1};
System.out.println("测试用例 4 (每次只能跳 1 步) :");
System.out.print("nums = ");
printArray(nums4);
System.out.println("贪心算法结果: " + jumpGreedy(nums4)); // 预期输出: 4
System.out.println("动态规划结果: " + jumpDynamicProgramming(nums4)); // 预期输出: 4
System.out.println("BFS 结果: " + jumpBFS(nums4)); // 预期输出: 4
System.out.println();

// 测试用例 5 - 边界情况: 可以一次跳到终点
int[] nums5 = {10, 1, 1, 1, 1};
System.out.println("测试用例 5 (可以一次跳到终点) :");
System.out.print("nums = ");
printArray(nums5);
System.out.println("贪心算法结果: " + jumpGreedy(nums5)); // 预期输出: 1
System.out.println("动态规划结果: " + jumpDynamicProgramming(nums5)); // 预期输出: 1
System.out.println("BFS 结果: " + jumpBFS(nums5)); // 预期输出: 1
System.out.println();

// 性能测试 - 小规模数组
System.out.println("小规模数组性能测试:");
int[] smallArray = generateTestCase(100, false);
performanceTest(smallArray);
```

```

System.out.println();

// 性能测试 - 大规模数组 - 只测试贪心算法，因为其他算法在大规模数组上会很慢
System.out.println("大规模数组性能测试（只测试贪心算法）:");
int[] largeArray = generateTestCase(10000, false);
long startTime = System.currentTimeMillis();
int result = jumpGreedy(largeArray);
long endTime = System.currentTimeMillis();
System.out.println("贪心算法结果: " + result);
System.out.println("贪心算法耗时: " + (endTime - startTime) + "ms");

// 最坏情况性能测试
System.out.println("\n最坏情况性能测试:");
int[] worstCaseArray = generateTestCase(1000, true); // 小规模的最坏情况，否则动态规划和
BFS 会超时
performanceTest(worstCaseArray);
}
}

```

=====

文件: Code20\_JumpGameII.py

=====

```

import time
import random
from typing import List

"""

```

LeetCode 45. 跳跃游戏 II (Jump Game II)

题目描述:

给定一个非负整数数组 `nums`，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

假设你总是可以到达数组的最后一个位置。

示例 1:

输入: `nums = [2, 3, 1, 1, 4]`

输出: 2

解释: 跳到最后一个位置的最小跳跃数是 2。

从下标为 0 的位置跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

示例 2:

输入: nums = [2, 3, 0, 1, 4]

输出: 2

提示:

1 <= nums.length <=  $10^4$

0 <= nums[i] <= 1000

题目保证可以到达 nums[n-1]

题目链接: <https://leetcode.com/problems/jump-game-ii/>

解题思路:

这道题可以使用贪心算法来解决。我们的目标是用最少的跳跃次数到达数组的最后一个位置。

贪心策略: 在每一步中, 我们都选择能够到达的最远位置的下一步。

具体来说, 我们维护三个变量:

1. current\_end: 当前能够到达的最远边界
2. current\_farthest: 在遍历过程中找到的从当前位置可以到达的最远位置
3. jumps: 记录跳跃次数

当我们遍历数组时, 每当我们到达 current\_end, 就意味着我们需要进行一次跳跃, 此时将 jumps 加 1, 并将 current\_end 更新为 current\_farthest。

时间复杂度:  $O(n)$ , 其中 n 是数组的长度。我们只需要遍历数组一次。

空间复杂度:  $O(1)$ , 只使用了常数级别的额外空间。

此外, 我们还提供两种其他解法:

1. 动态规划解法: 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n)$
2. BFS 解法: 将问题视为图中的最短路径问题, 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n)$

"""

class Solution:

```
def jump_greedy(self, nums: List[int]) -> int:
 """
```

解法一: 贪心算法 (最优解)

Args:

nums: 非负整数数组

Returns:

到达最后一个位置的最小跳跃次数

"""\n

# 参数校验

```

if not nums or len(nums) <= 1:
 return 0 # 如果数组为空或只有一个元素，不需要跳跃

jumps = 0 # 跳跃次数
current_end = 0 # 当前能到达的最远边界
current_farthest = 0 # 在遍历过程中找到的最远可达位置

遍历数组，但不需要遍历到最后一个元素
for i in range(len(nums) - 1):
 # 更新从当前位置可达的最远位置
 current_farthest = max(current_farthest, i + nums[i])

 # 当到达当前边界时，必须进行一次跳跃
 if i == current_end:
 jumps += 1
 current_end = current_farthest # 更新边界为新的最远可达位置

 # 如果已经可以到达或超过最后一个位置，可以提前结束
 if current_end >= len(nums) - 1:
 break

return jumps

```

```

def jump_dynamic_programming(self, nums: List[int]) -> int:
 """

```

解法二：动态规划

Args:

nums: 非负整数数组

Returns:

到达最后一个位置的最小跳跃次数

"""

# 参数校验

```

if not nums or len(nums) <= 1:
 return 0 # 如果数组为空或只有一个元素，不需要跳跃

```

n = len(nums)

# dp[i]表示到达位置 i 所需的最小跳跃次数

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

dp[0] = 0 # 起始位置不需要跳跃

# 计算每个位置的最小跳跃次数

```

for i in range(n):
 # 如果当前位置无法到达，跳过
 if dp[i] == float('inf'):
 continue

 # 从当前位置可以跳跃到的所有位置
 max_jump = min(i + nums[i], n - 1) # 确保不超过数组边界
 for j in range(i + 1, max_jump + 1):
 # 更新到达位置 j 的最小跳跃次数
 if dp[j] > dp[i] + 1:
 dp[j] = dp[i] + 1

 # 如果已经到达最后一个位置，可以提前结束
 if j == n - 1:
 return dp[j]

return dp[n - 1]

```

```

def jump_bfs(self, nums: List[int]) -> int:
 """

```

解法三：BFS

将问题视为图中的最短路径问题，每个位置是一个节点，从位置  $i$  可以到  $i+1, i+2, \dots, i+nums[i]$

Args:

nums: 非负整数数组

Returns:

到达最后一个位置的最小跳跃次数

"""

# 参数校验

```

if not nums or len(nums) <= 1:
 return 0 # 如果数组为空或只有一个元素，不需要跳跃

```

n = len(nums)

visited = [False] \* n # 记录已经访问过的位置

queue = [] # BFS 队列

# 初始化队列，起始位置是 0，跳跃次数是 0

queue.append(0)

visited[0] = True

jumps = 0

while queue:

```

size = len(queue) # 当前层的节点数

处理当前层的所有节点
for _ in range(size):
 current = queue.pop(0)

 # 如果到达最后一个位置，返回跳跃次数
 if current == n - 1:
 return jumps

 # 将从当前位置可以到达的所有位置加入队列
 max_jump = min(current + nums[current], n - 1)
 # 反向遍历，优先考虑跳得更远的位置
 for j in range(max_jump, current, -1):
 if not visited[j]:
 visited[j] = True
 queue.append(j)

 # 如果下一层已经可以到达最后一个位置，可以提前结束当前层的处理
 if j == n - 1:
 return jumps + 1

jumps += 1 # 处理完一层，跳跃次数加1

return -1 # 根据题目描述，一定可以到达最后一个位置，所以不会执行到这里

```

def jump(self, nums: List[int]) -> int:

"""

LeetCode 官方接口的实现（使用贪心算法）

Args:

nums: 非负整数数组

Returns:

到达最后一个位置的最小跳跃次数

"""

return self.jump\_greedy(nums)

"""

打印数组

"""

```

def print_array(arr: List[int]) -> None:
 print(f"[{', '.join(map(str, arr))}]")

```

```
"""
```

## 性能测试

```
"""
```

```
def performance_test(nums: List[int], solution: Solution) -> None:
```

```
 # 测试贪心算法
```

```
 start_time = time.time()
```

```
 result1 = solution.jump_greedy(nums)
```

```
 end_time = time.time()
```

```
 print(f"贪心算法结果: {result1}")
```

```
 print(f"贪心算法耗时: {(end_time - start_time) * 1000:.2f}ms")
```

```
测试动态规划
```

```
 start_time = time.time()
```

```
 result2 = solution.jump_dynamic_programming(nums)
```

```
 end_time = time.time()
```

```
 print(f"动态规划结果: {result2}")
```

```
 print(f"动态规划耗时: {(end_time - start_time) * 1000:.2f}ms")
```

```
测试 BFS
```

```
 start_time = time.time()
```

```
 result3 = solution.jump_bfs(nums)
```

```
 end_time = time.time()
```

```
 print(f"BFS 结果: {result3}")
```

```
 print(f"BFS 耗时: {(end_time - start_time) * 1000:.2f}ms")
```

```
"""
```

## 生成测试用例

```
"""
```

```
def generate_test_case(n: int, worst_case: bool) -> List[int]:
```

```
 nums = [0] * n
```

```
 if worst_case:
```

```
 # 最坏情况: 每次只能跳 1 步
```

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

```
 nums[i] = 1
```

```
 else:
```

```
 # 随机情况: 生成 1 到 5 之间的随机数
```

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

```
 nums[i] = random.randint(1, 5)
```

```
 nums[n - 1] = 0 # 最后一个元素不影响
```

```
return nums
```

```
def main():
 solution = Solution()

 # 测试用例 1
 nums1 = [2, 3, 1, 1, 4]
 print("测试用例 1:")
 print("nums = ", end="")
 print_array(nums1)
 print(f"贪心算法结果: {solution.jump_greedy(nums1)}") # 预期输出: 2
 print(f"动态规划结果: {solution.jump_dynamic_programming(nums1)}") # 预期输出: 2
 print(f"BFS 结果: {solution.jump_bfs(nums1)}") # 预期输出: 2
 print()

 # 测试用例 2
 nums2 = [2, 3, 0, 1, 4]
 print("测试用例 2:")
 print("nums = ", end="")
 print_array(nums2)
 print(f"贪心算法结果: {solution.jump_greedy(nums2)}") # 预期输出: 2
 print(f"动态规划结果: {solution.jump_dynamic_programming(nums2)}") # 预期输出: 2
 print(f"BFS 结果: {solution.jump_bfs(nums2)}") # 预期输出: 2
 print()

 # 测试用例 3 - 边界情况: 只有一个元素
 nums3 = [0]
 print("测试用例 3 (单元素数组) :")
 print("nums = ", end="")
 print_array(nums3)
 print(f"贪心算法结果: {solution.jump_greedy(nums3)}") # 预期输出: 0
 print(f"动态规划结果: {solution.jump_dynamic_programming(nums3)}") # 预期输出: 0
 print(f"BFS 结果: {solution.jump_bfs(nums3)}") # 预期输出: 0
 print()

 # 测试用例 4 - 边界情况: 每次只能跳 1 步
 nums4 = [1, 1, 1, 1, 1]
 print("测试用例 4 (每次只能跳 1 步) :")
 print("nums = ", end="")
 print_array(nums4)
 print(f"贪心算法结果: {solution.jump_greedy(nums4)}") # 预期输出: 4
 print(f"动态规划结果: {solution.jump_dynamic_programming(nums4)}") # 预期输出: 4
 print(f"BFS 结果: {solution.jump_bfs(nums4)}") # 预期输出: 4
 print()
```

```

测试用例 5 - 边界情况：可以一次跳到终点
nums5 = [10, 1, 1, 1, 1]
print("测试用例 5（可以一次跳到终点）:")
print("nums = ", end="")
print_array(nums5)
print(f"贪心算法结果: {solution.jump_greedy(nums5)}") # 预期输出: 1
print(f"动态规划结果: {solution.jump_dynamic_programming(nums5)}") # 预期输出: 1
print(f"BFS 结果: {solution.jump_bfs(nums5)}") # 预期输出: 1
print()

性能测试 - 小规模数组
print("小规模数组性能测试:")
small_array = generate_test_case(100, False)
performance_test(small_array, solution)
print()

性能测试 - 大规模数组 - 只测试贪心算法，因为其他算法在大规模数组上会很慢
print("大规模数组性能测试（只测试贪心算法）:")
large_array = generate_test_case(10000, False)
start_time = time.time()
result = solution.jump_greedy(large_array)
end_time = time.time()
print(f"贪心算法结果: {result}")
print(f"贪心算法耗时: {(end_time - start_time) * 1000:.2f}ms")

最坏情况性能测试
print("\n最坏情况性能测试:")
worst_case_array = generate_test_case(1000, True) # 小规模的最坏情况，否则动态规划和 BFS 会超时
performance_test(worst_case_array, solution)

if __name__ == "__main__":
 main()

```

=====

文件: Code21\_JumpGame.cpp

=====

```

#include <iostream>
#include <vector>
#include <chrono>
#include <random>
```

```
/**
 * LeetCode 55. 跳跃游戏 (Jump Game)
 *
 * 题目描述:
 * 给定一个非负整数数组 nums，你最初位于数组的第一个位置。
 * 数组中的每个元素代表你在该位置可以跳跃的最大长度。
 * 判断你是否能够到达最后一个位置。
 *
 * 示例 1:
 * 输入: nums = [2, 3, 1, 1, 4]
 * 输出: true
 * 解释: 可以先跳 1 步，从位置 0 到达位置 1，然后再从位置 1 跳 3 步到达最后一个位置。
 *
 * 示例 2:
 * 输入: nums = [3, 2, 1, 0, 4]
 * 输出: false
 * 解释: 无论怎样，总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0， 所以永远不能到达最后一个位置。
 *
 * 提示:
 * 1 <= nums.length <= 10^4
 * 0 <= nums[i] <= 10^5
 *
 * 题目链接: https://leetcode.com/problems/jump-game/
 *
 * 解题思路:
 * 这道题可以使用贪心算法来解决。我们的目标是判断是否能够到达最后一个位置。
 *
 * 贪心策略: 维护一个变量表示当前能够到达的最远位置。遍历数组，不断更新这个最远位置。
 * 如果任何时候，当前能够到达的最远位置小于当前遍历到的索引，说明无法到达该位置，也就无法到达最后一个位置。
 *
 * 具体来说，我们维护一个变量 maxReach，表示当前能够到达的最远位置。初始时，maxReach = 0。
 * 遍历数组，对于每个位置 i，如果 i > maxReach，说明无法到达位置 i，返回 false。
 * 否则，更新 maxReach = max(maxReach, i + nums[i])。
 * 如果 maxReach >= nums.size() - 1，说明已经可以到达最后一个位置，返回 true。
 *
 * 时间复杂度: O(n)，其中 n 是数组的长度。我们只需要遍历数组一次。
 * 空间复杂度: O(1)，只使用了常数级别的额外空间。
 *
 * 此外，我们还提供三种其他解法:
 * 1. 动态规划解法（自顶向下）: 时间复杂度 O(n^2)，空间复杂度 O(n)
```

```
* 2. 动态规划解法（自底向上）：时间复杂度 O(n^2)，空间复杂度 O(n)
* 3. 回溯解法：时间复杂度 O(2^n)，空间复杂度 O(n)，但在大规模输入时可能会超时
*/
```

```
class Solution {
public:
 /**
 * 解法一：贪心算法（最优解）
 *
 * @param nums 非负整数数组
 * @return 是否能够到达最后一个位置
 */
 bool canJumpGreedy(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 int maxReach = 0; // 当前能够到达的最远位置

 // 遍历数组
 for (int i = 0; i < nums.size(); i++) {
 // 如果当前位置已经无法到达，返回 false
 if (i > maxReach) {
 return false;
 }

 // 更新能够到达的最远位置
 maxReach = std::max(maxReach, i + nums[i]);
 }

 // 如果已经可以到达或超过最后一个位置，可以提前返回 true
 if (maxReach >= static_cast<int>(nums.size() - 1)) {
 return true;
 }
 }

 // 遍历完整个数组后，判断是否能够到达最后一个位置
 return maxReach >= static_cast<int>(nums.size() - 1);
}

/**
 * 解法二：动态规划 - 自顶向下
 *
```

```

* @param nums 非负整数数组
* @return 是否能够到达最后一个位置
*/
bool canJumpDynamicProgrammingTopDown(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 int n = nums.size();
 // memo[i]表示从位置 i 是否可以到达最后一个位置
 // 0: 未计算, 1: 可以到达, 2: 无法到达
 std::vector<int> memo(n, 0);
 // 最后一个位置可以到达自身
 memo[n - 1] = 1;

 return canJumpFromPosition(0, nums, memo);
}

/***
 * 辅助方法: 判断从位置 pos 是否可以到达最后一个位置
 *
 * @param pos 当前位置
 * @param nums 非负整数数组
 * @param memo 记忆化数组
 * @return 是否能够到达最后一个位置
*/
bool canJumpFromPosition(int pos, const std::vector<int>& nums, std::vector<int>& memo) {
 // 如果已经计算过, 直接返回结果
 if (memo[pos] != 0) {
 return memo[pos] == 1;
 }

 // 计算从当前位置可以到达的最远位置
 int furthestJump = std::min(pos + nums[pos], static_cast<int>(nums.size()) - 1);

 // 尝试从当前位置跳到所有可能的位置
 for (int nextPos = pos + 1; nextPos <= furthestJump; nextPos++) {
 if (canJumpFromPosition(nextPos, nums, memo)) {
 memo[pos] = 1; // 可以到达
 return true;
 }
 }
}

```

```

 memo[pos] = 2; // 无法到达
 return false;
 }

/***
 * 解法三：动态规划 - 自底向上
 *
 * @param nums 非负整数数组
 * @return 是否能够到达最后一个位置
 */
bool canJumpDynamicProgrammingBottomUp(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 int n = nums.size();
 // dp[i]表示从位置 i 是否可以到达最后一个位置
 std::vector<bool> dp(n, false);
 // 最后一个位置可以到达自身
 dp[n - 1] = true;

 // 从后往前遍历
 for (int i = n - 2; i >= 0; i--) {
 // 计算从当前位置可以到达的最远位置
 int furthestJump = std::min(i + nums[i], n - 1);

 // 检查从当前位置能否跳到一个可以到达终点的位置
 for (int j = i + 1; j <= furthestJump; j++) {
 if (dp[j]) {
 dp[i] = true;
 break; // 一旦找到一个可达的位置，就可以停止检查
 }
 }
 }

 // 返回是否可以从起始位置到达终点
 return dp[0];
}

/***
 * 解法四：回溯（暴力解法，在大规模输入时可能会超时）
*/

```

```

*
* @param nums 非负整数数组
* @return 是否能够到达最后一个位置
*/
bool canJumpBacktracking(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 return canJumpFromPositionBacktracking(0, nums);
}

/***
* 辅助方法：使用回溯判断从位置 pos 是否可以到达最后一个位置
*
* @param pos 当前位置
* @param nums 非负整数数组
* @return 是否能够到达最后一个位置
*/
bool canJumpFromPositionBacktracking(int pos, const std::vector<int>& nums) {
 // 基本情况：已经到达最后一个位置
 if (pos == static_cast<int>(nums.size()) - 1) {
 return true;
 }

 // 计算从当前位置可以到达的最远位置
 int furthestJump = std::min(pos + nums[pos], static_cast<int>(nums.size()) - 1);

 // 尝试从当前位置跳到所有可能的位置（优先尝试跳得更远）
 for (int nextPos = furthestJump; nextPos > pos; nextPos--) {
 if (canJumpFromPositionBacktracking(nextPos, nums)) {
 return true;
 }
 }

 return false;
}

/***
* LeetCode 官方接口的实现（使用贪心算法）
*
* @param nums 非负整数数组
*/

```

```

 * @return 是否能够到达最后一个位置
 */
bool canJump(const std::vector<int>& nums) {
 return canJumpGreedy(nums);
}

};

/***
 * 打印数组
*/
void printArray(const std::vector<int>& arr) {
 std::cout << "[";
 for (size_t i = 0; i < arr.size(); i++) {
 std::cout << arr[i];
 if (i < arr.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 性能测试
*/
void performanceTest(const std::vector<int>& nums, Solution& solution) {
 // 测试贪心算法
 auto startTime = std::chrono::high_resolution_clock::now();
 bool result1 = solution.canJumpGreedy(nums);
 auto endTime = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
 std::cout << "贪心算法结果: " << (result1 ? "true" : "false") << std::endl;
 std::cout << "贪心算法耗时: " << duration.count() << "ms" << std::endl;

 // 测试动态规划自底向上
 startTime = std::chrono::high_resolution_clock::now();
 bool result2 = solution.canJumpDynamicProgrammingBottomUp(nums);
 endTime = std::chrono::high_resolution_clock::now();
 duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
 std::cout << "动态规划(自底向上)结果: " << (result2 ? "true" : "false") << std::endl;
 std::cout << "动态规划(自底向上)耗时: " << duration.count() << "ms" << std::endl;

 // 注意: 以下两种方法在大规模数组上可能会超时, 所以只在小规模数组上测试
 if (nums.size() <= 1000) {

```

```

// 测试动态规划自顶向下
startTime = std::chrono::high_resolution_clock::now();
bool result3 = solution.canJumpDynamicProgrammingTopDown(nums);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "动态规划(自顶向下)结果: " << (result3 ? "true" : "false") << std::endl;
std::cout << "动态规划(自顶向下)耗时: " << duration.count() << "ms" << std::endl;

// 测试回溯算法
if (nums.size() <= 30) { // 回溯算法在小数据量下才能快速运行
 startTime = std::chrono::high_resolution_clock::now();
 bool result4 = solution.canJumpBacktracking(nums);
 endTime = std::chrono::high_resolution_clock::now();
 duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime -
startTime);
 std::cout << "回溯算法结果: " << (result4 ? "true" : "false") << std::endl;
 std::cout << "回溯算法耗时: " << duration.count() << "ms" << std::endl;
} else {
 std::cout << "数组过大, 跳过回溯算法测试" << std::endl;
}
} else {
 std::cout << "数组过大, 跳过动态规划(自顶向下)和回溯算法测试" << std::endl;
}

/***
 * 生成测试用例
 */
std::vector<int> generateTestCase(int n, bool canReachEnd) {
 std::vector<int> nums(n);

 std::random_device rd;
 std::mt19937 gen(rd());
 std::uniform_int_distribution<int> distrib(1, 5);

 if (canReachEnd) {
 // 生成可以到达终点的数组
 for (int i = 0; i < n - 1; i++) {
 // 确保可以到达终点, 当前位置的值至少为 n-1-i
 int minVal = n - 1 - i;
 int randVal = distrib(gen);
 nums[i] = std::max(randVal, minVal);
 }
 }
}

```

```

} else {
 // 生成无法到达终点的数组
 std::uniform_int_distribution<> posDistrib(1, n - 2);
 int zeroPosition = posDistrib(gen); // 确保 0 不在最后一个位置
 nums[zeroPosition] = 0;

 // 填充 0 之前的位置
 for (int i = 0; i < zeroPosition; i++) {
 // 确保无法越过 0
 int maxVal = zeroPosition - i;
 int randVal = distrib(gen);
 nums[i] = std::min(randVal, maxVal);
 }

 // 填充 0 之后的位置
 for (int i = zeroPosition + 1; i < n; i++) {
 nums[i] = distrib(gen);
 }
}

nums[n - 1] = 0; // 最后一个元素不影响
return nums;
}

int main() {
 Solution solution;

 // 测试用例 1
 std::vector<int> nums1 = {2, 3, 1, 1, 4};
 std::cout << "测试用例 1:" << std::endl;
 std::cout << "nums = ";
 printArray(nums1);
 std::cout << "贪心算法结果: " << (solution.canJumpGreedy(nums1) ? "true" : "false") <<
std::endl; // 预期输出: true
 std::cout << "动态规划(自顶向下)结果: " <<
(solution.canJumpDynamicProgrammingTopDown(nums1) ? "true" : "false") << std::endl; // 预期输出:
true
 std::cout << "动态规划(自底向上)结果: " <<
(solution.canJumpDynamicProgrammingBottomUp(nums1) ? "true" : "false") << std::endl; // 预期输出:
true
 std::cout << "回溯算法结果: " << (solution.canJumpBacktracking(nums1) ? "true" : "false") <<
std::endl; // 预期输出: true
 std::cout << std::endl;
}

```

```

// 测试用例 2
std::vector<int> nums2 = {3, 2, 1, 0, 4};
std::cout << "测试用例 2:" << std::endl;
std::cout << "nums = ";
printArray(nums2);
std::cout << "贪心算法结果: " << (solution.canJumpGreedy(nums2) ? "true" : "false") <<
std::endl; // 预期输出: false
std::cout << "动态规划(自顶向下)结果: " <<
(solution.canJumpDynamicProgrammingTopDown(nums2) ? "true" : "false") << std::endl; // 预期输出:
false
std::cout << "动态规划(自底向上)结果: " <<
(solution.canJumpDynamicProgrammingBottomUp(nums2) ? "true" : "false") << std::endl; // 预期输出:
false
std::cout << "回溯算法结果: " << (solution.canJumpBacktracking(nums2) ? "true" : "false") <<
std::endl; // 预期输出: false
std::cout << std::endl;

// 测试用例 3 - 边界情况: 只有一个元素
std::vector<int> nums3 = {0};
std::cout << "测试用例 3 (单元素数组) :" << std::endl;
std::cout << "nums = ";
printArray(nums3);
std::cout << "贪心算法结果: " << (solution.canJumpGreedy(nums3) ? "true" : "false") <<
std::endl; // 预期输出: true
std::cout << "动态规划(自顶向下)结果: " <<
(solution.canJumpDynamicProgrammingTopDown(nums3) ? "true" : "false") << std::endl; // 预期输出:
true
std::cout << "动态规划(自底向上)结果: " <<
(solution.canJumpDynamicProgrammingBottomUp(nums3) ? "true" : "false") << std::endl; // 预期输出:
true
std::cout << "回溯算法结果: " << (solution.canJumpBacktracking(nums3) ? "true" : "false") <<
std::endl; // 预期输出: true
std::cout << std::endl;

// 测试用例 4 - 边界情况: 全是 0
std::vector<int> nums4 = {0, 0, 0, 0, 0};
std::cout << "测试用例 4 (全是 0) :" << std::endl;
std::cout << "nums = ";
printArray(nums4);
std::cout << "贪心算法结果: " << (solution.canJumpGreedy(nums4) ? "true" : "false") <<
std::endl; // 预期输出: false
std::cout << "动态规划(自顶向下)结果: " <<

```

```

(solution.canJumpDynamicProgrammingTopDown(nums4) ? "true" : "false") << std::endl; // 预期输出:
false

 std::cout << "动态规划(自底向上)结果: " <<
(solution.canJumpDynamicProgrammingBottomUp(nums4) ? "true" : "false") << std::endl; // 预期输出:
false

 std::cout << "回溯算法结果: " << (solution.canJumpBacktracking(nums4) ? "true" : "false") <<
std::endl; // 预期输出: false

 std::cout << std::endl;

// 测试用例 5 - 边界情况: 可以一次跳到终点
std::vector<int> nums5 = {10, 0, 0, 0, 0};
std::cout << "测试用例 5 (可以一次跳到终点) :" << std::endl;
std::cout << "nums = ";
printArray(nums5);
std::cout << "贪心算法结果: " << (solution.canJumpGreedy(nums5) ? "true" : "false") <<
std::endl; // 预期输出: true

 std::cout << "动态规划(自顶向下)结果: " <<
(solution.canJumpDynamicProgrammingTopDown(nums5) ? "true" : "false") << std::endl; // 预期输出:
true

 std::cout << "动态规划(自底向上)结果: " <<
(solution.canJumpDynamicProgrammingBottomUp(nums5) ? "true" : "false") << std::endl; // 预期输出:
true

 std::cout << "回溯算法结果: " << (solution.canJumpBacktracking(nums5) ? "true" : "false") <<
std::endl; // 预期输出: true

 std::cout << std::endl;

// 性能测试 - 小规模数组
std::cout << "小规模数组性能测试 (可以到达终点) :" << std::endl;
std::vector<int> smallArray1 = generateTestCase(100, true);
performanceTest(smallArray1, solution);
std::cout << std::endl;

std::cout << "小规模数组性能测试 (无法到达终点) :" << std::endl;
std::vector<int> smallArray2 = generateTestCase(100, false);
performanceTest(smallArray2, solution);
std::cout << std::endl;

// 性能测试 - 大规模数组 - 只测试贪心算法, 因为其他算法在大规模数组上会很慢
std::cout << "大规模数组性能测试 (只测试贪心算法) :" << std::endl;
std::vector<int> largeArray = generateTestCase(10000, true);
auto startTime = std::chrono::high_resolution_clock::now();
bool result = solution.canJumpGreedy(largeArray);
auto endTime = std::chrono::high_resolution_clock::now();

```

```
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "贪心算法结果: " << (result ? "true" : "false") << std::endl;
std::cout << "贪心算法耗时: " << duration.count() << "ms" << std::endl;

return 0;
}
```

---

文件: Code21\_JumpGame.java

```
=====
import java.util.*;
import java.util.stream.Collectors;

/**
 * LeetCode 55. 跳跃游戏 (Jump Game)
 *
 * 题目描述:
 * 给定一个非负整数数组 nums，你最初位于数组的第一个位置。
 * 数组中的每个元素代表你在该位置可以跳跃的最大长度。
 * 判断你是否能够到达最后一个位置。
 *
 * 示例 1:
 * 输入: nums = [2, 3, 1, 1, 4]
 * 输出: true
 * 解释: 可以先跳 1 步，从位置 0 到达位置 1，然后再从位置 1 跳 3 步到达最后一个位置。
 *
 * 示例 2:
 * 输入: nums = [3, 2, 1, 0, 4]
 * 输出: false
 * 解释: 无论怎样，总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0， 所以永远不能到达最后一个位置。
 *
 * 提示:
 * 1 <= nums.length <= 10^4
 * 0 <= nums[i] <= 10^5
 *
 * 题目链接: https://leetcode.com/problems/jump-game/
 *
 * 解题思路:
 * 这道题可以使用贪心算法来解决。我们的目标是判断是否能够到达最后一个位置。
 *
 * 贪心策略: 维护一个变量表示当前能够到达的最远位置。遍历数组，不断更新这个最远位置。
```

- \* 如果在任何时候，当前能够到达的最远位置小于当前遍历到的索引，说明无法到达该位置，也就无法到达最后一个位置。
  - \*
  - \* 具体来说，我们维护一个变量 maxReach，表示当前能够到达的最远位置。初始时，`maxReach = 0`。
  - \* 遍历数组，对于每个位置 `i`，如果 `i > maxReach`，说明无法到达位置 `i`，返回 `false`。
  - \* 否则，更新 `maxReach = max(maxReach, i + nums[i])`。
  - \* 如果 `maxReach >= nums.length - 1`，说明已经可以到达最后一个位置，返回 `true`。
  - \*
  - \* 时间复杂度： $O(n)$ ，其中 `n` 是数组的长度。我们只需要遍历数组一次。
  - \* 空间复杂度： $O(1)$ ，只使用了常数级别的额外空间。
  - \*
- \* 此外，我们还提供两种其他解法：
  - \* 1. 动态规划解法：时间复杂度  $O(n^2)$ ，空间复杂度  $O(n)$
  - \* 2. 回溯解法：时间复杂度  $O(2^n)$ ，空间复杂度  $O(n)$ ，但在大数输入时可能会超时

```
public class Code21_JumpGame {

 /**
 * 解法一：贪心算法（最优解）
 *
 * @param nums 非负整数数组
 * @return 是否能够到达最后一个位置
 */

 public static boolean canJumpGreedy(int[] nums) {
 // 参数校验
 if (nums == null || nums.length <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 int maxReach = 0; // 当前能够到达的最远位置

 // 遍历数组
 for (int i = 0; i < nums.length; i++) {
 // 如果当前位置已经无法到达，返回 false
 if (i > maxReach) {
 return false;
 }

 // 更新能够到达的最远位置
 maxReach = Math.max(maxReach, i + nums[i]);
 }

 // 如果已经可以到达或超过最后一个位置，可以提前返回 true
 }
}
```

```

 if (maxReach >= nums.length - 1) {
 return true;
 }
 }

 // 遍历完整个数组后，判断是否能够到达最后一个位置
 return maxReach >= nums.length - 1;
}

/***
 * 解法二：动态规划 - 自顶向下
 *
 * @param nums 非负整数数组
 * @return 是否能够到达最后一个位置
 */
public static boolean canJumpDynamicProgrammingTopDown(int[] nums) {
 // 参数校验
 if (nums == null || nums.length <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 int n = nums.length;
 // memo[i]表示从位置 i 是否可以到达最后一个位置
 // 0: 未计算, 1: 可以到达, 2: 无法到达
 int[] memo = new int[n];
 // 最后一个位置可以到达自身
 memo[n - 1] = 1;

 return canJumpFromPosition(0, nums, memo);
}

/***
 * 辅助方法：判断从位置 pos 是否可以到达最后一个位置
 *
 * @param pos 当前位置
 * @param nums 非负整数数组
 * @param memo 记忆化数组
 * @return 是否能够到达最后一个位置
 */
private static boolean canJumpFromPosition(int pos, int[] nums, int[] memo) {
 // 如果已经计算过，直接返回结果
 if (memo[pos] != 0) {
 return memo[pos] == 1;
 }
}

```

```

}

// 计算从当前位置可以到达的最远位置
int furthestJump = Math.min(pos + nums[pos], nums.length - 1);

// 尝试从当前位置跳到所有可能的位置
for (int nextPos = pos + 1; nextPos <= furthestJump; nextPos++) {
 if (canJumpFromPosition(nextPos, nums, memo)) {
 memo[pos] = 1; // 可以到达
 return true;
 }
}

memo[pos] = 2; // 无法到达
return false;
}

/***
 * 解法三：动态规划 - 自底向上
 *
 * @param nums 非负整数数组
 * @return 是否能够到达最后一个位置
 */
public static boolean canJumpDynamicProgrammingBottomUp(int[] nums) {
 // 参数校验
 if (nums == null || nums.length <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 int n = nums.length;
 // dp[i] 表示从位置 i 是否可以到达最后一个位置
 boolean[] dp = new boolean[n];
 // 最后一个位置可以到达自身
 dp[n - 1] = true;

 // 从后往前遍历
 for (int i = n - 2; i >= 0; i--) {
 // 计算从当前位置可以到达的最远位置
 int furthestJump = Math.min(i + nums[i], n - 1);

 // 检查从当前位置能否跳到一个可以到达终点的位置
 for (int j = i + 1; j <= furthestJump; j++) {
 if (dp[j]) {

```

```

 dp[i] = true;
 break; // 一旦找到一个可达的位置，就可以停止检查
 }
}

// 返回是否可以从起始位置到达终点
return dp[0];
}

/***
 * 解法四：回溯（暴力解法，在大规模输入时可能会超时）
 *
 * @param nums 非负整数数组
 * @return 是否能够到达最后一个位置
 */
public static boolean canJumpBacktracking(int[] nums) {
 // 参数校验
 if (nums == null || nums.length <= 1) {
 return true; // 如果数组为空或只有一个元素，已经在终点
 }

 return canJumpFromPositionBacktracking(0, nums);
}

/***
 * 辅助方法：使用回溯判断从位置 pos 是否可以到达最后一个位置
 *
 * @param pos 当前位置
 * @param nums 非负整数数组
 * @return 是否能够到达最后一个位置
 */
private static boolean canJumpFromPositionBacktracking(int pos, int[] nums) {
 // 基本情况：已经到达最后一个位置
 if (pos == nums.length - 1) {
 return true;
 }

 // 计算从当前位置可以到达的最远位置
 int furthestJump = Math.min(pos + nums[pos], nums.length - 1);

 // 尝试从当前位置跳到所有可能的位置
 for (int nextPos = furthestJump; nextPos > pos; nextPos--) {

```

```
// 优先尝试跳得更远，这样可能更快找到解
if (canJumpFromPositionBacktracking(nextPos, nums)) {
 return true;
}

return false;
}

/**
 * 打印数组
 */
public static void printArray(int[] arr) {
 System.out.println(Arrays.stream(arr)
 .mapToObj(String::valueOf)
 .collect(Collectors.joining(", ", "[", "]")));
}

/**
 * 性能测试
 */
public static void performanceTest(int[] nums) {
 // 测试贪心算法
 long startTime = System.currentTimeMillis();
 boolean result1 = canJumpGreedy(nums);
 long endTime = System.currentTimeMillis();
 System.out.println("贪心算法结果: " + result1);
 System.out.println("贪心算法耗时: " + (endTime - startTime) + "ms");

 // 测试动态规划自底向上
 startTime = System.currentTimeMillis();
 boolean result2 = canJumpDynamicProgrammingBottomUp(nums);
 endTime = System.currentTimeMillis();
 System.out.println("动态规划(自底向上)结果: " + result2);
 System.out.println("动态规划(自底向上)耗时: " + (endTime - startTime) + "ms");

 // 注意: 以下两种方法在大规模数组上可能会超时, 所以只在小规模数组上测试
 if (nums.length <= 1000) {
 // 测试动态规划自顶向下
 startTime = System.currentTimeMillis();
 boolean result3 = canJumpDynamicProgrammingTopDown(nums);
 endTime = System.currentTimeMillis();
 System.out.println("动态规划(自顶向下)结果: " + result3);
 }
}
```

```

System.out.println("动态规划(自顶向下)耗时: " + (endTime - startTime) + "ms");

// 测试回溯算法
if (nums.length <= 30) { // 回溯算法在小数据量下才能快速运行
 startTime = System.currentTimeMillis();
 boolean result4 = canJumpBacktracking(nums);
 endTime = System.currentTimeMillis();
 System.out.println("回溯算法结果: " + result4);
 System.out.println("回溯算法耗时: " + (endTime - startTime) + "ms");
} else {
 System.out.println("数组过大, 跳过回溯算法测试");
}
} else {
 System.out.println("数组过大, 跳过动态规划(自顶向下)和回溯算法测试");
}
}

/***
 * 生成测试用例
 */
public static int[] generateTestCase(int n, boolean canReachEnd) {
 int[] nums = new int[n];
 Random rand = new Random();

 if (canReachEnd) {
 // 生成可以到达终点的数组
 for (int i = 0; i < n - 1; i++) {
 // 确保可以到达终点, 当前位置的值至少为 n-1-i
 nums[i] = Math.max(rand.nextInt(5) + 1, n - 1 - i);
 }
 } else {
 // 生成无法到达终点的数组
 // 创建一个 0, 使得无法越过
 int zeroPosition = rand.nextInt(n - 1) + 1; // 确保 0 不在最后一个位置
 nums[zeroPosition] = 0;

 // 填充 0 之前的位置
 for (int i = 0; i < zeroPosition; i++) {
 // 确保无法越过 0
 nums[i] = Math.min(rand.nextInt(5) + 1, zeroPosition - i);
 }

 // 填充 0 之后的位置
 }
}

```

```
 for (int i = zeroPosition + 1; i < n; i++) {
 nums[i] = rand.nextInt(5) + 1;
 }
 }

 nums[n - 1] = 0; // 最后一个元素不影响
 return nums;
}

public static void main(String[] args) {
 // 测试用例 1
 int[] nums1 = {2, 3, 1, 1, 4};
 System.out.println("测试用例 1:");
 System.out.print("nums = ");
 printArray(nums1);
 System.out.println("贪心算法结果: " + canJumpGreedy(nums1)); // 预期输出: true
 System.out.println("动态规划(自顶向下)结果: " + canJumpDynamicProgrammingTopDown(nums1));
 // 预期输出: true
 System.out.println("动态规划(自底向上)结果: " +
 canJumpDynamicProgrammingBottomUp(nums1)); // 预期输出: true
 System.out.println("回溯算法结果: " + canJumpBacktracking(nums1)); // 预期输出: true
 System.out.println();

 // 测试用例 2
 int[] nums2 = {3, 2, 1, 0, 4};
 System.out.println("测试用例 2:");
 System.out.print("nums = ");
 printArray(nums2);
 System.out.println("贪心算法结果: " + canJumpGreedy(nums2)); // 预期输出: false
 System.out.println("动态规划(自顶向下)结果: " + canJumpDynamicProgrammingTopDown(nums2));
 // 预期输出: false
 System.out.println("动态规划(自底向上)结果: " +
 canJumpDynamicProgrammingBottomUp(nums2)); // 预期输出: false
 System.out.println("回溯算法结果: " + canJumpBacktracking(nums2)); // 预期输出: false
 System.out.println();

 // 测试用例 3 - 边界情况: 只有一个元素
 int[] nums3 = {0};
 System.out.println("测试用例 3 (单元素数组):");
 System.out.print("nums = ");
 printArray(nums3);
 System.out.println("贪心算法结果: " + canJumpGreedy(nums3)); // 预期输出: true
 System.out.println("动态规划(自顶向下)结果: " + canJumpDynamicProgrammingTopDown(nums3));
```

```

// 预期输出: true
 System.out.println("动态规划(自底向上)结果: " +
canJumpDynamicProgrammingBottomUp(nums3)); // 预期输出: true
 System.out.println("回溯算法结果: " + canJumpBacktracking(nums3)); // 预期输出: true
 System.out.println();

// 测试用例 4 - 边界情况: 全是 0
int[] nums4 = {0, 0, 0, 0, 0};
System.out.println("测试用例 4 (全是 0) :");
System.out.print("nums = ");
printArray(nums4);
System.out.println("贪心算法结果: " + canJumpGreedy(nums4)); // 预期输出: false (除了第一个元素为 0 且只有一个元素的情况)
 System.out.println("动态规划(自顶向下)结果: " + canJumpDynamicProgrammingTopDown(nums4));
// 预期输出: false
 System.out.println("动态规划(自底向上)结果: " +
canJumpDynamicProgrammingBottomUp(nums4)); // 预期输出: false
 System.out.println("回溯算法结果: " + canJumpBacktracking(nums4)); // 预期输出: false
 System.out.println();

// 测试用例 5 - 边界情况: 可以一次跳到终点
int[] nums5 = {10, 0, 0, 0, 0};
System.out.println("测试用例 5 (可以一次跳到终点) :");
System.out.print("nums = ");
printArray(nums5);
System.out.println("贪心算法结果: " + canJumpGreedy(nums5)); // 预期输出: true
System.out.println("动态规划(自顶向下)结果: " + canJumpDynamicProgrammingTopDown(nums5));
// 预期输出: true
 System.out.println("动态规划(自底向上)结果: " +
canJumpDynamicProgrammingBottomUp(nums5)); // 预期输出: true
 System.out.println("回溯算法结果: " + canJumpBacktracking(nums5)); // 预期输出: true
 System.out.println();

// 性能测试 - 小规模数组
System.out.println("小规模数组性能测试 (可以到达终点) :");
int[] smallArray1 = generateTestCase(100, true);
performanceTest(smallArray1);
System.out.println();

System.out.println("小规模数组性能测试 (无法到达终点) :");
int[] smallArray2 = generateTestCase(100, false);
performanceTest(smallArray2);
System.out.println();

```

```
// 性能测试 - 大规模数组 - 只测试贪心算法，因为其他算法在大规模数组上会很慢
System.out.println("大规模数组性能测试（只测试贪心算法）:");
int[] largeArray = generateTestCase(10000, true);
long startTime = System.currentTimeMillis();
boolean result = canJumpGreedy(largeArray);
long endTime = System.currentTimeMillis();
System.out.println("贪心算法结果: " + result);
System.out.println("贪心算法耗时: " + (endTime - startTime) + "ms");
}
}
```

=====

文件: Code21\_JumpGame.py

=====

```
import time
import random
from typing import List
```

"""

LeetCode 55. 跳跃游戏 (Jump Game)

题目描述:

给定一个非负整数数组 `nums`，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个位置。

示例 1:

输入: `nums = [2, 3, 1, 1, 4]`

输出: `true`

解释: 可以先跳 1 步，从位置 0 到达位置 1，然后再从位置 1 跳 3 步到达最后一个位置。

示例 2:

输入: `nums = [3, 2, 1, 0, 4]`

输出: `false`

解释: 无论怎样，总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0，所以永远不能到达最后一个位置。

提示:

$1 \leq \text{nums.length} \leq 10^4$

$0 \leq \text{nums}[i] \leq 10^5$

题目链接: <https://leetcode.com/problems/jump-game/>

解题思路:

这道题可以使用贪心算法来解决。我们的目标是判断是否能够到达最后一个位置。

贪心策略: 维护一个变量表示当前能够到达的最远位置。遍历数组, 不断更新这个最远位置。

如果在任何时候, 当前能够到达的最远位置小于当前遍历到的索引, 说明无法到达该位置, 也就无法到达最后一个位置。

具体来说, 我们维护一个变量 `max_reach`, 表示当前能够到达的最远位置。初始时, `max_reach = 0`。

遍历数组, 对于每个位置 `i`, 如果 `i > max_reach`, 说明无法到达位置 `i`, 返回 `False`。

否则, 更新 `max_reach = max(max_reach, i + nums[i])`。

如果 `max_reach >= len(nums) - 1`, 说明已经可以到达最后一个位置, 返回 `True`。

时间复杂度:  $O(n)$ , 其中  $n$  是数组的长度。我们只需要遍历数组一次。

空间复杂度:  $O(1)$ , 只使用了常数级别的额外空间。

此外, 我们还提供三种其他解法:

1. 动态规划解法 (自顶向下): 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n)$
2. 动态规划解法 (自底向上): 时间复杂度  $O(n^2)$ , 空间复杂度  $O(n)$
3. 回溯解法: 时间复杂度  $O(2^n)$ , 空间复杂度  $O(n)$ , 但在大规模输入时可能会超时

"""

```
class Solution:
```

```
 def can_jump_greedy(self, nums: List[int]) -> bool:
```

```
 """
```

解法一: 贪心算法 (最优解)

Args:

nums: 非负整数数组

Returns:

是否能够到达最后一个位置

```
 """
```

# 参数校验

```
 if not nums or len(nums) <= 1:
```

```
 return True # 如果数组为空或只有一个元素, 已经在终点
```

```
 max_reach = 0 # 当前能够到达的最远位置
```

# 遍历数组

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

```
 # 如果当前位置已经无法到达, 返回 False
```

```

 if i > max_reach:
 return False

 # 更新能够到达的最远位置
 max_reach = max(max_reach, i + nums[i])

 # 如果已经可以到达或超过最后一个位置，可以提前返回 True
 if max_reach >= len(nums) - 1:
 return True

遍历完整个数组后，判断是否能够到达最后一个位置
return max_reach >= len(nums) - 1

```

```
def can_jump_dynamic_programming_top_down(self, nums: List[int]) -> bool:
 """

```

解法二：动态规划 – 自顶向下

Args:

nums: 非负整数数组

Returns:

是否能够到达最后一个位置

"""

# 参数校验

```
if not nums or len(nums) <= 1:
 return True # 如果数组为空或只有一个元素，已经在终点
```

```
n = len(nums)
```

# memo[i] 表示从位置 i 是否可以到达最后一个位置

# 0: 未计算, 1: 可以到达, 2: 无法到达

```
memo = [0] * n
```

# 最后一个位置可以到达自身

```
memo[n - 1] = 1
```

```
def can_jump_from_position(pos: int) -> bool:
```

# 如果已经计算过，直接返回结果

```
if memo[pos] != 0:
```

```
 return memo[pos] == 1
```

# 计算从当前位置可以到达的最远位置

```
furthest_jump = min(pos + nums[pos], n - 1)
```

# 尝试从当前位置跳到所有可能的位置

```

 for next_pos in range(pos + 1, furthest_jump + 1):
 if can_jump_from_position(next_pos):
 memo[pos] = 1 # 可以到达
 return True

 memo[pos] = 2 # 无法到达
 return False

 return can_jump_from_position(0)

```

```

def can_jump_dynamic_programming_bottom_up(self, nums: List[int]) -> bool:
 """

```

解法三：动态规划 - 自底向上

Args:

nums: 非负整数数组

Returns:

是否能够到达最后一个位置

"""

# 参数校验

```

if not nums or len(nums) <= 1:
 return True # 如果数组为空或只有一个元素，已经在终点

```

n = len(nums)

# dp[i] 表示从位置 i 是否可以到达最后一个位置

dp = [False] \* n

# 最后一个位置可以到达自身

dp[n - 1] = True

# 从后往前遍历

for i in range(n - 2, -1, -1):

# 计算从当前位置可以到达的最远位置

furthest\_jump = min(i + nums[i], n - 1)

# 检查从当前位置能否跳到一个可以到达终点的位置

for j in range(i + 1, furthest\_jump + 1):

if dp[j]:

dp[i] = True

break # 一旦找到一个可达的位置，就可以停止检查

# 返回是否可以从起始位置到达终点

return dp[0]

```
def can_jump_backtracking(self, nums: List[int]) -> bool:
```

```
 """
```

解法四：回溯（暴力解法，在大规模输入时可能会超时）

Args:

    nums: 非负整数数组

Returns:

    是否能够到达最后一个位置

```
 """
```

# 参数校验

```
if not nums or len(nums) <= 1:
 return True # 如果数组为空或只有一个元素，已经在终点
```

```
def can_jump_from_position(pos: int) -> bool:
```

# 基本情况：已经到达最后一个位置

```
if pos == len(nums) - 1:
```

```
 return True
```

# 计算从当前位置可以到达的最远位置

```
furthest_jump = min(pos + nums[pos], len(nums) - 1)
```

# 尝试从当前位置跳到所有可能的位置（优先尝试跳得更远）

```
for next_pos in range(furthest_jump, pos, -1):
```

```
 if can_jump_from_position(next_pos):
```

```
 return True
```

```
return False
```

```
return can_jump_from_position(0)
```

```
def can_jump(self, nums: List[int]) -> bool:
```

```
 """
```

LeetCode 官方接口的实现（使用贪心算法）

Args:

    nums: 非负整数数组

Returns:

    是否能够到达最后一个位置

```
 """
```

```
return self.can_jump_greedy(nums)
```

```
"""
打印数组
"""

def print_array(arr: List[int]) -> None:
 print(f"[{', '.join(map(str, arr))}]")

"""
性能测试
"""

def performance_test(nums: List[int], solution: Solution) -> None:
 # 测试贪心算法
 start_time = time.time()
 result1 = solution.can_jump_greedy(nums)
 end_time = time.time()
 print(f"贪心算法结果: {result1}")
 print(f"贪心算法耗时: {(end_time - start_time) * 1000:.2f}ms")

 # 测试动态规划自底向上
 start_time = time.time()
 result2 = solution.can_jump_dynamic_programming_bottom_up(nums)
 end_time = time.time()
 print(f"动态规划(自底向上)结果: {result2}")
 print(f"动态规划(自底向上)耗时: {(end_time - start_time) * 1000:.2f}ms")

 # 注意: 以下两种方法在大规模数组上可能会超时, 所以只在小规模数组上测试
 if len(nums) <= 1000:
 # 测试动态规划自顶向下
 start_time = time.time()
 result3 = solution.can_jump_dynamic_programming_top_down(nums)
 end_time = time.time()
 print(f"动态规划(自顶向下)结果: {result3}")
 print(f"动态规划(自顶向下)耗时: {(end_time - start_time) * 1000:.2f}ms")

 # 测试回溯算法
 if len(nums) <= 30: # 回溯算法在小数据量下才能快速运行
 start_time = time.time()
 result4 = solution.can_jump_backtracking(nums)
 end_time = time.time()
 print(f"回溯算法结果: {result4}")
 print(f"回溯算法耗时: {(end_time - start_time) * 1000:.2f}ms")
 else:
 print("数组过大, 跳过回溯算法测试")
```

```
else:
 print("数组过大，跳过动态规划(自顶向下)和回溯算法测试")

"""
生成测试用例
"""

def generate_test_case(n: int, can_reach_end: bool) -> List[int]:
 nums = [0] * n

 if can_reach_end:
 # 生成可以到达终点的数组
 for i in range(n - 1):
 # 确保可以到达终点，当前位置的值至少为 n-1-i
 min_val = n - 1 - i
 rand_val = random.randint(1, 5)
 nums[i] = max(rand_val, min_val)
 else:
 # 生成无法到达终点的数组
 # 创建一个 0，使得无法越过
 zero_position = random.randint(1, n - 2) # 确保 0 不在最后一个位置
 nums[zero_position] = 0

 # 填充 0 之前的位置
 for i in range(zero_position):
 # 确保无法越过 0
 max_val = zero_position - i
 rand_val = random.randint(1, 5)
 nums[i] = min(rand_val, max_val)

 # 填充 0 之后的位置
 for i in range(zero_position + 1, n):
 nums[i] = random.randint(1, 5)

 nums[n - 1] = 0 # 最后一个元素不影响
 return nums

def main():
 solution = Solution()

 # 测试用例 1
 nums1 = [2, 3, 1, 1, 4]
 print("测试用例 1:")
 print("nums = ", end="")
```

```
print_array(nums1)
print(f"贪心算法结果: {solution.can_jump_greedy(nums1)}") # 预期输出: True
print(f"动态规划(自顶向下)结果: {solution.can_jump_dynamic_programming_top_down(nums1)}") #
预期输出: True
 print(f"动态规划(自底向上)结果: {solution.can_jump_dynamic_programming_bottom_up(nums1)}") #
预期输出: True
 print(f"回溯算法结果: {solution.can_jump_backtracking(nums1)}") # 预期输出: True
 print()

测试用例 2
nums2 = [3, 2, 1, 0, 4]
print("测试用例 2:")
print("nums =", end="")
print_array(nums2)
print(f"贪心算法结果: {solution.can_jump_greedy(nums2)}") # 预期输出: False
print(f"动态规划(自顶向下)结果: {solution.can_jump_dynamic_programming_top_down(nums2)}") #
预期输出: False
 print(f"动态规划(自底向上)结果: {solution.can_jump_dynamic_programming_bottom_up(nums2)}") #
预期输出: False
 print(f"回溯算法结果: {solution.can_jump_backtracking(nums2)}") # 预期输出: False
 print()

测试用例 3 - 边界情况: 只有一个元素
nums3 = [0]
print("测试用例 3 (单元素数组) :")
print("nums =", end="")
print_array(nums3)
print(f"贪心算法结果: {solution.can_jump_greedy(nums3)}") # 预期输出: True
print(f"动态规划(自顶向下)结果: {solution.can_jump_dynamic_programming_top_down(nums3)}") #
预期输出: True
 print(f"动态规划(自底向上)结果: {solution.can_jump_dynamic_programming_bottom_up(nums3)}") #
预期输出: True
 print(f"回溯算法结果: {solution.can_jump_backtracking(nums3)}") # 预期输出: True
 print()

测试用例 4 - 边界情况: 全是 0
nums4 = [0, 0, 0, 0, 0]
print("测试用例 4 (全是 0) :")
print("nums =", end="")
print_array(nums4)
print(f"贪心算法结果: {solution.can_jump_greedy(nums4)}") # 预期输出: False
print(f"动态规划(自顶向下)结果: {solution.can_jump_dynamic_programming_top_down(nums4)}") #
预期输出: False
```

```
print(f"动态规划(自底向上)结果: {solution.can_jump_dynamic_programming_bottom_up(nums4)}") #
预期输出: False
print(f"回溯算法结果: {solution.can_jump_backtracking(nums4)}") # 预期输出: False
print()

测试用例 5 - 边界情况: 可以一次跳到终点
nums5 = [10, 0, 0, 0, 0]
print("测试用例 5 (可以一次跳到终点) :")
print("nums = ", end="")
print_array(nums5)
print(f"贪心算法结果: {solution.can_jump_greedy(nums5)}") # 预期输出: True
print(f"动态规划(自顶向下)结果: {solution.can_jump_dynamic_programming_top_down(nums5)}") #
预期输出: True
print(f"动态规划(自底向上)结果: {solution.can_jump_dynamic_programming_bottom_up(nums5)}") #
预期输出: True
print(f"回溯算法结果: {solution.can_jump_backtracking(nums5)}") # 预期输出: True
print()

性能测试 - 小规模数组
print("小规模数组性能测试 (可以到达终点) :")
small_array1 = generate_test_case(100, True)
performance_test(small_array1, solution)
print()

print("小规模数组性能测试 (无法到达终点) :")
small_array2 = generate_test_case(100, False)
performance_test(small_array2, solution)
print()

性能测试 - 大规模数组 - 只测试贪心算法, 因为其他算法在大规模数组上会很慢
print("大规模数组性能测试 (只测试贪心算法) :")
large_array = generate_test_case(10000, True)
start_time = time.time()
result = solution.can_jump_greedy(large_array)
end_time = time.time()
print(f"贪心算法结果: {result}")
print(f"贪心算法耗时: {(end_time - start_time) * 1000:.2f}ms")

if __name__ == "__main__":
 main()

=====
```

文件: Code22\_RemoveDuplicatesFromSortedArray.cpp

```
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>

/***
 * LeetCode 26. 删除有序数组中的重复项 (Remove Duplicates from Sorted Array)
 *
 * 题目描述:
 * 给你一个升序排列的数组 nums，请你原地删除重复出现的元素，使得每个元素只出现一次，返回删除后数组的新长度。
 *
 * 元素的相对顺序应该保持一致。
 * 由于在某些语言中不能改变数组的长度，所以必须将结果放在数组 nums 的第一部分。
 * 更规范地说，如果在删除重复项之后有 k 个元素，那么 nums 的前 k 个元素应该保存最终结果。
 * 将最终结果插入 nums 的前 k 个位置后返回 k。
 * 不要使用额外的空间，你必须在原地修改输入数组并在 O(1) 额外空间的条件下完成。
 *
 * 示例 1:
 * 输入: nums = [1, 1, 2]
 * 输出: 2, nums = [1, 2, _]
 * 解释: 函数应该返回新的长度 2，并且原数组 nums 的前两个元素被修改为 1, 2。
 * 不需要考虑数组中超出新长度后面的元素。
 *
 * 示例 2:
 * 输入: nums = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]
 * 输出: 5, nums = [0, 1, 2, 3, 4, _, _, _, _, _]
 * 解释: 函数应该返回新的长度 5，并且原数组 nums 的前五个元素被修改为 0, 1, 2, 3, 4。
 * 不需要考虑数组中超出新长度后面的元素。
 *
 * 提示:
 * 0 <= nums.length <= 3 * 10^4
 * -10^4 <= nums[i] <= 10^4
 * nums 已按升序排列
 *
 * 题目链接: https://leetcode.com/problems/remove-duplicates-from-sorted-array/
 *
 * 解题思路:
 * 这道题是一个经典的双指针应用场景。由于数组是已排序的，所有重复的元素都会相邻。
 * 我们可以使用快慢指针来解决这个问题:
 * 1. 慢指针 slow 指向当前已处理好的不重复元素的最后一个位置
 * 2. 快指针 fast 遍历整个数组
```

```
* 3. 当 nums[fast] 不等于 nums[slow] 时，说明找到了一个新的不重复元素，将 slow 向前移动一位，然后将
nums[fast] 赋值给 nums[slow]
* 4. 当 nums[fast] 等于 nums[slow] 时，说明遇到了重复元素，fast 继续向前移动
* 5. 遍历结束后，slow+1 就是新数组的长度
*
* 时间复杂度：O(n)，其中 n 是数组的长度。快指针最多遍历数组一次。
* 空间复杂度：O(1)，只使用了常数级别的额外空间。
*/
```

```
class Solution {
public:
 /**
 * 解法一：快慢指针（最优解）
 *
 * @param nums 升序排列的数组
 * @return 删除重复元素后的数组新长度
 */
 int removeDuplicates(std::vector<int>& nums) {
 // 边界情况：数组长度为 0 或 1，直接返回原长度
 int n = nums.size();
 if (n <= 1) {
 return n;
 }

 int slow = 0; // 慢指针，指向当前已处理好的不重复元素的最后一个位置

 // 快指针遍历整个数组
 for (int fast = 1; fast < n; fast++) {
 // 当遇到不同的元素时
 if (nums[fast] != nums[slow]) {
 slow++; // 慢指针向前移动一位
 nums[slow] = nums[fast]; // 将不重复的元素移动到前面
 }
 // 当遇到相同的元素时，fast 继续向前移动，slow 保持不变
 }

 // 新数组的长度是 slow + 1
 return slow + 1;
 }

 /**
 * 解法二：快慢指针的另一种写法，包含更多优化
 *
```

```

* @param nums 升序排列的数组
* @return 删除重复元素后的数组新长度
*/
int removeDuplicatesOptimized(std::vector<int>& nums) {
 // 边界情况：数组长度为 0 或 1，直接返回原长度
 int n = nums.size();
 if (n <= 1) {
 return n;
 }

 int slow = 1; // 这里 slow 初始化为 1，表示第一个元素已经是唯一的

 // 从第二个元素开始遍历
 for (int fast = 1; fast < n; fast++) {
 // 当遇到不同的元素时
 if (nums[fast] != nums[fast - 1]) {
 nums[slow] = nums[fast]; // 将不重复的元素移动到前面
 slow++; // 慢指针向前移动一位
 }
 }
}

return slow;
}

/***
* 解法三：优化版快慢指针，适用于更通用的场景
*
* @param nums 升序排列的数组
* @return 删除重复元素后的数组新长度
*/
int removeDuplicatesGeneric(std::vector<int>& nums) {
 // 边界情况：数组长度为 0，直接返回 0
 int n = nums.size();
 if (n == 0) {
 return 0;
 }

 // 初始化慢指针
 int insertPos = 1;

 // 从第二个元素开始遍历
 for (int i = 1; i < n; i++) {
 // 如果当前元素与上一个保留的元素不同，则保留它
 }
}

```

```
 if (nums[i] != nums[insertPos - 1]) {
 nums[insertPos] = nums[i];
 insertPos++;
 }
 }

 return insertPos;
}

};

/***
 * 打印数组的前 k 个元素
 */
void printArray(const std::vector<int>& nums, int k) {
 std::cout << "[";
 for (int i = 0; i < k; i++) {
 std::cout << nums[i];
 if (i < k - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 打印完整的 vector 数组
 */
void printFullArray(const std::vector<int>& nums) {
 std::cout << "[";
 for (size_t i = 0; i < nums.size(); i++) {
 std::cout << nums[i];
 if (i < nums.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 测试函数
 */
void test() {
 Solution solution;
```

```
// 测试用例 1
std::vector<int> nums1 = {1, 1, 2};
std::cout << "测试用例 1:" << std::endl;
std::cout << "原始数组: ";
printFullArray(nums1);
int length1 = solution.removeDuplicates(nums1);
std::cout << "新长度: " << length1 << std::endl;
std::cout << "新数组前" << length1 << "个元素: ";
printArray(nums1, length1);
std::cout << std::endl;

// 测试用例 2
std::vector<int> nums2 = {0, 0, 1, 1, 1, 2, 2, 3, 3, 4};
std::cout << "测试用例 2:" << std::endl;
std::cout << "原始数组: ";
printFullArray(nums2);
int length2 = solution.removeDuplicates(nums2);
std::cout << "新长度: " << length2 << std::endl;
std::cout << "新数组前" << length2 << "个元素: ";
printArray(nums2, length2);
std::cout << std::endl;

// 测试用例 3 - 边界情况: 空数组
std::vector<int> nums3;
std::cout << "测试用例 3 (空数组) :" << std::endl;
std::cout << "原始数组: []" << std::endl;
int length3 = solution.removeDuplicates(nums3);
std::cout << "新长度: " << length3 << std::endl;
std::cout << "新数组前" << length3 << "个元素: []" << std::endl;
std::cout << std::endl;

// 测试用例 4 - 边界情况: 只有一个元素的数组
std::vector<int> nums4 = {5};
std::cout << "测试用例 4 (单元素数组) :" << std::endl;
std::cout << "原始数组: ";
printFullArray(nums4);
int length4 = solution.removeDuplicates(nums4);
std::cout << "新长度: " << length4 << std::endl;
std::cout << "新数组前" << length4 << "个元素: ";
printArray(nums4, length4);
std::cout << std::endl;
```

```
// 测试用例 5 - 边界情况：没有重复元素的数组
std::vector<int> nums5 = {1, 2, 3, 4, 5};
std::cout << "测试用例 5 (无重复元素数组) :" << std::endl;
std::cout << "原始数组: ";
printFullArray(nums5);
int length5 = solution.removeDuplicates(nums5);
std::cout << "新长度: " << length5 << std::endl;
std::cout << "新数组前" << length5 << "个元素: ";
printArray(nums5, length5);
std::cout << std::endl;
```

```
// 测试用例 6 - 边界情况：所有元素都相同的数组
std::vector<int> nums6 = {3, 3, 3, 3, 3};
std::cout << "测试用例 6 (全相同元素数组) :" << std::endl;
std::cout << "原始数组: ";
printFullArray(nums6);
int length6 = solution.removeDuplicates(nums6);
std::cout << "新长度: " << length6 << std::endl;
std::cout << "新数组前" << length6 << "个元素: ";
printArray(nums6, length6);
std::cout << std::endl;
```

```
}
```

```
/***
 * 性能测试
 */
void performanceTest() {
 Solution solution;
```

```
// 创建一个大数组进行性能测试
int size = 100000;
std::vector<int> largeArray;
largeArray.reserve(size);

// 填充数组，每 10 个元素重复一次
for (int i = 0; i < size; i++) {
 largeArray.push_back(i / 10);
}

// 测试解法一的性能
std::vector<int> array1(largeArray.begin(), largeArray.end());
auto startTime = std::chrono::high_resolution_clock::now();
int result1 = solution.removeDuplicates(array1);
```

```

auto endTime = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法一耗时: " << duration.count() << "ms, 结果长度: " << result1 << std::endl;

// 测试解法二的性能
std::vector<int> array2(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
int result2 = solution.removeDuplicatesOptimized(array2);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法二耗时: " << duration.count() << "ms, 结果长度: " << result2 << std::endl;

// 测试解法三的性能
std::vector<int> array3(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
int result3 = solution.removeDuplicatesGeneric(array3);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法三耗时: " << duration.count() << "ms, 结果长度: " << result3 << std::endl;

// 验证所有解法结果一致
std::cout << "所有解法结果一致: " << (result1 == result2 && result2 == result3 ? "是" : "否")
<< std::endl;
}

int main() {
 std::cout << "==== 测试用例 ===" << std::endl;
 test();

 std::cout << "==== 性能测试 ===" << std::endl;
 performanceTest();

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

文件: Code22\_RemoveDuplicatesFromSortedArray.java

```

/**
 * LeetCode 26. 删除有序数组中的重复项 (Remove Duplicates from Sorted Array)
 *
 * 题目描述:
```

- \* 给你一个升序排列的数组 `nums`, 请你原地删除重复出现的元素, 使得每个元素只出现一次, 返回删除后数组的新长度。
- \* 元素的相对顺序应该保持一致。
- \* 由于在某些语言中不能改变数组的长度, 所以必须将结果放在数组 `nums` 的第一部分。
- \* 更规范地说, 如果在删除重复项之后有  $k$  个元素, 那么 `nums` 的前  $k$  个元素应该保存最终结果。
- \* 将最终结果插入 `nums` 的前  $k$  个位置后返回  $k$ 。
- \* 不要使用额外的空间, 你必须在原地修改输入数组并在  $O(1)$  额外空间的条件下完成。
- \*
- \* 示例 1:
- \* 输入: `nums = [1, 1, 2]`
- \* 输出: `2, nums = [1, 2, _]`
- \* 解释: 函数应该返回新的长度 `2`, 并且原数组 `nums` 的前两个元素被修改为 `1, 2`。
- \* 不需要考虑数组中超出新长度后面的元素。
- \*
- \* 示例 2:
- \* 输入: `nums = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]`
- \* 输出: `5, nums = [0, 1, 2, 3, 4, _, _, _, _, _]`
- \* 解释: 函数应该返回新的长度 `5`, 并且原数组 `nums` 的前五个元素被修改为 `0, 1, 2, 3, 4`。
- \* 不需要考虑数组中超出新长度后面的元素。
- \*
- \* 提示:
- \*  $0 \leq \text{nums.length} \leq 3 * 10^4$
- \*  $-10^4 \leq \text{nums}[i] \leq 10^4$
- \* `nums` 已按升序排列
- \*
- \* 题目链接: <https://leetcode.com/problems/remove-duplicates-from-sorted-array/>
- \*
- \* 解题思路:
- \* 这道题是一个经典的双指针应用场景。由于数组是已排序的, 所有重复的元素都会相邻。
- \* 我们可以使用快慢指针来解决这个问题:
- \* 1. 慢指针 `slow` 指向当前已处理好的不重复元素的最后一个位置
- \* 2. 快指针 `fast` 遍历整个数组
- \* 3. 当 `nums[fast]` 不等于 `nums[slow]` 时, 说明找到了一个新的不重复元素, 将 `slow` 向前移动一位, 然后将 `nums[fast]` 赋值给 `nums[slow]`
- \* 4. 当 `nums[fast]` 等于 `nums[slow]` 时, 说明遇到了重复元素, `fast` 继续向前移动
- \* 5. 遍历结束后, `slow+1` 就是新数组的长度
- \*
- \* 时间复杂度:  $O(n)$ , 其中  $n$  是数组的长度。快指针最多遍历数组一次。
- \* 空间复杂度:  $O(1)$ , 只使用了常数级别的额外空间。
- \*/

```
import java.util.Arrays;
```

```
public class Code22_RemoveDuplicatesFromSortedArray {

 /**
 * 解法一：快慢指针（最优解）
 *
 * @param nums 升序排列的数组
 * @return 删除重复元素后的数组新长度
 */

 public static int removeDuplicates(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 // 边界情况：数组长度为 0 或 1，直接返回原长度
 if (nums.length <= 1) {
 return nums.length;
 }

 int slow = 0; // 慢指针，指向当前已处理好的不重复元素的最后一个位置

 // 快指针遍历整个数组
 for (int fast = 1; fast < nums.length; fast++) {
 // 当遇到不同的元素时
 if (nums[fast] != nums[slow]) {
 slow++; // 慢指针向前移动一位
 nums[slow] = nums[fast]; // 将不重复的元素移动到前面
 }
 // 当遇到相同的元素时，fast 继续向前移动，slow 保持不变
 }

 // 新数组的长度是 slow + 1
 return slow + 1;
 }

 /**
 * 解法二：快慢指针的另一种写法，包含更多优化
 *
 * @param nums 升序排列的数组
 * @return 删除重复元素后的数组新长度
 */

 public static int removeDuplicatesOptimized(int[] nums) {
 // 参数校验
```

```
if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
}

// 边界情况：数组长度为 0 或 1，直接返回原长度
int n = nums.length;
if (n <= 1) {
 return n;
}

int slow = 1; // 这里 slow 初始化为 1，表示第一个元素已经是唯一的

// 从第二个元素开始遍历
for (int fast = 1; fast < n; fast++) {
 // 当遇到不同的元素时
 if (nums[fast] != nums[fast - 1]) {
 nums[slow] = nums[fast]; // 将不重复的元素移动到前面
 slow++; // 慢指针向前移动一位
 }
}

return slow;
}

/**
 * 解法三：优化版快慢指针，适用于更通用的场景
 *
 * @param nums 升序排列的数组
 * @return 删除重复元素后的数组新长度
 */
public static int removeDuplicatesGeneric(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 // 边界情况：数组长度为 0，直接返回 0
 int n = nums.length;
 if (n == 0) {
 return 0;
 }

 // 初始化慢指针
 int slow = 1;
```

```
int insertPos = 1;

// 从第二个元素开始遍历
for (int i = 1; i < n; i++) {
 // 如果当前元素与上一个保留的元素不同，则保留它
 if (nums[i] != nums[insertPos - 1]) {
 nums[insertPos] = nums[i];
 insertPos++;
 }
}

return insertPos;
}

/***
 * 打印数组的前 k 个元素
 *
 * @param nums 数组
 * @param k 元素个数
 */
public static void printArray(int[] nums, int k) {
 System.out.print("[");
 for (int i = 0; i < k; i++) {
 System.out.print(nums[i]);
 if (i < k - 1) {
 System.out.print(", ");
 }
 }
 System.out.println("]");
}

/***
 * 测试函数
 */
public static void test() {
 // 测试用例 1
 int[] nums1 = {1, 1, 2};
 System.out.println("测试用例 1:");
 System.out.println("原始数组: " + Arrays.toString(nums1));
 int length1 = removeDuplicates(nums1);
 System.out.println("新长度: " + length1);
 System.out.print("新数组前" + length1 + "个元素: ");
 printArray(nums1, length1);
}
```

```
System.out.println();

// 测试用例 2
int[] nums2 = {0, 0, 1, 1, 1, 2, 2, 3, 3, 4};
System.out.println("测试用例 2:");
System.out.println("原始数组: " + Arrays.toString(nums2));
int length2 = removeDuplicates(nums2);
System.out.println("新长度: " + length2);
System.out.print("新数组前" + length2 + "个元素: ");
printArray(nums2, length2);
System.out.println();

// 测试用例 3 - 边界情况: 空数组
int[] nums3 = {};
System.out.println("测试用例 3 (空数组):");
System.out.println("原始数组: " + Arrays.toString(nums3));
int length3 = removeDuplicates(nums3);
System.out.println("新长度: " + length3);
System.out.print("新数组前" + length3 + "个元素: ");
printArray(nums3, length3);
System.out.println();

// 测试用例 4 - 边界情况: 只有一个元素的数组
int[] nums4 = {5};
System.out.println("测试用例 4 (单元素数组):");
System.out.println("原始数组: " + Arrays.toString(nums4));
int length4 = removeDuplicates(nums4);
System.out.println("新长度: " + length4);
System.out.print("新数组前" + length4 + "个元素: ");
printArray(nums4, length4);
System.out.println();

// 测试用例 5 - 边界情况: 没有重复元素的数组
int[] nums5 = {1, 2, 3, 4, 5};
System.out.println("测试用例 5 (无重复元素数组):");
System.out.println("原始数组: " + Arrays.toString(nums5));
int length5 = removeDuplicates(nums5);
System.out.println("新长度: " + length5);
System.out.print("新数组前" + length5 + "个元素: ");
printArray(nums5, length5);
System.out.println();

// 测试用例 6 - 边界情况: 所有元素都相同的数组
```

```
int[] nums6 = {3, 3, 3, 3, 3};
System.out.println("测试用例 6 (全相同元素数组) :");
System.out.println("原始数组: " + Arrays.toString(nums6));
int length6 = removeDuplicates(nums6);
System.out.println("新长度: " + length6);
System.out.print("新数组前" + length6 + "个元素: ");
printArray(nums6, length6);
}

/**
 * 主函数
 */
public static void main(String[] args) {
 test();

 // 性能测试
 System.out.println("\n性能测试:");

 // 创建一个大数组进行性能测试
 int size = 100000;
 int[] largeArray = new int[size];
 // 填充数组, 每 10 个元素重复一次
 for (int i = 0; i < size; i++) {
 largeArray[i] = i / 10;
 }

 // 测试解法一的性能
 int[] array1 = Arrays.copyOf(largeArray, size);
 long startTime = System.currentTimeMillis();
 int result1 = removeDuplicates(array1);
 long endTime = System.currentTimeMillis();
 System.out.println("解法一耗时: " + (endTime - startTime) + "ms, 结果长度: " + result1);

 // 测试解法二的性能
 int[] array2 = Arrays.copyOf(largeArray, size);
 startTime = System.currentTimeMillis();
 int result2 = removeDuplicatesOptimized(array2);
 endTime = System.currentTimeMillis();
 System.out.println("解法二耗时: " + (endTime - startTime) + "ms, 结果长度: " + result2);

 // 测试解法三的性能
 int[] array3 = Arrays.copyOf(largeArray, size);
 startTime = System.currentTimeMillis();
```

```
 int result3 = removeDuplicatesGeneric(array3);
 endTime = System.currentTimeMillis();
 System.out.println("解法三耗时: " + (endTime - startTime) + "ms, 结果长度: " + result3);
}
=====
```

文件: Code22\_RemoveDuplicatesFromSortedArray.py

```
=====
import time
from typing import List
```

```
"""
LeetCode 26. 删除有序数组中的重复项 (Remove Duplicates from Sorted Array)
```

题目描述:

给你一个升序排列的数组 `nums`, 请你原地删除重复出现的元素, 使得每个元素只出现一次, 返回删除后数组的新长度。

元素的相对顺序应该保持一致。

由于在某些语言中不能改变数组的长度, 所以必须将结果放在数组 `nums` 的第一部分。

更规范地说, 如果在删除重复项之后有  $k$  个元素, 那么 `nums` 的前  $k$  个元素应该保存最终结果。

将最终结果插入 `nums` 的前  $k$  个位置后返回  $k$ 。

不要使用额外的空间, 你必须在原地修改输入数组并在  $O(1)$  额外空间的条件下完成。

示例 1:

输入: `nums` = [1, 1, 2]

输出: 2, `nums` = [1, 2, \_]

解释: 函数应该返回新的长度 2, 并且原数组 `nums` 的前两个元素被修改为 1, 2。

不需要考虑数组中超出新长度后面的元素。

示例 2:

输入: `nums` = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]

输出: 5, `nums` = [0, 1, 2, 3, 4, \_, \_, \_, \_, \_]

解释: 函数应该返回新的长度 5, 并且原数组 `nums` 的前五个元素被修改为 0, 1, 2, 3, 4。

不需要考虑数组中超出新长度后面的元素。

提示:

$0 \leq \text{nums.length} \leq 3 * 10^4$

$-10^4 \leq \text{nums}[i] \leq 10^4$

`nums` 已按升序排列

题目链接: <https://leetcode.com/problems/remove-duplicates-from-sorted-array/>

解题思路：

这道题是一个经典的双指针应用场景。由于数组是已排序的，所有重复的元素都会相邻。

我们可以使用快慢指针来解决这个问题：

1. 慢指针 slow 指向当前已处理好的不重复元素的最后一个位置
2. 快指针 fast 遍历整个数组
3. 当 `nums[fast]` 不等于 `nums[slow]` 时，说明找到了一个新的不重复元素，将 `slow` 向前移动一位，然后将 `nums[fast]` 赋值给 `nums[slow]`
4. 当 `nums[fast]` 等于 `nums[slow]` 时，说明遇到了重复元素，`fast` 继续向前移动
5. 遍历结束后，`slow+1` 就是新数组的长度

时间复杂度：O(n)，其中 n 是数组的长度。快指针最多遍历数组一次。

空间复杂度：O(1)，只使用了常数级别的额外空间。

"""

```
class Solution:
```

```
 def remove_duplicates(self, nums: List[int]) -> int:
```

```
 """
```

```
 LeetCode 官方接口的实现（使用快慢指针最优解）
```

Args:

    nums: 升序排列的数组

Returns:

    删除重复元素后的数组新长度

Raises:

    TypeError: 如果输入不是列表

```
 """
```

# 参数校验

```
 if not isinstance(nums, list):
 raise TypeError("输入必须是列表类型")
```

# 边界情况：数组长度为 0 或 1，直接返回原长度

```
 n = len(nums)
 if n <= 1:
 return n
```

```
 slow = 0 # 慢指针，指向当前已处理好的不重复元素的最后一个位置
```

# 快指针遍历整个数组

```
 for fast in range(1, n):
 # 当遇到不同的元素时
```

```
 if nums[fast] != nums[slow]:
 slow += 1 # 慢指针向前移动一位
 nums[slow] = nums[fast] # 将不重复的元素移动到前面
 # 当遇到相同的元素时, fast 继续向前移动, slow 保持不变

新数组的长度是 slow + 1
return slow + 1
```

```
def remove_duplicates_optimized(self, nums: List[int]) -> int:
 """
```

解法二：快慢指针的另一种写法，包含更多优化

Args:

    nums: 升序排列的数组

Returns:

    删除重复元素后的数组新长度

```
"""
```

# 边界情况：数组长度为 0 或 1，直接返回原长度

n = len(nums)

if n <= 1:

    return n

slow = 1 # 这里 slow 初始化为 1，表示第一个元素已经是唯一的

# 从第二个元素开始遍历

for fast in range(1, n):

# 当遇到不同的元素时

if nums[fast] != nums[fast - 1]:

nums[slow] = nums[fast] # 将不重复的元素移动到前面

slow += 1 # 慢指针向前移动一位

return slow

```
def remove_duplicates_generic(self, nums: List[int]) -> int:
 """
```

解法三：优化版快慢指针，适用于更通用的场景

Args:

    nums: 升序排列的数组

Returns:

    删除重复元素后的数组新长度

```
"""
边界情况：数组长度为 0， 直接返回 0
n = len(nums)
if n == 0:
 return 0

初始化慢指针
insert_pos = 1

从第二个元素开始遍历
for i in range(1, n):
 # 如果当前元素与上一个保留的元素不同，则保留它
 if nums[i] != nums[insert_pos - 1]:
 nums[insert_pos] = nums[i]
 insert_pos += 1

return insert_pos
```

```
"""
```

```
打印数组的前 k 个元素
```

```
"""
def print_array(nums: List[int], k: int) -> None:
 print(f"[{', '.join(map(str, nums[:k]))}]")
```

```
"""
打印完整的数组
"""
def print_full_array(nums: List[int]) -> None:
 print(f"[{', '.join(map(str, nums))}]")
```

```
"""
测试函数
"""
def test():
 solution = Solution()
```

```
测试用例 1
nums1 = [1, 1, 2]
print("测试用例 1:")
print("原始数组: ", end="")
print_full_array(nums1)
length1 = solution.remove_duplicates(nums1)
print(f"新长度: {length1}")
```

```
print(f"新数组前{length1}个元素: ", end="")
print_array(nums1, length1)
print()
```

```
测试用例 2
nums2 = [0, 0, 1, 1, 1, 2, 2, 3, 3, 4]
print("测试用例 2:")
print("原始数组: ", end="")
print_full_array(nums2)
length2 = solution.remove_duplicates(nums2)
print(f"新长度: {length2}")
print(f"新数组前{length2}个元素: ", end="")
print_array(nums2, length2)
print()
```

```
测试用例 3 - 边界情况: 空数组
nums3 = []
print("测试用例 3 (空数组) :")
print("原始数组: []")
length3 = solution.remove_duplicates(nums3)
print(f"新长度: {length3}")
print(f"新数组前{length3}个元素: []")
print()
```

```
测试用例 4 - 边界情况: 只有一个元素的数组
nums4 = [5]
print("测试用例 4 (单元素数组) :")
print("原始数组: ", end="")
print_full_array(nums4)
length4 = solution.remove_duplicates(nums4)
print(f"新长度: {length4}")
print(f"新数组前{length4}个元素: ", end="")
print_array(nums4, length4)
print()
```

```
测试用例 5 - 边界情况: 没有重复元素的数组
nums5 = [1, 2, 3, 4, 5]
print("测试用例 5 (无重复元素数组) :")
print("原始数组: ", end="")
print_full_array(nums5)
length5 = solution.remove_duplicates(nums5)
print(f"新长度: {length5}")
print(f"新数组前{length5}个元素: ", end="")
```

```
print_array(nums5, length5)
print()

测试用例 6 - 边界情况: 所有元素都相同的数组
nums6 = [3, 3, 3, 3, 3]
print("测试用例 6 (全相同元素数组) :")
print("原始数组: ", end="")
print_full_array(nums6)
length6 = solution.remove_duplicates(nums6)
print(f"新长度: {length6}")
print(f"新数组前{length6}个元素: ", end="")
print_array(nums6, length6)
print()
```

"""

## 性能测试

"""

```
def performance_test():
 solution = Solution()

 # 创建一个大数组进行性能测试
 size = 100000
 large_array = [i // 10 for i in range(size)]

 # 测试解法一的性能
 array1 = large_array.copy()
 start_time = time.time()
 result1 = solution.remove_duplicates(array1)
 end_time = time.time()
 print(f"解法一耗时: {((end_time - start_time) * 1000:.2f}ms, 结果长度: {result1}")

 # 测试解法二的性能
 array2 = large_array.copy()
 start_time = time.time()
 result2 = solution.remove_duplicates_optimized(array2)
 end_time = time.time()
 print(f"解法二耗时: {((end_time - start_time) * 1000:.2f}ms, 结果长度: {result2}")

 # 测试解法三的性能
 array3 = large_array.copy()
 start_time = time.time()
 result3 = solution.remove_duplicates_generic(array3)
 end_time = time.time()
```

```

print(f"解法三耗时: {(end_time - start_time) * 1000:.2f}ms, 结果长度: {result3}")

验证所有解法结果一致
print(f"所有解法结果一致: {result1 == result2 and result2 == result3}")

"""
主函数
"""

def main():
 print("== 测试用例 ==")
 test()

 print("== 性能测试 ==")
 performance_test()

if __name__ == "__main__":
 main()

```

=====

文件: Code23\_RemoveElement.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

/***
 * LeetCode 27. 移除元素 (Remove Element)
 *
 * 题目描述:
 * 给你一个数组 nums 和一个值 val，你需要原地移除所有数值等于 val 的元素，并返回移除后数组的新长度。
 *
 * 不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并原地修改输入数组。
 *
 * 元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。
 *
 * 示例 1:
 * 输入: nums = [3, 2, 2, 3], val = 3
 * 输出: 2, nums = [2, 2]
 * 解释: 函数应该返回新的长度 2, 并且 nums 中的前两个元素均为 2。
 * 不需要考虑数组中超出新长度后面的元素。例如，函数返回的新长度为 2 ,
 * 而 nums = [2, 2, 3, 3] 或 nums = [2, 2, 0, 0]，也会被视作正确答案。
 *

```

\* 示例 2:

\* 输入: nums = [0, 1, 2, 2, 3, 0, 4, 2], val = 2

\* 输出: 5, nums = [0, 1, 4, 0, 3]

\* 解释: 函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。

\* 注意这五个元素可为任意顺序。你不需要考虑数组中超出新长度后面的元素。

\*

\* 提示:

\*  $0 \leq \text{nums.length} \leq 100$

\*  $0 \leq \text{nums}[i] \leq 50$

\*  $0 \leq \text{val} \leq 100$

\*

\* 题目链接: <https://leetcode.com/problems/remove-element/>

\*

\* 解题思路:

\* 这道题与删除有序数组中的重复项类似, 都可以使用双指针技术。

\* 我们可以使用一个慢指针来跟踪应该放置下一个非 val 元素的位置,

\* 然后用一个快指针来遍历整个数组。

\* 当快指针找到一个不等于 val 的元素时, 我们将该元素放到慢指针指向的位置, 然后将慢指针向前移动一位。

\*

\* 时间复杂度:  $O(n)$ , 其中  $n$  是数组的长度。快指针最多遍历数组一次。

\* 空间复杂度:  $O(1)$ , 只使用了常数级别的额外空间。

\*/

```
class Solution {
public:
 /**
 * 解法一: 双指针 (最优解)
 *
 * @param nums 输入数组
 * @param val 要移除的元素值
 * @return 移除后数组的新长度
 */
 int removeElement(std::vector<int>& nums, int val) {
 int slow = 0; // 慢指针, 指向下一个非 val 元素应该放置的位置

 // 快指针遍历整个数组
 for (int fast = 0; fast < nums.size(); fast++) {
 // 如果当前元素不等于 val, 则将其移到慢指针位置, 并将慢指针向前移动
 if (nums[fast] != val) {
 nums[slow] = nums[fast];
 slow++;
 }
 }
 }
}
```

```
}

// 慢指针的值就是新数组的长度
return slow;
}

/***
 * 解法二：优化的双指针（当需要删除的元素很少时）
 *
 * 当要删除的元素很少时，我们可以将不等于 val 的元素保留，而将等于 val 的元素与数组末尾的元素交换，
 * 然后减少数组的长度。这样可以减少不必要的赋值操作。
 *
 * @param nums 输入数组
 * @param val 要移除的元素值
 * @return 移除后数组的新长度
 */
int removeElementOptimized(std::vector<int>& nums, int val) {
 int left = 0;
 int right = nums.size();

 while (left < right) {
 if (nums[left] == val) {
 // 将当前元素与数组末尾元素交换
 nums[left] = nums[right - 1];
 // 减少数组长度
 right--;
 } else {
 // 当前元素不等于 val，保留它
 left++;
 }
 }

 return right;
}

/***
 * 解法三：简洁版双指针
 *
 * @param nums 输入数组
 * @param val 要移除的元素值
 * @return 移除后数组的新长度
 */
```

```
int removeElementConcise(std::vector<int>& nums, int val) {
 int index = 0;

 for (int i = 0; i < nums.size(); i++) {
 if (nums[i] != val) {
 nums[index++] = nums[i];
 }
 }

 return index;
}

/***
 * 打印数组的前 k 个元素
 */
void printArray(const std::vector<int>& nums, int k) {
 std::cout << "[";
 for (int i = 0; i < k; i++) {
 std::cout << nums[i];
 if (i < k - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 打印完整的 vector 数组
 */
void printFullArray(const std::vector<int>& nums) {
 std::cout << "[";
 for (size_t i = 0; i < nums.size(); i++) {
 std::cout << nums[i];
 if (i < nums.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 测试函数
*/
```

```
*/
void test() {
 Solution solution;

 // 测试用例 1
 std::vector<int> nums1 = {3, 2, 2, 3};
 int val1 = 3;
 std::cout << "测试用例 1:" << std::endl;
 std::cout << "原始数组: ";
 printFullArray(nums1);
 std::cout << "要移除的值: " << val1 << std::endl;
 int length1 = solution.removeElement(nums1, val1);
 std::cout << "新长度: " << length1 << std::endl;
 std::cout << "新数组前" << length1 << "个元素: ";
 printArray(nums1, length1);
 std::cout << std::endl;

 // 测试用例 2
 std::vector<int> nums2 = {0, 1, 2, 2, 3, 0, 4, 2};
 int val2 = 2;
 std::cout << "测试用例 2:" << std::endl;
 std::cout << "原始数组: ";
 printFullArray(nums2);
 std::cout << "要移除的值: " << val2 << std::endl;
 int length2 = solution.removeElement(nums2, val2);
 std::cout << "新长度: " << length2 << std::endl;
 std::cout << "新数组前" << length2 << "个元素: ";
 printArray(nums2, length2);
 std::cout << std::endl;

 // 测试用例 3 - 边界情况: 空数组
 std::vector<int> nums3;
 int val3 = 0;
 std::cout << "测试用例 3 (空数组) :" << std::endl;
 std::cout << "原始数组: []" << std::endl;
 std::cout << "要移除的值: " << val3 << std::endl;
 int length3 = solution.removeElement(nums3, val3);
 std::cout << "新长度: " << length3 << std::endl;
 std::cout << "新数组前" << length3 << "个元素: []" << std::endl;
 std::cout << std::endl;

 // 测试用例 4 - 边界情况: 所有元素都等于 val
 std::vector<int> nums4 = {5, 5, 5, 5};
```

```
int val4 = 5;
std::cout << "测试用例 4 (全等于 val 的数组) :" << std::endl;
std::cout << "原始数组: ";
printFullArray(nums4);
std::cout << "要移除的值: " << val4 << std::endl;
int length4 = solution.removeElement(nums4, val4);
std::cout << "新长度: " << length4 << std::endl;
std::cout << "新数组前" << length4 << "个元素: []" << std::endl;
std::cout << std::endl;

// 测试用例 5 - 边界情况: 没有元素等于 val
std::vector<int> nums5 = {1, 2, 3, 4, 5};
int val5 = 6;
std::cout << "测试用例 5 (无等于 val 的元素) :" << std::endl;
std::cout << "原始数组: ";
printFullArray(nums5);
std::cout << "要移除的值: " << val5 << std::endl;
int length5 = solution.removeElement(nums5, val5);
std::cout << "新长度: " << length5 << std::endl;
std::cout << "新数组前" << length5 << "个元素: ";
printArray(nums5, length5);
std::cout << std::endl;
}
```

```
/***
 * 性能测试
 */
void performanceTest() {
 Solution solution;

 // 创建一个大数组进行性能测试
 int size = 1000000;
 std::vector<int> largeArray;
 largeArray.reserve(size);

 // 填充数组, 其中约 20%的元素等于 val
 int val = 5;
 for (int i = 0; i < size; i++) {
 largeArray.push_back(i % 10 == 0 ? val : i % 10);
 }

 // 测试解法一的性能
 std::vector<int> array1(largeArray.begin(), largeArray.end());
```

```

auto startTime = std::chrono::high_resolution_clock::now();
int result1 = solution.removeElement(array1, val);
auto endTime = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法一耗时: " << duration.count() << "ms, 结果长度: " << result1 << std::endl;

// 测试解法二的性能
std::vector<int> array2(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
int result2 = solution.removeElementOptimized(array2, val);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法二耗时: " << duration.count() << "ms, 结果长度: " << result2 << std::endl;

// 测试解法三的性能
std::vector<int> array3(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
int result3 = solution.removeElementConcise(array3, val);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法三耗时: " << duration.count() << "ms, 结果长度: " << result3 << std::endl;

// 验证所有解法结果一致
std::cout << "所有解法结果一致: " << (result1 == result2 && result2 == result3 ? "是" : "否")
<< std::endl;
}

int main() {
 std::cout << "==== 测试用例 ===" << std::endl;
 test();

 std::cout << "==== 性能测试 ===" << std::endl;
 performanceTest();

 return 0;
}

```

文件: Code23\_RemoveElement.java

```
=====
package class050;
```

```
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

/**
 * LeetCode 27. 移除元素 (Remove Element)
 *
 * 题目描述:
 * 给你一个数组 nums 和一个值 val，你需要原地移除所有数值等于 val 的元素，并返回移除后数组的新长度。
 *
 * 不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并原地修改输入数组。
 * 元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。
 *
 * 示例 1:
 * 输入: nums = [3, 2, 2, 3], val = 3
 * 输出: 2, nums = [2, 2]
 * 解释: 函数应该返回新的长度 2, 并且 nums 中的前两个元素均为 2。
 * 不需要考虑数组中超出新长度后面的元素。例如，函数返回的新长度为 2 ,
 * 而 nums = [2, 2, 3, 3] 或 nums = [2, 2, 0, 0]，也会被视作正确答案。
 *
 * 示例 2:
 * 输入: nums = [0, 1, 2, 2, 3, 0, 4, 2], val = 2
 * 输出: 5, nums = [0, 1, 4, 0, 3]
 * 解释: 函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。
 * 注意这五个元素可为任意顺序。你不需要考虑数组中超出新长度后面的元素。
 *
 * 提示:
 * 0 <= nums.length <= 100
 * 0 <= nums[i] <= 50
 * 0 <= val <= 100
 *
 * 题目链接: https://leetcode.com/problems/remove-element/
 *
 * 解题思路:
 * 这道题与删除有序数组中的重复项类似，都可以使用双指针技术。
 * 我们可以使用一个慢指针来跟踪应该放置下一个非 val 元素的位置，
 * 然后用一个快指针来遍历整个数组。
 * 当快指针找到一个不等于 val 的元素时，我们将该元素放到慢指针指向的位置，然后将慢指针向前移动一位。
 *
 * 时间复杂度: O(n)，其中 n 是数组的长度。快指针最多遍历数组一次。
 * 空间复杂度: O(1)，只使用了常数级别的额外空间。
 */

```

```
public class Code23_RemoveElement {

 /**
 * 解法一：双指针（最优解）
 *
 * @param nums 输入数组
 * @param val 要移除的元素值
 * @return 移除后数组的新长度
 * @throws IllegalArgumentException 如果输入数组为 null
 */
 public static int removeElement(int[] nums, int val) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 int slow = 0; // 慢指针，指向下一个非 val 元素应该放置的位置

 // 快指针遍历整个数组
 for (int fast = 0; fast < nums.length; fast++) {
 // 如果当前元素不等于 val，则将其移到慢指针位置，并将慢指针向前移动
 if (nums[fast] != val) {
 nums[slow] = nums[fast];
 slow++;
 }
 }

 // 慢指针的值就是新数组的长度
 return slow;
 }

 /**
 * 解法二：优化的双指针（当需要删除的元素很少时）
 *
 * 当要删除的元素很少时，我们可以将不等于 val 的元素保留，而将等于 val 的元素与数组末尾的元素交换，
 * 然后减少数组的长度。这样可以减少不必要的赋值操作。
 *
 * @param nums 输入数组
 * @param val 要移除的元素值
 * @return 移除后数组的新长度
 */
 public static int removeElementOptimized(int[] nums, int val) {
```

```
// 参数校验
if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
}

int left = 0;
int right = nums.length;

while (left < right) {
 if (nums[left] == val) {
 // 将当前元素与数组末尾元素交换
 nums[left] = nums[right - 1];
 // 减少数组长度
 right--;
 } else {
 // 当前元素不等于 val, 保留它
 left++;
 }
}

return right;
}

/**
 * 解法三：简洁版双指针
 *
 * @param nums 输入数组
 * @param val 要移除的元素值
 * @return 移除后数组的新长度
 */
public static int removeElementConcise(int[] nums, int val) {
 int index = 0;

 for (int i = 0; i < nums.length; i++) {
 if (nums[i] != val) {
 nums[index++] = nums[i];
 }
 }

 return index;
}

/**

```

```
* 打印数组的前 k 个元素
*/
public static void printArray(int[] nums, int k) {
 System.out.print("[");
 for (int i = 0; i < k; i++) {
 System.out.print(nums[i]);
 if (i < k - 1) {
 System.out.print(", ");
 }
 }
 System.out.println("]");
}

/***
 * 打印完整的数组
*/
public static void printFullArray(int[] nums) {
 System.out.println(Arrays.toString(nums));
}

/***
 * 测试函数
*/
public static void test() {
 // 测试用例 1
 int[] nums1 = {3, 2, 2, 3};
 int val1 = 3;
 System.out.println("测试用例 1:");
 System.out.print("原始数组: ");
 printFullArray(nums1);
 System.out.println("要移除的值: " + val1);
 int length1 = removeElement(nums1, val1);
 System.out.println("新长度: " + length1);
 System.out.print("新数组前" + length1 + "个元素: ");
 printArray(nums1, length1);
 System.out.println();

 // 测试用例 2
 int[] nums2 = {0, 1, 2, 2, 3, 0, 4, 2};
 int val2 = 2;
 System.out.println("测试用例 2:");
 System.out.print("原始数组: ");
 printFullArray(nums2);
```

```
System.out.println("要移除的值: " + val2);
int length2 = removeElement(nums2, val2);
System.out.println("新长度: " + length2);
System.out.print("新数组前" + length2 + "个元素: ");
printArray(nums2, length2);
System.out.println();

// 测试用例 3 - 边界情况: 空数组
int[] nums3 = {};
int val3 = 0;
System.out.println("测试用例 3 (空数组) :");
System.out.print("原始数组: []");
System.out.println("要移除的值: " + val3);
int length3 = removeElement(nums3, val3);
System.out.println("新长度: " + length3);
System.out.print("新数组前" + length3 + "个元素: []");
System.out.println();

// 测试用例 4 - 边界情况: 所有元素都等于 val
int[] nums4 = {5, 5, 5, 5};
int val4 = 5;
System.out.println("测试用例 4 (全等于 val 的数组) :");
System.out.print("原始数组: ");
printFullArray(nums4);
System.out.println("要移除的值: " + val4);
int length4 = removeElement(nums4, val4);
System.out.println("新长度: " + length4);
System.out.print("新数组前" + length4 + "个元素: []");
System.out.println();

// 测试用例 5 - 边界情况: 没有元素等于 val
int[] nums5 = {1, 2, 3, 4, 5};
int val5 = 6;
System.out.println("测试用例 5 (无等于 val 的元素) :");
System.out.print("原始数组: ");
printFullArray(nums5);
System.out.println("要移除的值: " + val5);
int length5 = removeElement(nums5, val5);
System.out.println("新长度: " + length5);
System.out.print("新数组前" + length5 + "个元素: ");
printArray(nums5, length5);
System.out.println();
}
```

```
/**
 * 性能测试
 */
public static void performanceTest() {
 // 创建一个大数组进行性能测试
 int size = 1000000;
 int[] largeArray = new int[size];

 // 填充数组，其中约 20% 的元素等于 val
 int val = 5;
 for (int i = 0; i < size; i++) {
 largeArray[i] = i % 10 == 0 ? val : i % 10;
 }

 // 测试解法一的性能
 int[] array1 = Arrays.copyOf(largeArray, size);
 long startTime = System.nanoTime();
 int result1 = removeElement(array1, val);
 long endTime = System.nanoTime();
 long duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
 System.out.println("解法一耗时: " + duration + "ms, 结果长度: " + result1);

 // 测试解法二的性能
 int[] array2 = Arrays.copyOf(largeArray, size);
 startTime = System.nanoTime();
 int result2 = removeElementOptimized(array2, val);
 endTime = System.nanoTime();
 duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
 System.out.println("解法二耗时: " + duration + "ms, 结果长度: " + result2);

 // 测试解法三的性能
 int[] array3 = Arrays.copyOf(largeArray, size);
 startTime = System.nanoTime();
 int result3 = removeElementConcise(array3, val);
 endTime = System.nanoTime();
 duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
 System.out.println("解法三耗时: " + duration + "ms, 结果长度: " + result3);

 // 验证所有解法结果一致
 System.out.println("所有解法结果一致: " + (result1 == result2 && result2 == result3));
}
```

```
public static void main(String[] args) {
 System.out.println("==> 测试用例 ==>");
 test();

 System.out.println("==> 性能测试 ==>");
 performanceTest();
}
}
```

=====

文件: Code23\_RemoveElement.py

=====

```
import time
from typing import List

"""
LeetCode 27. 移除元素 (Remove Element)
"""
```

题目描述:

给你一个数组 `nums` 和一个值 `val`，你需要原地移除所有数值等于 `val` 的元素，并返回移除后数组的新长度。  
不要使用额外的数组空间，你必须仅使用  $O(1)$  额外空间并原地修改输入数组。  
元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

示例 1:

输入: `nums = [3, 2, 2, 3]`, `val = 3`

输出: 2, `nums = [2, 2]`

解释: 函数应该返回新的长度 2，并且 `nums` 中的前两个元素均为 2。

不需要考虑数组中超出新长度后面的元素。例如，函数返回的新长度为 2，  
而 `nums = [2, 2, 3, 3]` 或 `nums = [2, 2, 0, 0]`，也会被视作正确答案。

示例 2:

输入: `nums = [0, 1, 2, 2, 3, 0, 4, 2]`, `val = 2`

输出: 5, `nums = [0, 1, 4, 0, 3]`

解释: 函数应该返回新的长度 5，并且 `nums` 中的前五个元素为 0, 1, 3, 0, 4。

注意这五个元素可为任意顺序。你不需要考虑数组中超出新长度后面的元素。

提示:

$0 \leq \text{nums.length} \leq 100$

$0 \leq \text{nums}[i] \leq 50$

$0 \leq \text{val} \leq 100$

题目链接: <https://leetcode.com/problems/remove-element/>

解题思路：

这道题与删除有序数组中的重复项类似，都可以使用双指针技术。

我们可以使用一个慢指针来跟踪应该放置下一个非 val 元素的位置，

然后用一个快指针来遍历整个数组。

当快指针找到一个不等于 val 的元素时，我们将该元素放到慢指针指向的位置，然后将慢指针向前移动一位。

时间复杂度：O(n)，其中 n 是数组的长度。快指针最多遍历数组一次。

空间复杂度：O(1)，只使用了常数级别的额外空间。

"""

```
class Solution:
```

```
 def remove_element(self, nums: List[int], val: int) -> int:
```

```
 """
```

解法一：双指针（最优解）

Args:

nums: 输入数组

val: 要移除的元素值

Returns:

移除后数组的新长度

Raises:

TypeError: 如果输入不是列表类型

```
 """
```

# 参数校验

```
if not isinstance(nums, list):
```

```
 raise TypeError("输入必须是列表类型")
```

```
slow = 0 # 慢指针，指向下一个非 val 元素应该放置的位置
```

# 快指针遍历整个数组

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

# 如果当前元素不等于 val，则将其移到慢指针位置，并将慢指针向前移动

```
 if nums[fast] != val:
```

```
 nums[slow] = nums[fast]
```

```
 slow += 1
```

# 慢指针的值就是新数组的长度

```
return slow
```

```
def remove_element_optimized(self, nums: List[int], val: int) -> int:
```

```
"""
```

## 解法二：优化的双指针（当需要删除的元素很少时）

当要删除的元素很少时，我们可以将不等于 val 的元素保留，而将等于 val 的元素与数组末尾的元素交换，

然后减少数组的长度。这样可以减少不必要的赋值操作。

Args:

nums: 输入数组

val: 要移除的元素值

Returns:

移除后数组的新长度

```
"""
```

left = 0

right = len(nums)

while left < right:

if nums[left] == val:

# 将当前元素与数组末尾元素交换

nums[left] = nums[right - 1]

# 减少数组长度

right -= 1

else:

# 当前元素不等于 val，保留它

left += 1

return right

```
def remove_element_concise(self, nums: List[int], val: int) -> int:
```

```
"""
```

## 解法三：简洁版双指针

Args:

nums: 输入数组

val: 要移除的元素值

Returns:

移除后数组的新长度

```
"""
```

index = 0

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

```
 if nums[i] != val:
 nums[index] = nums[i]
 index += 1

 return index
```

```
def removeElement(self, nums: List[int], val: int) -> int:
```

```
 """
```

```
 LeetCode 官方接口的实现（使用双指针最优解）
```

Args:

```
 nums: 输入数组
 val: 要移除的元素值
```

Returns:

```
 移除后数组的新长度
```

```
 """
```

```
slow = 0
```

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

```
 if nums[fast] != val:
 nums[slow] = nums[fast]
 slow += 1
```

```
return slow
```

```
"""
```

```
打印数组的前 k 个元素
```

```
"""
```

```
def print_array(nums: List[int], k: int) -> None:
```

```
 print(f"[{', '.join(map(str, nums[:k]))}]")
```

```
"""
```

```
打印完整的数组
```

```
"""
```

```
def print_full_array(nums: List[int]) -> None:
```

```
 print(f"[{', '.join(map(str, nums))}]")
```

```
"""
```

```
测试函数
```

```
"""
```

```
def test():
```

```
 solution = Solution()
```

```
测试用例 1
nums1 = [3, 2, 2, 3]
val1 = 3
print("测试用例 1:")
print("原始数组: ", end="")
print_full_array(nums1)
print(f"要移除的值: {val1}")
length1 = solution.remove_element(nums1, val1)
print(f"新长度: {length1}")
print(f"新数组前{length1}个元素: ", end="")
print_array(nums1, length1)
print()
```

```
测试用例 2
nums2 = [0, 1, 2, 2, 3, 0, 4, 2]
val2 = 2
print("测试用例 2:")
print("原始数组: ", end="")
print_full_array(nums2)
print(f"要移除的值: {val2}")
length2 = solution.remove_element(nums2, val2)
print(f"新长度: {length2}")
print(f"新数组前{length2}个元素: ", end="")
print_array(nums2, length2)
print()
```

```
测试用例 3 - 边界情况: 空数组
nums3 = []
val3 = 0
print("测试用例 3 (空数组) :")
print("原始数组: []")
print(f"要移除的值: {val3}")
length3 = solution.remove_element(nums3, val3)
print(f"新长度: {length3}")
print(f"新数组前{length3}个元素: []")
print()
```

```
测试用例 4 - 边界情况: 所有元素都等于 val
nums4 = [5, 5, 5, 5]
val4 = 5
print("测试用例 4 (全等于 val 的数组) :")
print("原始数组: ", end="")
```

```
print_full_array(nums4)
print(f"要移除的值: {val4}")
length4 = solution.remove_element(nums4, val4)
print(f"新长度: {length4}")
print(f"新数组前{length4}个元素: []")
print()
```

```
测试用例 5 - 边界情况: 没有元素等于 val
nums5 = [1, 2, 3, 4, 5]
val5 = 6
print("测试用例 5 (无等于 val 的元素) :")
print("原始数组: ", end="")
print_full_array(nums5)
print(f"要移除的值: {val5}")
length5 = solution.remove_element(nums5, val5)
print(f"新长度: {length5}")
print(f"新数组前{length5}个元素: ", end="")
print_array(nums5, length5)
print()
```

"""

## 性能测试

"""

```
def performance_test():
 solution = Solution()

 # 创建一个大数组进行性能测试
 size = 1000000
 # 填充数组, 其中约 20% 的元素等于 val
 val = 5
 large_array = [val if i % 10 == 0 else i % 10 for i in range(size)]

 # 测试解法一的性能
 array1 = large_array.copy()
 start_time = time.time()
 result1 = solution.remove_element(array1, val)
 end_time = time.time()
 print(f"解法一耗时: {(end_time - start_time) * 1000:.2f}ms, 结果长度: {result1}")

 # 测试解法二的性能
 array2 = large_array.copy()
 start_time = time.time()
 result2 = solution.remove_element_optimized(array2, val)
```

```

end_time = time.time()
print(f"解法二耗时: {(end_time - start_time) * 1000:.2f}ms, 结果长度: {result2}")

测试解法三的性能
array3 = large_array.copy()
start_time = time.time()
result3 = solution.remove_element_concise(array3, val)
end_time = time.time()
print(f"解法三耗时: {(end_time - start_time) * 1000:.2f}ms, 结果长度: {result3}")

验证所有解法结果一致
print(f"所有解法结果一致: {result1 == result2 and result2 == result3}")

"""

主函数
"""

def main():
 print("== 测试用例 ==")
 test()

 print("== 性能测试 ==")
 performance_test()

if __name__ == "__main__":
 main()

```

---

文件: Code24\_SortColors.cpp

---

```

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

/***
 * LeetCode 75. 颜色分类 (Sort Colors)
 *
 * 题目描述:
 * 给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。
 * 我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

```

```
* 必须在不使用库内置的 sort 函数的情况下解决这个问题。
*
* 示例 1:
* 输入: nums = [2, 0, 2, 1, 1, 0]
* 输出: [0, 0, 1, 1, 2, 2]
*
* 示例 2:
* 输入: nums = [2, 0, 1]
* 输出: [0, 1, 2]
*
* 提示:
* n == nums.length
* 1 <= n <= 300
* nums[i] 为 0、1 或 2
*
* 题目链接: https://leetcode.com/problems/sort-colors/
*
* 解题思路:
* 这道题是一个经典的「荷兰国旗问题」，可以使用双指针或者三指针技术来解决。
*
* 方法一（三指针法）:
* 1. 使用三个指针：left（指向 0 的右边界）、mid（当前处理的元素）、right（指向 2 的左边界）
* 2. 初始化 left=0, mid=0, right=nums.length-1
* 3. 当 mid<=right 时，根据 nums[mid] 的值进行不同的处理：
* - 如果 nums[mid]==0，交换 nums[left] 和 nums[mid]，然后 left++, mid++
* - 如果 nums[mid]==1, mid++
* - 如果 nums[mid]==2，交换 nums[mid] 和 nums[right]，然后 right--（注意此时 mid 不增加，因为交换后的元素还未处理）
*
* 方法二（两次遍历）:
* 1. 第一次遍历统计 0、1、2 的个数
* 2. 第二次遍历根据统计结果填充数组
*
* 最优解是三指针法，时间复杂度 O(n)，空间复杂度 O(1)，且只需要一次遍历。
*/

```

```
class Solution {
public:
 /**
 * 解法一：三指针法（最优解）
 *
 * @param nums 输入数组，只包含 0、1、2 三种元素
 */
}
```

```

void sortColors(std::vector<int>& nums) {
 int left = 0; // 0 的右边界
 int mid = 0; // 当前处理的元素
 int right = nums.size() - 1; // 2 的左边界

 while (mid <= right) {
 switch (nums[mid]) {
 case 0:
 // 当前元素是 0，放到 left 指针位置
 std::swap(nums[left], nums[mid]);
 left++;
 mid++;
 break;
 case 1:
 // 当前元素是 1，已经在正确位置
 mid++;
 break;
 case 2:
 // 当前元素是 2，放到 right 指针位置
 std::swap(nums[mid], nums[right]);
 right--;
 // 注意：此时 mid 不增加，因为交换后的元素还未处理
 break;
 default:
 // 非法输入检查
 throw std::invalid_argument("输入数组包含非法元素，只能包含 0、1、2");
 }
 }
}

/***
 * 解法二：两次遍历（计数排序思想）
 *
 * @param nums 输入数组，只包含 0、1、2 三种元素
 */
void sortColorsTwoPass(std::vector<int>& nums) {
 int count0 = 0, count1 = 0, count2 = 0;

 // 第一次遍历：统计 0、1、2 的个数
 for (int num : nums) {
 switch (num) {
 case 0:
 count0++;

```

```

 break;
 case 1:
 count1++;
 break;
 case 2:
 count2++;
 break;
 default:
 throw std::invalid_argument("输入数组包含非法元素，只能包含 0、1、2");
 }
}

// 第二次遍历：根据统计结果填充数组
int index = 0;
while (count0 > 0) {
 nums[index++] = 0;
 count0--;
}
while (count1 > 0) {
 nums[index++] = 1;
 count1--;
}
while (count2 > 0) {
 nums[index++] = 2;
 count2--;
}
}

/***
 * 解法三：双指针优化版
 *
 * @param nums 输入数组，只包含 0、1、2 三种元素
 */
void sortColorsTwoPointers(std::vector<int>& nums) {
 // 先将所有的 0 移到数组前面
 int p0 = 0;
 for (int i = 0; i < nums.size(); i++) {
 if (nums[i] == 0) {
 std::swap(nums[i], nums[p0]);
 p0++;
 }
 }
}

```

```
// 再将所有的 1 移到 0 之后
int p1 = p0;
for (int i = p0; i < nums.size(); i++) {
 if (nums[i] == 1) {
 std::swap(nums[i], nums[p1]);
 p1++;
 }
}
}

/***
 * 打印数组
 */
void printArray(const std::vector<int>& nums) {
 std::cout << "[";
 for (size_t i = 0; i < nums.size(); i++) {
 std::cout << nums[i];
 if (i < nums.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 验证排序结果是否正确
 */
bool isSorted(const std::vector<int>& nums) {
 for (size_t i = 1; i < nums.size(); i++) {
 if (nums[i] < nums[i - 1]) {
 return false;
 }
 }
 return true;
}

/***
 * 测试函数
 */
void test() {
 Solution solution;
```

```
// 测试用例 1
std::vector<int> nums1 = {2, 0, 2, 1, 1, 0};
std::cout << "测试用例 1:" << std::endl;
std::cout << "排序前: ";
printArray(nums1);
solution.sortColors(nums1);
std::cout << "排序后: ";
printArray(nums1);
std::cout << std::endl;

// 测试用例 2
std::vector<int> nums2 = {2, 0, 1};
std::cout << "测试用例 2:" << std::endl;
std::cout << "排序前: ";
printArray(nums2);
solution.sortColors(nums2);
std::cout << "排序后: ";
printArray(nums2);
std::cout << std::endl;

// 测试用例 3 - 边界情况: 只有一个元素
std::vector<int> nums3 = {0};
std::cout << "测试用例 3 (单元素数组) :" << std::endl;
std::cout << "排序前: ";
printArray(nums3);
solution.sortColors(nums3);
std::cout << "排序后: ";
printArray(nums3);
std::cout << std::endl;

// 测试用例 4 - 边界情况: 已经排序的数组
std::vector<int> nums4 = {0, 0, 1, 1, 2, 2};
std::cout << "测试用例 4 (已排序数组) :" << std::endl;
std::cout << "排序前: ";
printArray(nums4);
solution.sortColors(nums4);
std::cout << "排序后: ";
printArray(nums4);
std::cout << std::endl;

// 测试用例 5 - 边界情况: 逆序排列的数组
std::vector<int> nums5 = {2, 2, 1, 1, 0, 0};
std::cout << "测试用例 5 (逆序数组) :" << std::endl;
```

```

std::cout << "排序前: ";
printArray(nums5);
solution.sortColors(nums5);
std::cout << "排序后: ";
printArray(nums5);
std::cout << std::endl;

// 测试用例 6 - 边界情况: 所有元素都相同
std::vector<int> nums6 = {1, 1, 1, 1};
std::cout << "测试用例 6 (全相同元素) :" << std::endl;
std::cout << "排序前: ";
printArray(nums6);
solution.sortColors(nums6);
std::cout << "排序后: ";
printArray(nums6);
std::cout << std::endl;
}

/***
 * 性能测试
 */
void performanceTest() {
 Solution solution;

 // 创建一个大数组进行性能测试
 int size = 1000000;
 std::vector<int> largeArray;
 largeArray.reserve(size);

 // 随机填充 0、1、2
 for (int i = 0; i < size; i++) {
 largeArray.push_back(i % 3);
 }

 // 测试解法一的性能
 std::vector<int> array1(largeArray.begin(), largeArray.end());
 auto startTime = std::chrono::high_resolution_clock::now();
 solution.sortColors(array1);
 auto endTime = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
 std::cout << "解法一 (三指针) 耗时: " << duration.count() << "ms" << std::endl;

 // 测试解法二的性能
}

```

```

std::vector<int> array2(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
solution.sortColorsTwoPass(array2);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法二（两次遍历）耗时：" << duration.count() << "ms" << std::endl;

// 测试解法三的性能
std::vector<int> array3(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
solution.sortColorsTwoPointers(array3);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法三（双指针优化）耗时：" << duration.count() << "ms" << std::endl;

// 验证所有解法结果一致且已排序
bool resultsConsistent = true;
for (int i = 0; i < size; i++) {
 if (array1[i] != array2[i] || array1[i] != array3[i]) {
 resultsConsistent = false;
 break;
 }
}
std::cout << "所有解法结果一致：" << (resultsConsistent ? "是" : "否") << std::endl;
std::cout << "排序正确：" << (isSorted(array1) ? "是" : "否") << std::endl;
}

int main() {
 std::cout << "==== 测试用例 ===" << std::endl;
 test();

 std::cout << "==== 性能测试 ===" << std::endl;
 performanceTest();

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

文件: Code24\_SortColors.java

```
=====
package class050;
```

```
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

/**
 * LeetCode 75. 颜色分类 (Sort Colors)
 *
 * 题目描述:
 * 给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。
 * 我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。
 * 必须在不使用库内置的 sort 函数的情况下解决这个问题。
 *
 * 示例 1:
 * 输入: nums = [2, 0, 2, 1, 1, 0]
 * 输出: [0, 0, 1, 1, 2, 2]
 *
 * 示例 2:
 * 输入: nums = [2, 0, 1]
 * 输出: [0, 1, 2]
 *
 * 提示:
 * n == nums.length
 * 1 <= n <= 300
 * nums[i] 为 0、1 或 2
 *
 * 题目链接: https://leetcode.com/problems/sort-colors/
 *
 * 解题思路:
 * 这道题是一个经典的「荷兰国旗问题」，可以使用双指针或者三指针技术来解决。
 *
 * 方法一（三指针法）:
 * 1. 使用三个指针: left (指向 0 的右边界)、mid (当前处理的元素)、right (指向 2 的左边界)
 * 2. 初始化 left=0, mid=0, right=nums.length-1
 * 3. 当 mid<=right 时，根据 nums[mid] 的值进行不同的处理:
 * - 如果 nums[mid]==0，交换 nums[left] 和 nums[mid]，然后 left++, mid++
 * - 如果 nums[mid]==1, mid++
 * - 如果 nums[mid]==2, 交换 nums[mid] 和 nums[right]，然后 right-- (注意此时 mid 不增加，因为交换后的元素还未处理)
 *
 * 方法二（两次遍历）:
 * 1. 第一次遍历统计 0、1、2 的个数
 * 2. 第二次遍历根据统计结果填充数组
 *
```

\* 最优解是三指针法，时间复杂度 O(n)，空间复杂度 O(1)，且只需要一次遍历。

\*/

```
public class Code24_SortColors {

 /**
 * 解法一：三指针法（最优解）
 *
 * @param nums 输入数组，只包含 0、1、2 三种元素
 * @throws IllegalArgumentException 如果输入数组为 null
 */
 public static void sortColors(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 int left = 0; // 0 的右边界
 int mid = 0; // 当前处理的元素
 int right = nums.length - 1; // 2 的左边界

 while (mid <= right) {
 switch (nums[mid]) {
 case 0:
 // 当前元素是 0，放到 left 指针位置
 swap(nums, left, mid);
 left++;
 mid++;
 break;
 case 1:
 // 当前元素是 1，已经在正确位置
 mid++;
 break;
 case 2:
 // 当前元素是 2，放到 right 指针位置
 swap(nums, mid, right);
 right--;
 // 注意：此时 mid 不增加，因为交换后的元素还未处理
 break;
 default:
 // 非法输入检查
 throw new IllegalArgumentException("输入数组包含非法元素，只能包含 0、1、2");
 }
 }
 }
}
```

```
 }
}

/***
 * 解法二：两次遍历（计数排序思想）
 *
 * @param nums 输入数组，只包含 0、1、2 三种元素
 */
public static void sortColorsTwoPass(int[] nums) {
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 int count0 = 0, count1 = 0, count2 = 0;

 // 第一次遍历：统计 0、1、2 的个数
 for (int num : nums) {
 switch (num) {
 case 0:
 count0++;
 break;
 case 1:
 count1++;
 break;
 case 2:
 count2++;
 break;
 default:
 throw new IllegalArgumentException("输入数组包含非法元素，只能包含 0、1、2");
 }
 }

 // 第二次遍历：根据统计结果填充数组
 int index = 0;
 while (count0 > 0) {
 nums[index++] = 0;
 count0--;
 }
 while (count1 > 0) {
 nums[index++] = 1;
 count1--;
 }
 while (count2 > 0) {
```

```
 nums[index++] = 2;
 count2--;
 }
}

/***
 * 解法三：双指针优化版
 *
 * @param nums 输入数组，只包含 0、1、2 三种元素
 */
public static void sortColorsTwoPointers(int[] nums) {
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 // 先将所有的 0 移到数组前面
 int p0 = 0;
 for (int i = 0; i < nums.length; i++) {
 if (nums[i] == 0) {
 swap(nums, i, p0);
 p0++;
 }
 }

 // 再将所有的 1 移到 0 之后
 int p1 = p0;
 for (int i = p0; i < nums.length; i++) {
 if (nums[i] == 1) {
 swap(nums, i, p1);
 p1++;
 }
 }

}

/***
 * 交换数组中的两个元素
 */
private static void swap(int[] nums, int i, int j) {
 int temp = nums[i];
 nums[i] = nums[j];
 nums[j] = temp;
}
```

```
/**
 * 打印数组
 */
public static void printArray(int[] nums) {
 System.out.println(Arrays.toString(nums));
}

/**
 * 测试函数
 */
public static void test() {
 // 测试用例 1
 int[] nums1 = {2, 0, 2, 1, 1, 0};
 System.out.println("测试用例 1:");
 System.out.print("排序前: ");
 printArray(nums1);
 sortColors(nums1);
 System.out.print("排序后: ");
 printArray(nums1);
 System.out.println();

 // 测试用例 2
 int[] nums2 = {2, 0, 1};
 System.out.println("测试用例 2:");
 System.out.print("排序前: ");
 printArray(nums2);
 sortColors(nums2);
 System.out.print("排序后: ");
 printArray(nums2);
 System.out.println();

 // 测试用例 3 - 边界情况: 只有一个元素
 int[] nums3 = {0};
 System.out.println("测试用例 3 (单元素数组):");
 System.out.print("排序前: ");
 printArray(nums3);
 sortColors(nums3);
 System.out.print("排序后: ");
 printArray(nums3);
 System.out.println();

 // 测试用例 4 - 边界情况: 已经排序的数组
 int[] nums4 = {0, 0, 1, 1, 2, 2};
```

```
System.out.println("测试用例 4 (已排序数组) :");
System.out.print("排序前: ");
printArray(nums4);
sortColors(nums4);
System.out.print("排序后: ");
printArray(nums4);
System.out.println();

// 测试用例 5 - 边界情况: 逆序排列的数组
int[] nums5 = {2, 2, 1, 1, 0, 0};
System.out.println("测试用例 5 (逆序数组) :");
System.out.print("排序前: ");
printArray(nums5);
sortColors(nums5);
System.out.print("排序后: ");
printArray(nums5);
System.out.println();

// 测试用例 6 - 边界情况: 所有元素都相同
int[] nums6 = {1, 1, 1, 1};
System.out.println("测试用例 6 (全相同元素) :");
System.out.print("排序前: ");
printArray(nums6);
sortColors(nums6);
System.out.print("排序后: ");
printArray(nums6);
System.out.println();
}

/**
 * 性能测试
 */
public static void performanceTest() {
 // 创建一个大数组进行性能测试
 int size = 1000000;
 int[] largeArray = new int[size];

 // 随机填充 0、1、2
 for (int i = 0; i < size; i++) {
 largeArray[i] = i % 3;
 }

 // 测试解法一的性能
}
```

```
int[] array1 = Arrays.copyOf(largeArray, size);
long startTime = System.nanoTime();
sortColors(array1);
long endTime = System.nanoTime();
long duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法一（三指针）耗时: " + duration + "ms");

// 测试解法二的性能
int[] array2 = Arrays.copyOf(largeArray, size);
startTime = System.nanoTime();
sortColorsTwoPass(array2);
endTime = System.nanoTime();
duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法二（两次遍历）耗时: " + duration + "ms");

// 测试解法三的性能
int[] array3 = Arrays.copyOf(largeArray, size);
startTime = System.nanoTime();
sortColorsTwoPointers(array3);
endTime = System.nanoTime();
duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法三（双指针优化）耗时: " + duration + "ms");

// 验证所有解法结果一致
boolean resultsConsistent = true;
for (int i = 0; i < size; i++) {
 if (array1[i] != array2[i] || array1[i] != array3[i]) {
 resultsConsistent = false;
 break;
 }
}
System.out.println("所有解法结果一致: " + resultsConsistent);
}

/**
 * 验证排序结果是否正确
 */
public static boolean isSorted(int[] nums) {
 for (int i = 1; i < nums.length; i++) {
 if (nums[i] < nums[i - 1]) {
 return false;
 }
 }
}
```

```
 return true;
}

public static void main(String[] args) {
 System.out.println("== 测试用例 ==");
 test();

 System.out.println("== 性能测试 ==");
 performanceTest();
}
}
```

=====

文件: Code24\_SortColors.py

=====

```
from typing import List
import time

class Solution:
 """
 LeetCode 75. 颜色分类 (Sort Colors)
```

题目描述:

给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

我们使用整数 0、1 和 2 分别表示红色、白色和蓝色。

必须在不使用库内置的 sort 函数的情况下解决这个问题。

示例 1:

输入: nums = [2, 0, 2, 1, 1, 0]

输出: [0, 0, 1, 1, 2, 2]

示例 2:

输入: nums = [2, 0, 1]

输出: [0, 1, 2]

提示:

n == nums.length

1 <= n <= 300

nums[i] 为 0、1 或 2

题目链接: <https://leetcode.com/problems/sort-colors/>

解题思路：

这道题是一个经典的「荷兰国旗问题」，可以使用双指针或者三指针技术来解决。

方法一（三指针法）：

1. 使用三个指针：left（指向 0 的右边界）、mid（当前处理的元素）、right（指向 2 的左边界）
2. 初始化 left=0, mid=0, right=nums.length-1
3. 当 mid<=right 时，根据 nums[mid] 的值进行不同的处理：
  - 如果 nums[mid]==0，交换 nums[left] 和 nums[mid]，然后 left++, mid++
  - 如果 nums[mid]==1, mid++
  - 如果 nums[mid]==2，交换 nums[mid] 和 nums[right]，然后 right--（注意此时 mid 不增加，因为交换后的元素还未处理）

方法二（两次遍历）：

1. 第一次遍历统计 0、1、2 的个数
2. 第二次遍历根据统计结果填充数组

最优解是三指针法，时间复杂度  $O(n)$ ，空间复杂度  $O(1)$ ，且只需要一次遍历。

"""

```
def sort_colors(self, nums: List[int]) -> None:
```

"""

解法一：三指针法（最优解）

Args:

    nums: 输入数组，只包含 0、1、2 三种元素

Returns:

    None: 原地修改数组

时间复杂度:  $O(n)$  - 只需要一次遍历

空间复杂度:  $O(1)$  - 只使用常量额外空间

"""

```
if nums is None:
```

```
 raise ValueError("输入数组不能为 None")
```

```
left = 0 # 0 的右边界
```

```
mid = 0 # 当前处理的元素
```

```
right = len(nums) - 1 # 2 的左边界
```

```
while mid <= right:
```

```
 if nums[mid] == 0:
```

```
 # 当前元素是 0，放到 left 指针位置
```

```

 nums[left], nums[mid] = nums[mid], nums[left]
 left += 1
 mid += 1
 elif nums[mid] == 1:
 # 当前元素是 1，已经在正确位置
 mid += 1
 elif nums[mid] == 2:
 # 当前元素是 2，放到 right 指针位置
 nums[mid], nums[right] = nums[right], nums[mid]
 right -= 1
 # 注意：此时 mid 不增加，因为交换后的元素还未处理
 else:
 # 非法输入检查
 raise ValueError("输入数组包含非法元素，只能包含 0、1、2")

```

def sort\_colors\_two\_pass(self, nums: List[int]) -> None:

"""

解法二：两次遍历（计数排序思想）

Args:

nums: 输入数组，只包含 0、1、2 三种元素

Returns:

None: 原地修改数组

时间复杂度:  $O(n)$  - 需要两次遍历

空间复杂度:  $O(1)$  - 只使用常量额外空间

"""

if nums is None:

raise ValueError("输入数组不能为 None")

count0, count1, count2 = 0, 0, 0

# 第一次遍历：统计 0、1、2 的个数

for num in nums:

if num == 0:

count0 += 1

elif num == 1:

count1 += 1

elif num == 2:

count2 += 1

else:

raise ValueError("输入数组包含非法元素，只能包含 0、1、2")

```

第二次遍历：根据统计结果填充数组
index = 0
while count0 > 0:
 nums[index] = 0
 index += 1
 count0 -= 1
while count1 > 0:
 nums[index] = 1
 index += 1
 count1 -= 1
while count2 > 0:
 nums[index] = 2
 index += 1
 count2 -= 1

def sort_colors_two_pointers(self, nums: List[int]) -> None:
"""

```

解法三：双指针优化版

Args:

nums: 输入数组，只包含 0、1、2 三种元素

Returns:

None: 原地修改数组

时间复杂度:  $O(n)$  - 需要两次遍历

空间复杂度:  $O(1)$  - 只使用常量额外空间

"""

```

if nums is None:
 raise ValueError("输入数组不能为 None")

```

# 先将所有的 0 移到数组前面

```

p0 = 0
for i in range(len(nums)):
 if nums[i] == 0:
 nums[i], nums[p0] = nums[p0], nums[i]
 p0 += 1

```

# 再将所有的 1 移到 0 之后

```

p1 = p0
for i in range(p0, len(nums)):
 if nums[i] == 1:

```

```
 nums[i], nums[p1] = nums[p1], nums[i]
 p1 += 1

def is_sorted(self, nums: List[int]) -> bool:
 """
 验证排序结果是否正确

 Args:
 nums: 要验证的数组

 Returns:
 bool: 如果数组已排序返回 True, 否则返回 False
 """
 for i in range(1, len(nums)):
 if nums[i] < nums[i-1]:
 return False
 return True

def test(self):
 """
 测试函数
 """

 # 测试用例 1
 nums1 = [2, 0, 2, 1, 1, 0]
 print("测试用例 1:")
 print(f"排序前: {nums1}")
 self.sort_colors(nums1)
 print(f"排序后: {nums1}")
 print()

 # 测试用例 2
 nums2 = [2, 0, 1]
 print("测试用例 2:")
 print(f"排序前: {nums2}")
 self.sort_colors(nums2)
 print(f"排序后: {nums2}")
 print()

 # 测试用例 3 - 边界情况: 只有一个元素
 nums3 = [0]
 print("测试用例 3 (单元素数组):")
 print(f"排序前: {nums3}")
 self.sort_colors(nums3)
```

```
print(f"排序后: {nums3}")
print()

测试用例 4 - 边界情况: 已经排序的数组
nums4 = [0, 0, 1, 1, 2, 2]
print("测试用例 4 (已排序数组) :")
print(f"排序前: {nums4}")
self.sort_colors(nums4)
print(f"排序后: {nums4}")
print()

测试用例 5 - 边界情况: 逆序排列的数组
nums5 = [2, 2, 1, 1, 0, 0]
print("测试用例 5 (逆序数组) :")
print(f"排序前: {nums5}")
self.sort_colors(nums5)
print(f"排序后: {nums5}")
print()

测试用例 6 - 边界情况: 所有元素都相同
nums6 = [1, 1, 1, 1]
print("测试用例 6 (全相同元素) :")
print(f"排序前: {nums6}")
self.sort_colors(nums6)
print(f"排序后: {nums6}")
print()

def performance_test(self):
 """
 性能测试
 """
 # 创建一个大数组进行性能测试
 size = 1000000
 large_array = [i % 3 for i in range(size)]

 # 测试解法一的性能
 array1 = large_array.copy()
 start_time = time.time()
 self.sort_colors(array1)
 end_time = time.time()
 duration = (end_time - start_time) * 1000 # 转换为毫秒
 print(f"解法一 (三指针) 耗时: {duration:.2f}ms")
```

```

测试解法二的性能
array2 = large_array.copy()
start_time = time.time()
self.sort_colors_two_pass(array2)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法二（两次遍历）耗时: {duration:.2f}ms")

测试解法三的性能
array3 = large_array.copy()
start_time = time.time()
self.sort_colors_two_pointers(array3)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法三（双指针优化）耗时: {duration:.2f}ms")

验证所有解法结果一致且已排序
results_consistent = True
for i in range(size):
 if array1[i] != array2[i] or array1[i] != array3[i]:
 results_consistent = False
 break
print(f"所有解法结果一致: {results_consistent}")
print(f"排序正确: {self.is_sorted(array1)})"

主函数
if __name__ == "__main__":
 solution = Solution()

 print("== 测试用例 ==")
 solution.test()

 print("== 性能测试 ==")
 solution.performance_test()

```

=====

文件: Code25\_RemoveDuplicatesFromSortedArrayII.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
```

```
#include <stdexcept>

/**
 * LeetCode 80. 删除有序数组中的重复项 II (Remove Duplicates from Sorted Array II)
 *
 * 题目描述:
 * 给你一个有序数组 nums，请你原地删除重复出现的元素，使得出现次数超过两次的元素只出现两次，返回删除后数组的新长度。
 * 不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。
 *
 * 示例 1:
 * 输入: nums = [1,1,1,2,2,3]
 * 输出: 5, nums = [1,1,2,2,3]
 * 解释: 函数应返回新长度 length = 5，并且原数组的前五个元素被修改为 [1,1,2,2,3]。不需要考虑数组中超出新长度后面的元素。
 *
 * 示例 2:
 * 输入: nums = [0,0,1,1,1,2,3,3]
 * 输出: 7, nums = [0,0,1,1,2,3,3]
 * 解释: 函数应返回新长度 length = 7，并且原数组的前七个元素被修改为 [0,0,1,1,2,3,3]。不需要考虑数组中超出新长度后面的元素。
 *
 * 提示:
 * 1 <= nums.length <= 3 * 10^4
 * -10^4 <= nums[i] <= 10^4
 * nums 已按升序排列
 *
 * 题目链接: https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/
 *
 * 解题思路:
 * 这道题是 LeetCode 26 的升级版，允许元素最多出现两次。我们可以使用快慢指针的方法来解决：
 *
 * 方法一（快慢指针）:
 * 1. 使用 slow 指针指向当前已处理好的有效部分的下一个位置
 * 2. 使用 count 变量记录当前元素的重复次数
 * 3. 遍历数组，当发现当前元素与前一个元素相同时，增加计数；否则重置计数为 1
 * 4. 如果当前元素的计数不超过 2，将其移动到 slow 指向的位置，然后 slow++
 *
 * 方法二（优化的快慢指针）:
 * 由于数组是有序的，我们可以简化为直接比较 nums[fast] 和 nums[slow-2] 的值
 * - 如果 nums[fast] != nums[slow-2]，说明当前元素可以保留，放到 slow 位置，然后 slow++
 * - 否则，说明这个元素已经出现了两次以上，跳过
 *
```

\* 最优解是方法二，时间复杂度 O(n)，空间复杂度 O(1)。

\*/

```
class Solution {
public:
 /**
 * 解法一：快慢指针 + 计数法
 *
 * @param nums 输入的有序数组
 * @return 删除重复元素后的新长度
 */
 int removeDuplicates(std::vector<int>& nums) {
 // 边界情况：数组长度小于等于 2，所有元素都可以保留
 int n = nums.size();
 if (n <= 2) {
 return n;
 }

 int slow = 1; // 慢指针，指向当前已处理好的不重复元素的最后一个位置
 int count = 1; // 记录当前元素重复的次数

 // 从第二个元素开始遍历
 for (int fast = 1; fast < n; fast++) {
 // 如果当前元素与前一个元素相同，增加计数
 if (nums[fast] == nums[fast - 1]) {
 count++;
 } else {
 // 否则重置计数
 count = 1;
 }

 // 如果当前元素的计数不超过 2，将其移到慢指针位置
 if (count <= 2) {
 slow++;
 // 只有当 fast 和 slow 不同时才需要复制，避免不必要的操作
 if (fast != slow - 1) {
 nums[slow - 1] = nums[fast];
 }
 }
 }

 return slow;
 }
}
```

```

/**
 * 解法二：优化的快慢指针（最优解）
 *
 * @param nums 输入的有序数组
 * @return 删除重复元素后的新长度
 */
int removeDuplicatesOptimized(std::vector<int>& nums) {
 // 边界情况：数组长度小于等于 2，所有元素都可以保留
 int n = nums.size();
 if (n <= 2) {
 return n;
 }

 // 慢指针初始化为 2，因为前两个元素肯定可以保留
 int slow = 2;

 // 快指针从第三个元素开始遍历
 for (int fast = 2; fast < n; fast++) {
 // 如果当前元素与 slow-2 位置的元素不同，说明这个元素可以保留
 if (nums[fast] != nums[slow - 2]) {
 nums[slow] = nums[fast];
 slow++;
 }
 }

 return slow;
}

/**
 * 解法三：通用解法，支持最多 k 个重复元素
 *
 * @param nums 输入的有序数组
 * @param k 允许的最大重复次数
 * @return 删除重复元素后的新长度
 */
int removeDuplicatesGeneric(std::vector<int>& nums, int k) {
 if (k <= 0) {
 throw std::invalid_argument("k 必须为正整数");
 }

 // 边界情况：数组长度小于等于 k，所有元素都可以保留
 int n = nums.size();

```

```

 if (n <= k) {
 return n;
 }

 // 慢指针初始化为 k, 因为前 k 个元素肯定可以保留
 int slow = k;

 // 快指针从第 k+1 个元素开始遍历
 for (int fast = k; fast < n; fast++) {
 // 如果当前元素与 slow-k 位置的元素不同, 说明这个元素可以保留
 if (nums[fast] != nums[slow - k]) {
 nums[slow] = nums[fast];
 slow++;
 }
 }

 return slow;
}

};

/***
 * 打印数组的前 len 个元素
 */
void printArray(const std::vector<int>& nums, int len) {
 std::cout << "[";
 for (int i = 0; i < len; i++) {
 std::cout << nums[i];
 if (i < len - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]" << std::endl;
}

/***
 * 验证结果是否正确
 */
bool validateResult(const std::vector<int>& original, const std::vector<int>& expected, int
expectedLen) {
 // 检查长度
 if (expectedLen != expected.size()) {
 return false;
 }
}

```

```

// 检查前 expectedLen 个元素
for (int i = 0; i < expectedLen; i++) {
 if (original[i] != expected[i]) {
 return false;
 }
}

// 检查数组是否仍然有序
for (int i = 1; i < expectedLen; i++) {
 if (original[i] < original[i - 1]) {
 return false;
 }
}

return true;
}

/***
 * 测试函数
 */
void test() {
 Solution solution;

 // 测试用例 1
 std::vector<int> nums1 = {1, 1, 1, 2, 2, 3};
 std::vector<int> expected1 = {1, 1, 2, 2, 3};
 std::cout << "测试用例 1:" << std::endl;
 std::cout << "原数组: [";
 for (size_t i = 0; i < nums1.size(); i++) {
 std::cout << nums1[i];
 if (i < nums1.size() - 1) std::cout << ", ";
 }
 std::cout << "]" << std::endl;
 int len1 = solution.removeDuplicatesOptimized(nums1);
 std::cout << "处理后: ";
 printArray(nums1, len1);
 std::cout << "长度: " << len1 << std::endl;
 std::cout << "验证结果: " << (validateResult(nums1, expected1, len1) ? "通过" : "失败") << std::endl;
 std::cout << std::endl;

 // 测试用例 2
}

```

```

std::vector<int> nums2 = {0, 0, 1, 1, 1, 1, 2, 3, 3};
std::vector<int> expected2 = {0, 0, 1, 1, 2, 3, 3};
std::cout << "测试用例 2:" << std::endl;
std::cout << "原数组: [";
for (size_t i = 0; i < nums2.size(); i++) {
 std::cout << nums2[i];
 if (i < nums2.size() - 1) std::cout << ", ";
}
std::cout << "]" << std::endl;
int len2 = solution.removeDuplicatesOptimized(nums2);
std::cout << "处理后: ";
printArray(nums2, len2);
std::cout << "长度: " << len2 << std::endl;
std::cout << "验证结果: " << (validateResult(nums2, expected2, len2) ? "通过" : "失败") <<
std::endl;
std::cout << std::endl;

// 测试用例 3 - 边界情况: 数组长度小于等于 2
std::vector<int> nums3 = {1, 1};
std::cout << "测试用例 3 (数组长度为 2) :" << std::endl;
std::cout << "原数组: [";
for (size_t i = 0; i < nums3.size(); i++) {
 std::cout << nums3[i];
 if (i < nums3.size() - 1) std::cout << ", ";
}
std::cout << "]" << std::endl;
int len3 = solution.removeDuplicatesOptimized(nums3);
std::cout << "处理后: ";
printArray(nums3, len3);
std::cout << "长度: " << len3 << std::endl;
std::cout << std::endl;

// 测试用例 4 - 边界情况: 所有元素都相同
std::vector<int> nums4 = {2, 2, 2, 2, 2};
std::vector<int> expected4 = {2, 2};
std::cout << "测试用例 4 (所有元素相同) :" << std::endl;
std::cout << "原数组: [";
for (size_t i = 0; i < nums4.size(); i++) {
 std::cout << nums4[i];
 if (i < nums4.size() - 1) std::cout << ", ";
}
std::cout << "]" << std::endl;
int len4 = solution.removeDuplicatesOptimized(nums4);

```

```

std::cout << "处理后: ";
printArray(nums4, len4);
std::cout << "长度: " << len4 << std::endl;
std::cout << "验证结果: " << (validateResult(nums4, expected4, len4) ? "通过" : "失败") <<
std::endl;
std::cout << std::endl;

// 测试用例 5 - 边界情况: 没有重复元素
std::vector<int> nums5 = {1, 2, 3, 4, 5};
std::cout << "测试用例 5 (无重复元素) :" << std::endl;
std::cout << "原数组: [";
for (size_t i = 0; i < nums5.size(); i++) {
 std::cout << nums5[i];
 if (i < nums5.size() - 1) std::cout << ", ";
}
std::cout << "]" << std::endl;
int len5 = solution.removeDuplicatesOptimized(nums5);
std::cout << "处理后: ";
printArray(nums5, len5);
std::cout << "长度: " << len5 << std::endl;
std::cout << "验证结果: " << (validateResult(nums5, std::vector<int>(nums5.begin(),
nums5.begin() + len5), len5) ? "通过" : "失败") << std::endl;
std::cout << std::endl;
}

/**
 * 性能测试
 */
void performanceTest() {
 Solution solution;

 // 创建一个大数组进行性能测试
 int size = 1000000;
 std::vector<int> largeArray;
 largeArray.reserve(size);

 // 生成测试数据: 交替重复元素
 for (int i = 0; i < size; i++) {
 largeArray.push_back(i / 10); // 每个数字重复 10 次
 }

 // 测试解法一的性能
 std::vector<int> array1(largeArray.begin(), largeArray.end());
}

```

```

auto startTime = std::chrono::high_resolution_clock::now();
int len1 = solution.removeDuplicates(array1);
auto endTime = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法一（快慢指针+计数）耗时：" << duration.count() << "ms, 处理后长度：" <<
len1 << std::endl;

// 测试解法二的性能
std::vector<int> array2(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
int len2 = solution.removeDuplicatesOptimized(array2);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法二（优化快慢指针）耗时：" << duration.count() << "ms, 处理后长度：" << len2
<< std::endl;

// 测试解法三的性能
std::vector<int> array3(largeArray.begin(), largeArray.end());
startTime = std::chrono::high_resolution_clock::now();
int len3 = solution.removeDuplicatesGeneric(array3, 2);
endTime = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);
std::cout << "解法三（通用解法）耗时：" << duration.count() << "ms, 处理后长度：" << len3 <<
std::endl;

// 验证所有解法结果一致
bool resultsConsistent = (len1 == len2 && len2 == len3);
if (resultsConsistent) {
 for (int i = 0; i < len1; i++) {
 if (array1[i] != array2[i] || array1[i] != array3[i]) {
 resultsConsistent = false;
 break;
 }
 }
}
std::cout << "所有解法结果一致：" << (resultsConsistent ? "是" : "否") << std::endl;
}

int main() {
 std::cout << "==== 测试用例 ===" << std::endl;
 test();
}

std::cout << "==== 性能测试 ===" << std::endl;

```

```
 performanceTest();

 return 0;
}
```

---

文件: Code25\_RemoveDuplicatesFromSortedArrayII.java

---

```
package class050;

import java.util.Arrays;
import java.util.concurrent.TimeUnit;

/**
 * LeetCode 80. 删除有序数组中的重复项 II (Remove Duplicates from Sorted Array II)
 *
 * 题目描述:
 * 给你一个有序数组 nums，请你原地删除重复出现的元素，使得出现次数超过两次的元素只出现两次，返回
删除后数组的新长度。
 * 不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。
 *
 * 示例 1:
 * 输入: nums = [1, 1, 1, 2, 2, 3]
 * 输出: 5, nums = [1, 1, 2, 2, 3]
 * 解释: 函数应返回新长度 length = 5，并且原数组的前五个元素被修改为 [1, 1, 2, 2, 3]。不需要考虑数组
中超出新长度后面的元素。
 *
 * 示例 2:
 * 输入: nums = [0, 0, 1, 1, 1, 1, 2, 3, 3]
 * 输出: 7, nums = [0, 0, 1, 1, 2, 3, 3]
 * 解释: 函数应返回新长度 length = 7，并且原数组的前七个元素被修改为 [0, 0, 1, 1, 2, 3, 3]。不需要考虑
数组中超出新长度后面的元素。
 *
 * 提示:
 * 1 <= nums.length <= 3 * 10^4
 * -10^4 <= nums[i] <= 10^4
 * nums 已按升序排列
 *
 * 题目链接: https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/
 *
 * 解题思路:
 * 这道题是 LeetCode 26 的升级版，允许元素最多出现两次。我们可以使用快慢指针的方法来解决：
```

```

*
* 方法一（快慢指针）：
* 1. 使用 slow 指针指向当前已处理好的有效部分的下一个位置
* 2. 使用 count 变量记录当前元素的重复次数
* 3. 遍历数组，当发现当前元素与前一个元素相同时，增加计数；否则重置计数为 1
* 4. 如果当前元素的计数不超过 2，将其移动到 slow 指向的位置，然后 slow++
*
* 方法二（优化的快慢指针）：
* 由于数组是有序的，我们可以简化为直接比较 nums[fast] 和 nums[slow-2] 的值
* - 如果 nums[fast] != nums[slow-2]，说明当前元素可以保留，放到 slow 位置，然后 slow++
* - 否则，说明这个元素已经出现了两次以上，跳过
*
* 最优解是方法二，时间复杂度 O(n)，空间复杂度 O(1)。
*/

```

```

public class Code25_RemoveDuplicatesFromSortedArrayII {

 /**
 * 解法一：快慢指针 + 计数法
 *
 * @param nums 输入的有序数组
 * @return 删除重复元素后的新长度
 * @throws IllegalArgumentException 如果输入数组为 null
 */
 public static int removeDuplicates(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 // 边界情况：数组长度小于等于 2，所有元素都可以保留
 int n = nums.length;
 if (n <= 2) {
 return n;
 }

 int slow = 1; // 慢指针，指向当前已处理好的不重复元素的最后一个位置
 int count = 1; // 记录当前元素重复的次数
 }
}
```

```

int slow = 1; // 慢指针，指向当前已处理好的不重复元素的最后一个位置
int count = 1; // 记录当前元素重复的次数

```

```

// 从第二个元素开始遍历
for (int fast = 1; fast < n; fast++) {
 // 如果当前元素与前一个元素相同，增加计数
 if (nums[fast] == nums[fast - 1]) {

```

```

 count++;
 } else {
 // 否则重置计数
 count = 1;
 }

 // 如果当前元素的计数不超过 2，将其移到慢指针位置
 if (count <= 2) {
 slow++;
 // 只有当 fast 和 slow 不同时才需要复制，避免不必要的操作
 if (fast != slow - 1) {
 nums[slow - 1] = nums[fast];
 }
 }
}

return slow;
}

/**
 * 解法二：优化的快慢指针（最优解）
 *
 * @param nums 输入的有序数组
 * @return 删除重复元素后的新长度
 */
public static int removeDuplicatesOptimized(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 // 边界情况：数组长度小于等于 2，所有元素都可以保留
 int n = nums.length;
 if (n <= 2) {
 return n;
 }

 // 慢指针初始化为 2，因为前两个元素肯定可以保留
 int slow = 2;

 // 快指针从第三个元素开始遍历
 for (int fast = 2; fast < n; fast++) {
 // 如果当前元素与 slow-2 位置的元素不同，说明这个元素可以保留

```

```

 if (nums[fast] != nums[slow - 2]) {
 nums[slow] = nums[fast];
 slow++;
 }
 }

 return slow;
}

/***
 * 解法三：通用解法，支持最多 k 个重复元素
 *
 * @param nums 输入的有序数组
 * @param k 允许的最大重复次数
 * @return 删除重复元素后的新长度
 */
public static int removeDuplicatesGeneric(int[] nums, int k) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 if (k <= 0) {
 throw new IllegalArgumentException("k 必须为正整数");
 }

 // 边界情况：数组长度小于等于 k，所有元素都可以保留
 int n = nums.length;
 if (n <= k) {
 return n;
 }

 // 慢指针初始化为 k，因为前 k 个元素肯定可以保留
 int slow = k;

 // 快指针从第 k+1 个元素开始遍历
 for (int fast = k; fast < n; fast++) {
 // 如果当前元素与 slow-k 位置的元素不同，说明这个元素可以保留
 if (nums[fast] != nums[slow - k]) {
 nums[slow] = nums[fast];
 slow++;
 }
 }
}

```

```
 return slow;
 }

/**
 * 打印数组的前 len 个元素
 */
public static void printArray(int[] nums, int len) {
 System.out.print("[");
 for (int i = 0; i < len; i++) {
 System.out.print(nums[i]);
 if (i < len - 1) {
 System.out.print(", ");
 }
 }
 System.out.println("]");
}

/**
 * 验证结果是否正确
 */
public static boolean validateResult(int[] original, int[] expected, int expectedLen) {
 // 检查长度
 if (expectedLen != expected.length) {
 return false;
 }

 // 检查前 expectedLen 个元素
 for (int i = 0; i < expectedLen; i++) {
 if (original[i] != expected[i]) {
 return false;
 }
 }

 // 检查数组是否仍然有序
 for (int i = 1; i < expectedLen; i++) {
 if (original[i] < original[i - 1]) {
 return false;
 }
 }

 return true;
}
```

```
/**
 * 测试函数
 */

public static void test() {

 // 测试用例 1
 int[] nums1 = {1, 1, 1, 2, 2, 3};
 int[] expected1 = {1, 1, 2, 2, 3};
 System.out.println("测试用例 1:");
 System.out.print("原数组: ");
 System.out.println(Arrays.toString(nums1));
 int len1 = removeDuplicatesOptimized(nums1);
 System.out.print("处理后: ");
 printArray(nums1, len1);
 System.out.println("长度: " + len1);
 System.out.println("验证结果: " + validateResult(nums1, expected1, len1));
 System.out.println();

 // 测试用例 2
 int[] nums2 = {0, 0, 1, 1, 1, 1, 2, 3, 3};
 int[] expected2 = {0, 0, 1, 1, 2, 3, 3};
 System.out.println("测试用例 2:");
 System.out.print("原数组: ");
 System.out.println(Arrays.toString(nums2));
 int len2 = removeDuplicatesOptimized(nums2);
 System.out.print("处理后: ");
 printArray(nums2, len2);
 System.out.println("长度: " + len2);
 System.out.println("验证结果: " + validateResult(nums2, expected2, len2));
 System.out.println();

 // 测试用例 3 - 边界情况: 数组长度小于等于 2
 int[] nums3 = {1, 1};
 System.out.println("测试用例 3 (数组长度为 2) :");
 System.out.print("原数组: ");
 System.out.println(Arrays.toString(nums3));
 int len3 = removeDuplicatesOptimized(nums3);
 System.out.print("处理后: ");
 printArray(nums3, len3);
 System.out.println("长度: " + len3);
 System.out.println();

 // 测试用例 4 - 边界情况: 所有元素都相同
 int[] nums4 = {2, 2, 2, 2, 2};
```

```

int[] expected4 = {2, 2};
System.out.println("测试用例 4 (所有元素相同) :");
System.out.print("原数组: ");
System.out.println(Arrays.toString(nums4));
int len4 = removeDuplicatesOptimized(nums4);
System.out.print("处理后: ");
printArray(nums4, len4);
System.out.println("长度: " + len4);
System.out.println("验证结果: " + validateResult(nums4, expected4, len4));
System.out.println();

// 测试用例 5 - 边界情况: 没有重复元素
int[] nums5 = {1, 2, 3, 4, 5};
System.out.println("测试用例 5 (无重复元素) :");
System.out.print("原数组: ");
System.out.println(Arrays.toString(nums5));
int len5 = removeDuplicatesOptimized(nums5);
System.out.print("处理后: ");
printArray(nums5, len5);
System.out.println("长度: " + len5);
System.out.println("验证结果: " + validateResult(nums5, Arrays.copyOf(nums5, len5),
len5));
System.out.println();
}

/***
 * 性能测试
 */
public static void performanceTest() {
 // 创建一个大数组进行性能测试
 int size = 1000000;
 int[] largeArray = new int[size];

 // 生成测试数据: 交替重复元素
 for (int i = 0; i < size; i++) {
 largeArray[i] = i / 10; // 每个数字重复 10 次
 }

 // 测试解法一的性能
 int[] array1 = Arrays.copyOf(largeArray, size);
 long startTime = System.nanoTime();
 int len1 = removeDuplicates(array1);
 long endTime = System.nanoTime();
}

```

```

long duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法一（快慢指针+计数）耗时：" + duration + "ms, 处理后长度：" +
len1);

// 测试解法二的性能
int[] array2 = Arrays.copyOf(largeArray, size);
startTime = System.nanoTime();
int len2 = removeDuplicatesOptimized(array2);
endTime = System.nanoTime();
duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法二（优化快慢指针）耗时：" + duration + "ms, 处理后长度：" +
len2);

// 测试解法三的性能
int[] array3 = Arrays.copyOf(largeArray, size);
startTime = System.nanoTime();
int len3 = removeDuplicatesGeneric(array3, 2);
endTime = System.nanoTime();
duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法三（通用解法）耗时：" + duration + "ms, 处理后长度：" + len3);

// 验证所有解法结果一致
boolean resultsConsistent = (len1 == len2 && len2 == len3);
if (resultsConsistent) {
 for (int i = 0; i < len1; i++) {
 if (array1[i] != array2[i] || array1[i] != array3[i]) {
 resultsConsistent = false;
 break;
 }
 }
}
System.out.println("所有解法结果一致：" + resultsConsistent);
}

public static void main(String[] args) {
 System.out.println("== 测试用例 ==");
 test();

 System.out.println("== 性能测试 ==");
 performanceTest();
}
}

```

文件: Code25\_RemoveDuplicatesFromSortedArrayII.py

```
=====
from typing import List
import time

class Solution:
 """
 LeetCode 80. 删除有序数组中的重复项 II (Remove Duplicates from Sorted Array II)
```

### 题目描述:

给你一个有序数组 `nums`, 请你原地删除重复出现的元素, 使得出现次数超过两次的元素只出现两次, 返回删除后数组的新长度。

不要使用额外的数组空间, 你必须在原地修改输入数组并在使用  $O(1)$  额外空间的条件下完成。

### 示例 1:

输入: `nums = [1, 1, 1, 2, 2, 3]`

输出: 5, `nums = [1, 1, 2, 2, 3]`

解释: 函数应返回新长度 `length = 5`, 并且原数组的前五个元素被修改为 `[1, 1, 2, 2, 3]`。 不需要考虑数组中超出新长度后面的元素。

### 示例 2:

输入: `nums = [0, 0, 1, 1, 1, 1, 2, 3, 3]`

输出: 7, `nums = [0, 0, 1, 1, 2, 3, 3]`

解释: 函数应返回新长度 `length = 7`, 并且原数组的前七个元素被修改为 `[0, 0, 1, 1, 2, 3, 3]`。 不需要考虑数组中超出新长度后面的元素。

### 提示:

$1 \leq \text{nums.length} \leq 3 * 10^4$

$-10^4 \leq \text{nums}[i] \leq 10^4$

`nums` 已按升序排列

题目链接: <https://leetcode.com/problems/remove-duplicates-from-sorted-array-ii/>

### 解题思路:

这道题是 LeetCode 26 的升级版, 允许元素最多出现两次。我们可以使用快慢指针的方法来解决:

### 方法一 (快慢指针):

1. 使用 `slow` 指针指向当前已处理好的有效部分的下一个位置
2. 使用 `count` 变量记录当前元素的重复次数
3. 遍历数组, 当发现当前元素与前一个元素相同时, 增加计数; 否则重置计数为 1
4. 如果当前元素的计数不超过 2, 将其移动到 `slow` 指向的位置, 然后 `slow++`

## 方法二（优化的快慢指针）：

由于数组是有序的，我们可以简化为直接比较 `nums[fast]` 和 `nums[slow-2]` 的值

- 如果 `nums[fast] != nums[slow-2]`，说明当前元素可以保留，放到 `slow` 位置，然后 `slow++`
- 否则，说明这个元素已经出现了两次以上，跳过

最优解是方法二，时间复杂度  $O(n)$ ，空间复杂度  $O(1)$ 。

"""

```
def remove_duplicates(self, nums: List[int]) -> int:
```

"""

解法一：快慢指针 + 计数法

Args:

nums: 输入的有序数组

Returns:

int: 删除重复元素后的新长度

时间复杂度:  $O(n)$  - 只需要一次遍历

空间复杂度:  $O(1)$  - 只使用常量额外空间

"""

# 边界情况：数组长度小于等于 2，所有元素都可以保留

n = len(nums)

if n <= 2:

return n

slow = 1 # 慢指针，指向当前已处理好的不重复元素的最后一个位置

count = 1 # 记录当前元素重复的次数

# 从第二个元素开始遍历

for fast in range(1, n):

# 如果当前元素与前一个元素相同，增加计数

if nums[fast] == nums[fast - 1]:

count += 1

else:

# 否则重置计数

count = 1

# 如果当前元素的计数不超过 2，将其移到慢指针位置

if count <= 2:

slow += 1

# 只有当 fast 和 slow 不同时才需要复制，避免不必要的操作

```
 if fast != slow - 1:
 nums[slow - 1] = nums[fast]

 return slow
```

```
def remove_duplicates_optimized(self, nums: List[int]) -> int:
 """
```

解法二：优化的快慢指针（最优解）

Args:

    nums: 输入的有序数组

Returns:

    int: 删除重复元素后的新长度

时间复杂度:  $O(n)$  - 只需要一次遍历

空间复杂度:  $O(1)$  - 只使用常量额外空间

"""

# 边界情况: 数组长度小于等于 2, 所有元素都可以保留

n = len(nums)

if n <= 2:

return n

# 慢指针初始化为 2, 因为前两个元素肯定可以保留

slow = 2

# 快指针从第三个元素开始遍历

for fast in range(2, n):

# 如果当前元素与 slow-2 位置的元素不同, 说明这个元素可以保留

if nums[fast] != nums[slow - 2]:

nums[slow] = nums[fast]

slow += 1

return slow

```
def remove_duplicates_generic(self, nums: List[int], k: int) -> int:
 """
```

解法三：通用解法，支持最多  $k$  个重复元素

Args:

    nums: 输入的有序数组

    k: 允许的最大重复次数

Returns:

int: 删除重复元素后的新长度

时间复杂度:  $O(n)$  - 只需要一次遍历

空间复杂度:  $O(1)$  - 只使用常量额外空间

"""

if k <= 0:

    raise ValueError("k 必须为正整数")

# 边界情况: 数组长度小于等于 k, 所有元素都可以保留

n = len(nums)

if n <= k:

    return n

# 慢指针初始化为 k, 因为前 k 个元素肯定可以保留

slow = k

# 快指针从第 k+1 个元素开始遍历

for fast in range(k, n):

    # 如果当前元素与 slow-k 位置的元素不同, 说明这个元素可以保留

    if nums[fast] != nums[slow - k]:

        nums[slow] = nums[fast]

        slow += 1

return slow

```
def validate_result(self, original: List[int], expected: List[int], expected_len: int) ->
bool:
 """
```

验证结果是否正确

Args:

original: 处理后的数组

expected: 期望的结果数组

expected\_len: 期望的数组长度

Returns:

bool: 如果结果正确返回 True, 否则返回 False

"""

# 检查长度

if expected\_len != len(expected):

    return False

```
检查前 expected_len 个元素
for i in range(expected_len):
 if original[i] != expected[i]:
 return False

检查数组是否仍然有序
for i in range(1, expected_len):
 if original[i] < original[i - 1]:
 return False

return True

def test(self):
 """
 测试函数
 """

 # 测试用例 1
 nums1 = [1, 1, 1, 2, 2, 3]
 expected1 = [1, 1, 2, 2, 3]
 print("测试用例 1:")
 print(f"原数组: {nums1}")
 len1 = self.remove_duplicates_optimized(nums1)
 print(f"处理后: {nums1[:len1]}")
 print(f"长度: {len1}")
 print(f"验证结果: {self.validate_result(nums1, expected1, len1)}")
 print()

 # 测试用例 2
 nums2 = [0, 0, 1, 1, 1, 1, 2, 3, 3]
 expected2 = [0, 0, 1, 1, 2, 3, 3]
 print("测试用例 2:")
 print(f"原数组: {nums2}")
 len2 = self.remove_duplicates_optimized(nums2)
 print(f"处理后: {nums2[:len2]}")
 print(f"长度: {len2}")
 print(f"验证结果: {self.validate_result(nums2, expected2, len2)}")
 print()

 # 测试用例 3 - 边界情况: 数组长度小于等于 2
 nums3 = [1, 1]
 print("测试用例 3 (数组长度为 2):")
 print(f"原数组: {nums3}")
 len3 = self.remove_duplicates_optimized(nums3)
```

```

print(f"处理后: {nums3[:len3]}")
print(f"长度: {len3}")
print()

测试用例 4 - 边界情况: 所有元素都相同
nums4 = [2, 2, 2, 2, 2]
expected4 = [2, 2]
print("测试用例 4 (所有元素相同) :")
print(f"原数组: {nums4}")
len4 = self.remove_duplicates_optimized(nums4)
print(f"处理后: {nums4[:len4]}")
print(f"长度: {len4}")
print(f"验证结果: {self.validate_result(nums4, expected4, len4)}")
print()

测试用例 5 - 边界情况: 没有重复元素
nums5 = [1, 2, 3, 4, 5]
print("测试用例 5 (无重复元素) :")
print(f"原数组: {nums5}")
len5 = self.remove_duplicates_optimized(nums5)
print(f"处理后: {nums5[:len5]}")
print(f"长度: {len5}")
print(f"验证结果: {self.validate_result(nums5, nums5[:len5], len5)}")
print()

def performance_test(self):
 """
 性能测试
 """

 # 创建一个大数组进行性能测试
 size = 1000000
 large_array = [i // 10 for i in range(size)] # 每个数字重复 10 次

 # 测试解法一的性能
 array1 = large_array.copy()
 start_time = time.time()
 len1 = self.remove_duplicates(array1)
 end_time = time.time()
 duration = (end_time - start_time) * 1000 # 转换为毫秒
 print(f"解法一 (快慢指针+计数) 耗时: {duration:.2f}ms, 处理后长度: {len1}")

 # 测试解法二的性能
 array2 = large_array.copy()

```

```

start_time = time.time()
len2 = self.remove_duplicates_optimized(array2)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法二（优化快慢指针）耗时: {duration:.2f}ms, 处理后长度: {len2}")

测试解法三的性能
array3 = large_array.copy()
start_time = time.time()
len3 = self.remove_duplicates_generic(array3, 2)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法三（通用解法）耗时: {duration:.2f}ms, 处理后长度: {len3}")

验证所有解法结果一致
results_consistent = (len1 == len2 and len2 == len3)
if results_consistent:
 for i in range(len1):
 if array1[i] != array2[i] or array1[i] != array3[i]:
 results_consistent = False
 break
print(f"所有解法结果一致: {results_consistent}")

主函数
if __name__ == "__main__":
 solution = Solution()

 print("== 测试用例 ==")
 solution.test()

 print("== 性能测试 ==")
 solution.performance_test()

```

=====

文件: Code26\_MoveZeroes.cpp

=====

```

#include <iostream>
#include <vector>
#include <chrono>

/***
 * LeetCode 283. 移动零 (Move Zeroes)

```

```
*
* 题目描述:
* 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
* 请注意，必须在不复制数组的情况下原地对数组进行操作。
*
* 示例 1:
* 输入: nums = [0, 1, 0, 3, 12]
* 输出: [1, 3, 12, 0, 0]
*
* 示例 2:
* 输入: nums = [0]
* 输出: [0]
*
* 提示:
* 1 <= nums.length <= 10^4
* -2^31 <= nums[i] <= 2^31 - 1
*
* 题目链接: https://leetcode.com/problems/move-zeroes/
*
* 解题思路:
* 这道题可以使用双指针的方法来解决:
*
* 方法一（双指针）:
* 1. 使用两个指针 slow 和 fast，初始都指向 0
* 2. fast 指针用于遍历整个数组，当遇到非零元素时，将其移动到 slow 指向的位置，然后 slow++
* 3. 遍历结束后，将 slow 到数组末尾的所有元素都设为 0
*
* 方法二（优化的双指针）:
* 1. 同样使用两个指针 slow 和 fast，初始都指向 0
* 2. 当 fast 指向非零元素时，如果 slow != fast，交换 nums[slow] 和 nums[fast]，然后 slow++
* 3. 这种方法避免了不必要的赋值操作，只在需要时进行交换
*
* 最优解是方法二，时间复杂度 O(n)，空间复杂度 O(1)。
*/
```

```
/**
* 解法一：双指针基础版
*
* @param nums 输入数组
* @throws std::invalid_argument 如果输入数组为 null
*/
void moveZeroes(std::vector<int>& nums) {
 // 参数校验
```

```
if (nums.empty()) {
 return; // 空数组不需要操作
}

// 慢指针，指向当前应该放置非零元素的位置
int slow = 0;

// 快指针遍历整个数组
for (int fast = 0; fast < nums.size(); fast++) {
 // 如果当前元素不为 0，将其移到慢指针位置
 if (nums[fast] != 0) {
 nums[slow] = nums[fast];
 slow++;
 }
}

// 将 slow 之后的所有元素都设为 0
for (int i = slow; i < nums.size(); i++) {
 nums[i] = 0;
}

}

/***
 * 解法二：优化的双指针（最优解）
 *
 * @param nums 输入数组
 */
void moveZeroesOptimized(std::vector<int>& nums) {
 // 参数校验
 if (nums.empty()) {
 return; // 空数组不需要操作
 }

 // 慢指针，指向当前应该放置非零元素的位置
 int slow = 0;

 // 快指针遍历整个数组
 for (int fast = 0; fast < nums.size(); fast++) {
 // 如果当前元素不为 0，且 slow 和 fast 不同，交换它们
 if (nums[fast] != 0) {
 if (slow != fast) {
 std::swap(nums[slow], nums[fast]);
 }
 }
 }
}
```

```

 slow++;
 }
}

}

/***
 * 解法三：一次遍历的另一种实现
 *
 * @param nums 输入数组
 */
void moveZeroesOnePass(std::vector<int>& nums) {
 // 参数校验
 if (nums.empty()) {
 return; // 空数组不需要操作
 }

 int lastNonZeroFoundAt = 0;

 // 将所有非零元素移到数组前面
 for (int i = 0; i < nums.size(); i++) {
 if (nums[i] != 0) {
 nums[lastNonZeroFoundAt++] = nums[i];
 }
 }

 // 填充剩余位置为 0
 for (int i = lastNonZeroFoundAt; i < nums.size(); i++) {
 nums[i] = 0;
 }
}

/***
 * 打印向量内容
 */
void printVector(const std::vector<int>& vec) {
 std::cout << "[";
 for (size_t i = 0; i < vec.size(); i++) {
 std::cout << vec[i];
 if (i < vec.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]";
}

```

```
}

/***
 * 验证结果是否正确
 */
bool validateResult(const std::vector<int>& original, const std::vector<int>& expected) {
 if (original.size() != expected.size()) {
 return false;
 }

 // 验证所有元素相等
 for (size_t i = 0; i < original.size(); i++) {
 if (original[i] != expected[i]) {
 return false;
 }
 }

 // 验证 0 都在末尾
 bool zeroStarted = false;
 for (int num : original) {
 if (zeroStarted && num != 0) {
 return false; // 发现 0 后面有非零元素
 }

 if (num == 0) {
 zeroStarted = true;
 }
 }

 return true;
}

/***
 * 测试函数
 */
void test() {
 // 测试用例 1
 std::vector<int> nums1 = {0, 1, 0, 3, 12};
 std::vector<int> expected1 = {1, 3, 12, 0, 0};
 std::cout << "测试用例 1:\n";
 std::cout << "原数组: ";
 printVector(nums1);
 std::cout << std::endl;
 moveZeroesOptimized(nums1);
```

```
std::cout << "处理后: ";
printVector(nums1);
std::cout << std::endl;
std::cout << "验证结果: " << (validateResult(nums1, expected1) ? "true" : "false") <<
std::endl;
std::cout << std::endl;

// 测试用例 2
std::vector<int> nums2 = {0};
std::vector<int> expected2 = {0};
std::cout << "测试用例 2:\n";
std::cout << "原数组: ";
printVector(nums2);
std::cout << std::endl;
moveZeroesOptimized(nums2);
std::cout << "处理后: ";
printVector(nums2);
std::cout << std::endl;
std::cout << "验证结果: " << (validateResult(nums2, expected2) ? "true" : "false") <<
std::endl;
std::cout << std::endl;

// 测试用例 3 - 边界情况: 没有零元素
std::vector<int> nums3 = {1, 2, 3, 4, 5};
std::vector<int> expected3 = {1, 2, 3, 4, 5};
std::cout << "测试用例 3 (无零元素):\n";
std::cout << "原数组: ";
printVector(nums3);
std::cout << std::endl;
moveZeroesOptimized(nums3);
std::cout << "处理后: ";
printVector(nums3);
std::cout << std::endl;
std::cout << "验证结果: " << (validateResult(nums3, expected3) ? "true" : "false") <<
std::endl;
std::cout << std::endl;

// 测试用例 4 - 边界情况: 所有元素都是零
std::vector<int> nums4 = {0, 0, 0, 0, 0};
std::vector<int> expected4 = {0, 0, 0, 0, 0};
std::cout << "测试用例 4 (全零元素):\n";
std::cout << "原数组: ";
printVector(nums4);
```

```
 std::cout << std::endl;
 moveZeroesOptimized(nums4);
 std::cout << "处理后: ";
 printVector(nums4);
 std::cout << std::endl;
 std::cout << "验证结果: " << (validateResult(nums4, expected4) ? "true" : "false") <<
std::endl;
 std::cout << std::endl;

// 测试用例 5 - 边界情况: 零元素在开头
std::vector<int> nums5 = {0, 0, 1, 2, 3};
std::vector<int> expected5 = {1, 2, 3, 0, 0};
std::cout << "测试用例 5 (零在开头) :\n";
std::cout << "原数组: ";
printVector(nums5);
std::cout << std::endl;
moveZeroesOptimized(nums5);
std::cout << "处理后: ";
printVector(nums5);
std::cout << std::endl;
std::cout << "验证结果: " << (validateResult(nums5, expected5) ? "true" : "false") <<
std::endl;
 std::cout << std::endl;

// 测试用例 6 - 边界情况: 零元素在末尾
std::vector<int> nums6 = {1, 2, 3, 0, 0};
std::vector<int> expected6 = {1, 2, 3, 0, 0};
std::cout << "测试用例 6 (零在末尾) :\n";
std::cout << "原数组: ";
printVector(nums6);
std::cout << std::endl;
moveZeroesOptimized(nums6);
std::cout << "处理后: ";
printVector(nums6);
std::cout << std::endl;
std::cout << "验证结果: " << (validateResult(nums6, expected6) ? "true" : "false") <<
std::endl;
 std::cout << std::endl;
}

/**
 * 性能测试
*/
```

```
void performanceTest() {
 // 创建一个大数组进行性能测试
 const size_t size = 1000000;
 std::vector<int> largeArray(size);

 // 生成测试数据：交替放置 0 和 1
 for (size_t i = 0; i < size; i++) {
 largeArray[i] = i % 2; // 0, 1, 0, 1, ...
 }

 // 测试解法一的性能
 std::vector<int> array1 = largeArray;
 auto start = std::chrono::high_resolution_clock::now();
 moveZeroes(array1);
 auto end = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
 std::cout << "解法一（基础双指针）耗时：" << duration << "ms" << std::endl;

 // 测试解法二的性能
 std::vector<int> array2 = largeArray;
 start = std::chrono::high_resolution_clock::now();
 moveZeroesOptimized(array2);
 end = std::chrono::high_resolution_clock::now();
 duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
 std::cout << "解法二（优化双指针）耗时：" << duration << "ms" << std::endl;

 // 测试解法三的性能
 std::vector<int> array3 = largeArray;
 start = std::chrono::high_resolution_clock::now();
 moveZeroesOnePass(array3);
 end = std::chrono::high_resolution_clock::now();
 duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
 std::cout << "解法三（一次遍历）耗时：" << duration << "ms" << std::endl;

 // 验证所有解法结果一致
 bool resultsConsistent = true;
 for (size_t i = 0; i < size; i++) {
 if (array1[i] != array2[i] || array1[i] != array3[i]) {
 resultsConsistent = false;
 break;
 }
 }
 std::cout << "所有解法结果一致：" << (resultsConsistent ? "true" : "false") << std::endl;
```

```

// 验证结果正确性
bool isCorrect = true;
bool zeroStarted = false;
for (int num : array1) {
 if (zeroStarted && num != 0) {
 isCorrect = false;
 break;
 }
 if (num == 0) {
 zeroStarted = true;
 }
}
std::cout << "结果正确: " << (isCorrect ? "true" : "false") << std::endl;
}

int main() {
 std::cout << "==== 测试用例 ===" << std::endl;
 test();

 std::cout << "==== 性能测试 ===" << std::endl;
 performanceTest();

 return 0;
}

```

=====

文件: Code26\_MoveZeroes.java

=====

```

package class050;

import java.util.Arrays;
import java.util.concurrent.TimeUnit;

/**
 * LeetCode 283. 移动零 (Move Zeroes)
 *
 * 题目描述:
 * 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。
 * 请注意，必须在不复制数组的情况下原地对数组进行操作。
 *
 * 示例 1:

```

```
* 输入: nums = [0, 1, 0, 3, 12]
* 输出: [1, 3, 12, 0, 0]
*
* 示例 2:
* 输入: nums = [0]
* 输出: [0]
*
* 提示:
* 1 <= nums.length <= 10^4
* -2^31 <= nums[i] <= 2^31 - 1
*
* 题目链接: https://leetcode.com/problems/move-zeroes/
*
* 解题思路:
* 这道题可以使用双指针的方法来解决:
*
* 方法一 (双指针):
* 1. 使用两个指针 slow 和 fast, 初始都指向 0
* 2. fast 指针用于遍历整个数组, 当遇到非零元素时, 将其移动到 slow 指向的位置, 然后 slow++
* 3. 遍历结束后, 将 slow 到数组末尾的所有元素都设为 0
*
* 方法二 (优化的双指针):
* 1. 同样使用两个指针 slow 和 fast, 初始都指向 0
* 2. 当 fast 指向非零元素时, 如果 slow != fast, 交换 nums[slow] 和 nums[fast], 然后 slow++
* 3. 这种方法避免了不必要的赋值操作, 只在需要时进行交换
*
* 最优解是方法二, 时间复杂度 O(n), 空间复杂度 O(1)。
*/

```

```
public class Code26_MoveZeroes {

 /**
 * 解法一: 双指针基础版
 *
 * @param nums 输入数组
 * @throws IllegalArgumentException 如果输入数组为 null
 */
 public static void moveZeroes(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 }
}
```

```

// 慢指针，指向当前应该放置非零元素的位置
int slow = 0;

// 快指针遍历整个数组
for (int fast = 0; fast < nums.length; fast++) {
 // 如果当前元素不为 0，将其移到慢指针位置
 if (nums[fast] != 0) {
 nums[slow] = nums[fast];
 slow++;
 }
}

// 将 slow 之后的所有元素都设为 0
for (int i = slow; i < nums.length; i++) {
 nums[i] = 0;
}

}

/***
 * 解法二：优化的双指针（最优解）
 *
 * @param nums 输入数组
 */
public static void moveZeroesOptimized(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 // 慢指针，指向当前应该放置非零元素的位置
 int slow = 0;

 // 快指针遍历整个数组
 for (int fast = 0; fast < nums.length; fast++) {
 // 如果当前元素不为 0，且 slow 和 fast 不同，交换它们
 if (nums[fast] != 0) {
 if (slow != fast) {
 swap(nums, slow, fast);
 }
 slow++;
 }
 }
}

```

```
/**
 * 解法三：一次遍历的另一种实现
 *
 * @param nums 输入数组
 */
public static void moveZeroesOnePass(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 int lastNonZeroFoundAt = 0;

 // 将所有非零元素移到数组前面
 for (int i = 0; i < nums.length; i++) {
 if (nums[i] != 0) {
 nums[lastNonZeroFoundAt++] = nums[i];
 }
 }

 // 填充剩余位置为 0
 for (int i = lastNonZeroFoundAt; i < nums.length; i++) {
 nums[i] = 0;
 }
}

/**
 * 交换数组中的两个元素
 */
private static void swap(int[] nums, int i, int j) {
 int temp = nums[i];
 nums[i] = nums[j];
 nums[j] = temp;
}

/**
 * 验证结果是否正确
 */
public static boolean validateResult(int[] original, int[] expected) {
 if (original.length != expected.length) {
 return false;
 }
```

```
// 验证所有元素相等
for (int i = 0; i < original.length; i++) {
 if (original[i] != expected[i]) {
 return false;
 }
}

// 验证 0 都在末尾
boolean zeroStarted = false;
for (int num : original) {
 if (zeroStarted && num != 0) {
 return false; // 发现 0 后面有非零元素
 }
 if (num == 0) {
 zeroStarted = true;
 }
}

return true;
}

/***
 * 测试函数
 */
public static void test() {
 // 测试用例 1
 int[] nums1 = {0, 1, 0, 3, 12};
 int[] expected1 = {1, 3, 12, 0, 0};
 System.out.println("测试用例 1:");
 System.out.print("原数组: ");
 System.out.println(Arrays.toString(nums1));
 moveZeroesOptimized(nums1);
 System.out.print("处理后: ");
 System.out.println(Arrays.toString(nums1));
 System.out.println("验证结果: " + validateResult(nums1, expected1));
 System.out.println();

 // 测试用例 2
 int[] nums2 = {0};
 int[] expected2 = {0};
 System.out.println("测试用例 2:");
 System.out.print("原数组: ");
```

```
System.out.println(Arrays.toString(nums2));
moveZeroesOptimized(nums2);
System.out.print("处理后: ");
System.out.println(Arrays.toString(nums2));
System.out.println("验证结果: " + validateResult(nums2, expected2));
System.out.println();

// 测试用例 3 - 边界情况: 没有零元素
int[] nums3 = {1, 2, 3, 4, 5};
int[] expected3 = {1, 2, 3, 4, 5};
System.out.println("测试用例 3 (无零元素) :");
System.out.print("原数组: ");
System.out.println(Arrays.toString(nums3));
moveZeroesOptimized(nums3);
System.out.print("处理后: ");
System.out.println(Arrays.toString(nums3));
System.out.println("验证结果: " + validateResult(nums3, expected3));
System.out.println();

// 测试用例 4 - 边界情况: 所有元素都是零
int[] nums4 = {0, 0, 0, 0, 0};
int[] expected4 = {0, 0, 0, 0, 0};
System.out.println("测试用例 4 (全零元素) :");
System.out.print("原数组: ");
System.out.println(Arrays.toString(nums4));
moveZeroesOptimized(nums4);
System.out.print("处理后: ");
System.out.println(Arrays.toString(nums4));
System.out.println("验证结果: " + validateResult(nums4, expected4));
System.out.println();

// 测试用例 5 - 边界情况: 零元素在开头
int[] nums5 = {0, 0, 1, 2, 3};
int[] expected5 = {1, 2, 3, 0, 0};
System.out.println("测试用例 5 (零在开头) :");
System.out.print("原数组: ");
System.out.println(Arrays.toString(nums5));
moveZeroesOptimized(nums5);
System.out.print("处理后: ");
System.out.println(Arrays.toString(nums5));
System.out.println("验证结果: " + validateResult(nums5, expected5));
System.out.println();
```

```
// 测试用例 6 - 边界情况: 零元素在末尾
int[] nums6 = {1, 2, 3, 0, 0};
int[] expected6 = {1, 2, 3, 0, 0};
System.out.println("测试用例 6 (零在末尾) :");
System.out.print("原数组: ");
System.out.println(Arrays.toString(nums6));
moveZeroesOptimized(nums6);
System.out.print("处理后: ");
System.out.println(Arrays.toString(nums6));
System.out.println("验证结果: " + validateResult(nums6, expected6));
System.out.println();
}

/**
 * 性能测试
 */
public static void performanceTest() {
 // 创建一个大数组进行性能测试
 int size = 1000000;
 int[] largeArray = new int[size];

 // 生成测试数据: 交替放置 0 和 1
 for (int i = 0; i < size; i++) {
 largeArray[i] = i % 2; // 0, 1, 0, 1, ...
 }

 // 测试解法一的性能
 int[] array1 = Arrays.copyOf(largeArray, size);
 long startTime = System.nanoTime();
 moveZeroes(array1);
 long endTime = System.nanoTime();
 long duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
 System.out.println("解法一 (基础双指针) 耗时: " + duration + "ms");

 // 测试解法二的性能
 int[] array2 = Arrays.copyOf(largeArray, size);
 startTime = System.nanoTime();
 moveZeroesOptimized(array2);
 endTime = System.nanoTime();
 duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
 System.out.println("解法二 (优化双指针) 耗时: " + duration + "ms");

 // 测试解法三的性能
}
```

```
int[] array3 = Arrays.copyOf(largeArray, size);
startTime = System.nanoTime();
moveZeroesOnePass(array3);
endTime = System.nanoTime();
duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法三（一次遍历）耗时：" + duration + "ms");

// 验证所有解法结果一致
boolean resultsConsistent = true;
for (int i = 0; i < size; i++) {
 if (array1[i] != array2[i] || array1[i] != array3[i]) {
 resultsConsistent = false;
 break;
 }
}
System.out.println("所有解法结果一致：" + resultsConsistent);

// 验证结果正确性
boolean isCorrect = true;
boolean zeroStarted = false;
for (int num : array1) {
 if (zeroStarted && num != 0) {
 isCorrect = false;
 break;
 }
 if (num == 0) {
 zeroStarted = true;
 }
}
System.out.println("结果正确：" + isCorrect);
}

public static void main(String[] args) {
 System.out.println("== 测试用例 ==");
 test();

 System.out.println("== 性能测试 ==");
 performanceTest();
}
=====
```

文件: Code26\_MoveZeroes.py

```
=====
from typing import List
import time

class Solution:
 """
 LeetCode 283. 移动零 (Move Zeroes)
```

题目描述:

给定一个数组 `nums`, 编写一个函数将所有 0 移动到数组的末尾, 同时保持非零元素的相对顺序。  
请注意, 必须在不复制数组的情况下原地对数组进行操作。

示例 1:

输入: `nums = [0, 1, 0, 3, 12]`

输出: `[1, 3, 12, 0, 0]`

示例 2:

输入: `nums = [0]`

输出: `[0]`

提示:

`1 <= nums.length <= 10^4`

`-2^31 <= nums[i] <= 2^31 - 1`

题目链接: <https://leetcode.com/problems/move-zeroes/>

解题思路:

这道题可以使用双指针的方法来解决:

方法一 (双指针):

1. 使用两个指针 `slow` 和 `fast`, 初始都指向 0
2. `fast` 指针用于遍历整个数组, 当遇到非零元素时, 将其移动到 `slow` 指向的位置, 然后 `slow++`
3. 遍历结束后, 将 `slow` 到数组末尾的所有元素都设为 0

方法二 (优化的双指针):

1. 同样使用两个指针 `slow` 和 `fast`, 初始都指向 0
2. 当 `fast` 指向非零元素时, 如果 `slow != fast`, 交换 `nums[slow]` 和 `nums[fast]`, 然后 `slow++`
3. 这种方法避免了不必要的赋值操作, 只在需要时进行交换

最优解是方法二, 时间复杂度  $O(n)$ , 空间复杂度  $O(1)$ 。

"""

```
def move_zeroes(self, nums: List[int]) -> None:
 """
 解法一：双指针基础版

 Args:
 nums: 输入数组

 Returns:
 None: 原地修改数组

 时间复杂度: O(n) - 只需要一次遍历
 空间复杂度: O(1) - 只使用常量额外空间
 """

 # 参数校验
 if nums is None:
 raise ValueError("输入数组不能为 None")

 # 慢指针，指向当前应该放置非零元素的位置
 slow = 0

 # 快指针遍历整个数组
 for fast in range(len(nums)):
 # 如果当前元素不为 0，将其移到慢指针位置
 if nums[fast] != 0:
 nums[slow] = nums[fast]
 slow += 1

 # 将 slow 之后的所有元素都设为 0
 for i in range(slow, len(nums)):
 nums[i] = 0
```

```
def move_zeroes_optimized(self, nums: List[int]) -> None:
 """
```

解法二：优化的双指针（最优解）

Args:  
 nums: 输入数组

Returns:  
 None: 原地修改数组

时间复杂度: O(n) - 只需要一次遍历  
空间复杂度: O(1) - 只使用常量额外空间

```
"""
参数校验
if nums is None:
 raise ValueError("输入数组不能为 None")

慢指针，指向当前应该放置非零元素的位置
slow = 0

快指针遍历整个数组
for fast in range(len(nums)):
 # 如果当前元素不为 0，且 slow 和 fast 不同，交换它们
 if nums[fast] != 0:
 if slow != fast:
 nums[slow], nums[fast] = nums[fast], nums[slow]
 slow += 1
```

```
def move_zeroes_one_pass(self, nums: List[int]) -> None:
```

解法三：一次遍历的另一种实现

Args:

nums: 输入数组

Returns:

None: 原地修改数组

时间复杂度:  $O(n)$  - 只需要一次遍历

空间复杂度:  $O(1)$  - 只使用常量额外空间

"""

# 参数校验

```
if nums is None:
 raise ValueError("输入数组不能为 None")
```

```
last_non_zero_found_at = 0
```

# 将所有非零元素移到数组前面

```
for i in range(len(nums)):
 if nums[i] != 0:
 nums[last_non_zero_found_at] = nums[i]
 last_non_zero_found_at += 1
```

# 填充剩余位置为 0

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

```
nums[i] = 0

def validate_result(self, original: List[int], expected: List[int]) -> bool:
 """
 验证结果是否正确

 Args:
 original: 处理后的数组
 expected: 期望的结果数组

 Returns:
 bool: 如果结果正确返回 True, 否则返回 False
 """
 if len(original) != len(expected):
 return False

 # 验证所有元素相等
 for i in range(len(original)):
 if original[i] != expected[i]:
 return False

 # 验证 0 都在末尾
 zero_started = False
 for num in original:
 if zero_started and num != 0:
 return False # 发现 0 后面有非零元素
 if num == 0:
 zero_started = True

 return True

def test(self):
 """
 测试函数
 """
 # 测试用例 1
 nums1 = [0, 1, 0, 3, 12]
 expected1 = [1, 3, 12, 0, 0]
 print("测试用例 1:")
 print(f"原数组: {nums1}")
 self.move_zeroes_optimized(nums1)
 print(f"处理后: {nums1}")
 print(f"验证结果: {self.validate_result(nums1, expected1)}")
```

```
print()

测试用例 2
nums2 = [0]
expected2 = [0]
print("测试用例 2:")
print(f"原数组: {nums2}")
self.move_zeroes_optimized(nums2)
print(f"处理后: {nums2}")
print(f"验证结果: {self.validate_result(nums2, expected2)}")
print()

测试用例 3 - 边界情况: 没有零元素
nums3 = [1, 2, 3, 4, 5]
expected3 = [1, 2, 3, 4, 5]
print("测试用例 3 (无零元素) :")
print(f"原数组: {nums3}")
self.move_zeroes_optimized(nums3)
print(f"处理后: {nums3}")
print(f"验证结果: {self.validate_result(nums3, expected3)}")
print()

测试用例 4 - 边界情况: 所有元素都是零
nums4 = [0, 0, 0, 0, 0]
expected4 = [0, 0, 0, 0, 0]
print("测试用例 4 (全零元素) :")
print(f"原数组: {nums4}")
self.move_zeroes_optimized(nums4)
print(f"处理后: {nums4}")
print(f"验证结果: {self.validate_result(nums4, expected4)}")
print()

测试用例 5 - 边界情况: 零元素在开头
nums5 = [0, 0, 1, 2, 3]
expected5 = [1, 2, 3, 0, 0]
print("测试用例 5 (零在开头) :")
print(f"原数组: {nums5}")
self.move_zeroes_optimized(nums5)
print(f"处理后: {nums5}")
print(f"验证结果: {self.validate_result(nums5, expected5)}")
print()

测试用例 6 - 边界情况: 零元素在末尾
```

```
nums6 = [1, 2, 3, 0, 0]
expected6 = [1, 2, 3, 0, 0]
print("测试用例 6 (零在末尾) :")
print(f"原数组: {nums6}")
self.move_zeroes_optimized(nums6)
print(f"处理后: {nums6}")
print(f"验证结果: {self.validate_result(nums6, expected6)}")
print()

def performance_test(self):
 """
 性能测试
 """
 # 创建一个大数组进行性能测试
 size = 1000000
 large_array = [i % 2 for i in range(size)] # 交替放置 0 和 1

 # 测试解法一的性能
 array1 = large_array.copy()
 start_time = time.time()
 self.move_zeroes(array1)
 end_time = time.time()
 duration = (end_time - start_time) * 1000 # 转换为毫秒
 print(f"解法一 (基础双指针) 耗时: {duration:.2f}ms")

 # 测试解法二的性能
 array2 = large_array.copy()
 start_time = time.time()
 self.move_zeroes_optimized(array2)
 end_time = time.time()
 duration = (end_time - start_time) * 1000 # 转换为毫秒
 print(f"解法二 (优化双指针) 耗时: {duration:.2f}ms")

 # 测试解法三的性能
 array3 = large_array.copy()
 start_time = time.time()
 self.move_zeroes_one_pass(array3)
 end_time = time.time()
 duration = (end_time - start_time) * 1000 # 转换为毫秒
 print(f"解法三 (一次遍历) 耗时: {duration:.2f}ms")

 # 验证所有解法结果一致
 results_consistent = True
```

```

for i in range(size):
 if array1[i] != array2[i] or array1[i] != array3[i]:
 results_consistent = False
 break
 print(f"所有解法结果一致: {results_consistent}")

验证结果正确性
is_correct = True
zero_started = False
for num in array1:
 if zero_started and num != 0:
 is_correct = False
 break
 if num == 0:
 zero_started = True
print(f"结果正确: {is_correct}")

主函数
if __name__ == "__main__":
 solution = Solution()

 print("==== 测试用例 ===")
 solution.test()

 print("==== 性能测试 ===")
 solution.performance_test()

```

=====

文件: Code27\_ContainerWithMostWater.cpp

=====

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>

/***
 * LeetCode 11. 盛最多水的容器 (Container With Most Water)
 *
 * 题目描述:
 * 给定一个长度为 n 的整数数组 height。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i])。
 * 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

```

```
* 返回容器可以储存的最大水量。
* 说明：你不能倾斜容器。
*
* 示例 1：
* 输入：[1, 8, 6, 2, 5, 4, 8, 3, 7]
* 输出：49
* 解释：图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。
*
* 示例 2：
* 输入：height = [1, 1]
* 输出：1
*
* 提示：
* n == height.length
* 2 <= n <= 10^5
* 0 <= height[i] <= 10^4
*
* 题目链接：https://leetcode.com/problems/container-with-most-water/
*
* 解题思路：
* 这道题可以使用双指针的方法来解决：
*
* 方法一（暴力解法）：
* 遍历所有可能的两条线的组合，计算每个组合能容纳的水量，找出最大值。
* 时间复杂度：O(n^2)，空间复杂度：O(1)
*
* 方法二（双指针）：
* 1. 初始化两个指针 left 和 right 分别指向数组的开头和结尾
* 2. 计算当前指针所指两条线能容纳的水量：min(height[left], height[right]) * (right - left)
* 3. 更新最大水量
* 4. 移动较短的那条线对应的指针（因为如果移动较长的线，容纳的水量只会更小）
* 5. 重复步骤 2-4，直到两个指针相遇
* 时间复杂度：O(n)，空间复杂度：O(1)
*
* 最优解是方法二，时间复杂度 O(n)，空间复杂度 O(1)。
```

```
*/
```

```
/**
* 解法一：暴力解法（不推荐，可能会超时）
*
* @param height 输入数组
* @return 最大盛水量
```

```

* @throws std::invalid_argument 如果输入数组长度小于 2
*/
int maxAreaBruteForce(const std::vector<int>& height) {
 // 参数校验
 if (height.size() < 2) {
 throw std::invalid_argument("输入数组长度必须至少为 2");
 }

 int maxArea = 0;
 // 遍历所有可能的两条线的组合
 for (size_t i = 0; i < height.size(); i++) {
 for (size_t j = i + 1; j < height.size(); j++) {
 // 计算当前组合的盛水量
 int currentArea = std::min(height[i], height[j]) * static_cast<int>(j - i);
 // 更新最大盛水量
 maxArea = std::max(maxArea, currentArea);
 }
 }
 return maxArea;
}

/***
* 解法二：双指针（最优解）
*
* @param height 输入数组
* @return 最大盛水量
*/
int maxArea(const std::vector<int>& height) {
 // 参数校验
 if (height.size() < 2) {
 throw std::invalid_argument("输入数组长度必须至少为 2");
 }

 int maxArea = 0;
 int left = 0; // 左指针，初始指向数组开头
 int right = static_cast<int>(height.size() - 1); // 右指针，初始指向数组结尾

 while (left < right) {
 // 计算当前盛水量
 int currentHeight = std::min(height[left], height[right]);
 int currentWidth = right - left;
 int currentArea = currentHeight * currentWidth;

```

```

// 更新最大盛水量
maxArea = std::max(maxArea, currentArea);

// 移动较短的那条线对应的指针
if (height[left] < height[right]) {
 left++;
} else {
 right--;
}
}

return maxArea;
}

/***
 * 解法三：双指针优化版
 * 跳过相同高度的柱子，减少不必要的计算
 *
 * @param height 输入数组
 * @return 最大盛水量
 */
int maxAreaOptimized(const std::vector<int>& height) {
 // 参数校验
 if (height.size() < 2) {
 throw std::invalid_argument("输入数组长度必须至少为 2");
 }

 int maxArea = 0;
 int left = 0;
 int right = static_cast<int>(height.size() - 1);

 while (left < right) {
 // 计算当前盛水量
 int currentHeight = std::min(height[left], height[right]);
 int currentWidth = right - left;
 int currentArea = currentHeight * currentWidth;

 // 更新最大盛水量
 maxArea = std::max(maxArea, currentArea);

 // 移动较短的那条线对应的指针
 // 跳过相同高度的柱子
 if (height[left] < height[right]) {

```

```

 int currentLeftHeight = height[left];
 while (left < right && height[left] <= currentLeftHeight) {
 left++;
 }
 } else {
 int currentRightHeight = height[right];
 while (left < right && height[right] <= currentRightHeight) {
 right--;
 }
 }
}

return maxArea;
}

/***
 * 打印向量内容
 */
void printVector(const std::vector<int>& vec) {
 std::cout << "[";
 for (size_t i = 0; i < vec.size(); i++) {
 std::cout << vec[i];
 if (i < vec.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]";
}

/***
 * 测试函数
 */
void test() {
 // 测试用例 1
 std::vector<int> height1 = {1, 8, 6, 2, 5, 4, 8, 3, 7};
 int expected1 = 49;
 std::cout << "测试用例 1:\n";
 std::cout << "输入数组: ";
 printVector(height1);
 std::cout << std::endl;
 int result1 = maxArea(height1);
 std::cout << "最大盛水量: " << result1 << std::endl;
 std::cout << "验证结果: " << (result1 == expected1 ? "true" : "false") << std::endl;
}

```

```
std::cout << std::endl;

// 测试用例 2
std::vector<int> height2 = {1, 1};
int expected2 = 1;
std::cout << "测试用例 2:\n";
std::cout << "输入数组: ";
printVector(height2);
std::cout << std::endl;
int result2 = maxArea(height2);
std::cout << "最大盛水量: " << result2 << std::endl;
std::cout << "验证结果: " << (result2 == expected2 ? "true" : "false") << std::endl;
std::cout << std::endl;

// 测试用例 3 - 边界情况: 所有元素递增
std::vector<int> height3 = {1, 2, 3, 4, 5};
int expected3 = 6; // 由索引 0 和 4 的元素组成的容器
std::cout << "测试用例 3 (递增数组):\n";
std::cout << "输入数组: ";
printVector(height3);
std::cout << std::endl;
int result3 = maxArea(height3);
std::cout << "最大盛水量: " << result3 << std::endl;
std::cout << "验证结果: " << (result3 == expected3 ? "true" : "false") << std::endl;
std::cout << std::endl;

// 测试用例 4 - 边界情况: 所有元素递减
std::vector<int> height4 = {5, 4, 3, 2, 1};
int expected4 = 6; // 由索引 0 和 4 的元素组成的容器
std::cout << "测试用例 4 (递减数组):\n";
std::cout << "输入数组: ";
printVector(height4);
std::cout << std::endl;
int result4 = maxArea(height4);
std::cout << "最大盛水量: " << result4 << std::endl;
std::cout << "验证结果: " << (result4 == expected4 ? "true" : "false") << std::endl;
std::cout << std::endl;

// 测试用例 5 - 边界情况: 只有两个元素, 高度不同
std::vector<int> height5 = {3, 5};
int expected5 = 3; // 由索引 0 和 1 的元素组成的容器
std::cout << "测试用例 5 (两个元素):\n";
std::cout << "输入数组: ";
```

```
printVector(height5);

std::cout << std::endl;
int result5 = maxArea(height5);
std::cout << "最大盛水量: " << result5 << std::endl;
std::cout << "验证结果: " << (result5 == expected5 ? "true" : "false") << std::endl;
std::cout << std::endl;

// 测试用例 6 - 边界情况: 包含 0
std::vector<int> height6 = {0, 0, 0, 0, 0};
int expected6 = 0; // 所有元素都是 0, 盛水量为 0
std::cout << "测试用例 6 (全零数组) :\n";
std::cout << "输入数组: ";
printVector(height6);
std::cout << std::endl;
int result6 = maxArea(height6);
std::cout << "最大盛水量: " << result6 << std::endl;
std::cout << "验证结果: " << (result6 == expected6 ? "true" : "false") << std::endl;
std::cout << std::endl;
}

/***
 * 性能测试
 */
void performanceTest() {
 // 创建一个大数组进行性能测试
 const size_t size = 100000;
 std::vector<int> largeArray(size);

 // 生成测试数据: 交替增加和减少
 for (size_t i = 0; i < size; i++) {
 largeArray[i] = i % 100; // 0-99 循环
 }

 // 测试解法二的性能
 auto start = std::chrono::high_resolution_clock::now();
 int result2 = maxArea(largeArray);
 auto end = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
 std::cout << "解法二 (双指针) 耗时: " << duration << "ms, 最大盛水量: " << result2 <<
 std::endl;

 // 测试解法三的性能
 start = std::chrono::high_resolution_clock::now();
}
```

```
int result3 = maxAreaOptimized(largeArray);
end = std::chrono::high_resolution_clock::now();
duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
std::cout << "解法三（优化双指针）耗时：" << duration << "ms, 最大盛水量：" << result3 <<
std::endl;

// 验证两种解法结果一致
bool resultsConsistent = (result2 == result3);
std::cout << "所有解法结果一致：" << (resultsConsistent ? "true" : "false") << std::endl;
}

/***
 * 边界条件测试
 */
void boundaryTest() {
 try {
 // 测试长度为 1 的输入
 maxArea(std::vector<int>{5});
 std::cout << "边界测试失败：长度为 1 的输入没有抛出异常" << std::endl;
 } catch (const std::invalid_argument& e) {
 std::cout << "边界测试通过：长度为 1 的输入正确抛出异常：" << e.what() << std::endl;
 }

 try {
 // 测试空输入
 maxArea(std::vector<int>{});
 std::cout << "边界测试失败：空输入没有抛出异常" << std::endl;
 } catch (const std::invalid_argument& e) {
 std::cout << "边界测试通过：空输入正确抛出异常：" << e.what() << std::endl;
 }
}

int main() {
 std::cout << "==== 测试用例 ===" << std::endl;
 test();

 std::cout << "==== 性能测试 ===" << std::endl;
 performanceTest();

 std::cout << "==== 边界条件测试 ===" << std::endl;
 boundaryTest();

 return 0;
}
```

```
}
```

```
=====
```

文件: Code27\_ContainerWithMostWater.java

```
=====
```

```
package class050;
```

```
import java.util.Arrays;
```

```
import java.util.concurrent.TimeUnit;
```

```
/**
```

```
* LeetCode 11. 盛最多水的容器 (Container With Most Water)
```

```
*
```

```
* 题目描述:
```

```
* 给定一个长度为 n 的整数数组 height。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i])。
```

```
* 找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。
```

```
* 返回容器可以储存的最大水量。
```

```
* 说明：你不能倾斜容器。
```

```
*
```

```
* 示例 1:
```

```
* 输入: [1, 8, 6, 2, 5, 4, 8, 3, 7]
```

```
* 输出: 49
```

```
* 解释: 图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。
```

```
*
```

```
* 示例 2:
```

```
* 输入: height = [1, 1]
```

```
* 输出: 1
```

```
*
```

```
* 提示:
```

```
* n == height.length
```

```
* 2 <= n <= 10^5
```

```
* 0 <= height[i] <= 10^4
```

```
*
```

```
* 题目链接: https://leetcode.com/problems/container-with-most-water/
```

```
*
```

```
* 解题思路:
```

```
* 这道题可以使用双指针的方法来解决:
```

```
*
```

```
* 方法一（暴力解法）:
```

```
* 遍历所有可能的两条线的组合，计算每个组合能容纳的水量，找出最大值。
```

- \* 时间复杂度:  $O(n^2)$ , 空间复杂度:  $O(1)$
- \*
- \* 方法二 (双指针):
- \* 1. 初始化两个指针 left 和 right 分别指向数组的开头和结尾
- \* 2. 计算当前指针所指两条线能容纳的水量:  $\min(\text{height}[left], \text{height}[right]) * (\text{right} - \text{left})$
- \* 3. 更新最大水量
- \* 4. 移动较短的那条线对应的指针 (因为如果移动较长的线, 容纳的水量只会更小)
- \* 5. 重复步骤 2-4, 直到两个指针相遇
- \* 时间复杂度:  $O(n)$ , 空间复杂度:  $O(1)$
- \*
- \* 最优解是方法二, 时间复杂度  $O(n)$ , 空间复杂度  $O(1)$ 。
- \*/

```

public class Code27_ContainerWithMostWater {

 /**
 * 解法一: 暴力解法 (不推荐, 可能会超时)
 *
 * @param height 输入数组
 * @return 最大盛水量
 * @throws IllegalArgumentException 如果输入数组为 null 或长度小于 2
 */
 public static int maxAreaBruteForce(int[] height) {
 // 参数校验
 if (height == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 if (height.length < 2) {
 throw new IllegalArgumentException("输入数组长度必须至少为 2");
 }

 int maxArea = 0;
 // 遍历所有可能的两条线的组合
 for (int i = 0; i < height.length; i++) {
 for (int j = i + 1; j < height.length; j++) {
 // 计算当前组合的盛水量
 int currentArea = Math.min(height[i], height[j]) * (j - i);
 // 更新最大盛水量
 maxArea = Math.max(maxArea, currentArea);
 }
 }
 return maxArea;
 }
}

```

```
/**
 * 解法二：双指针（最优解）
 *
 * @param height 输入数组
 * @return 最大盛水量
 */

public static int maxArea(int[] height) {
 // 参数校验
 if (height == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 if (height.length < 2) {
 throw new IllegalArgumentException("输入数组长度必须至少为 2");
 }

 int maxArea = 0;
 int left = 0; // 左指针，初始指向数组开头
 int right = height.length - 1; // 右指针，初始指向数组结尾

 while (left < right) {
 // 计算当前盛水量
 int currentHeight = Math.min(height[left], height[right]);
 int currentWidth = right - left;
 int currentArea = currentHeight * currentWidth;

 // 更新最大盛水量
 maxArea = Math.max(maxArea, currentArea);

 // 移动较短的那条线对应的指针
 if (height[left] < height[right]) {
 left++;
 } else {
 right--;
 }
 }

 return maxArea;
}

/**
 * 解法三：双指针优化版
 * 跳过相同高度的柱子，减少不必要的计算
```

```
* @param height 输入数组
* @return 最大盛水量
*/
public static int maxAreaOptimized(int[] height) {
 // 参数校验
 if (height == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 if (height.length < 2) {
 throw new IllegalArgumentException("输入数组长度必须至少为 2");
 }

 int maxArea = 0;
 int left = 0;
 int right = height.length - 1;

 while (left < right) {
 // 计算当前盛水量
 int currentHeight = Math.min(height[left], height[right]);
 int currentWidth = right - left;
 int currentArea = currentHeight * currentWidth;

 // 更新最大盛水量
 maxArea = Math.max(maxArea, currentArea);

 // 移动较短的那条线对应的指针
 // 跳过相同高度的柱子
 if (height[left] < height[right]) {
 int currentLeftHeight = height[left];
 while (left < right && height[left] <= currentLeftHeight) {
 left++;
 }
 } else {
 int currentRightHeight = height[right];
 while (left < right && height[right] <= currentRightHeight) {
 right--;
 }
 }
 }

 return maxArea;
}
```

```
/**
 * 测试函数
 */
public static void test() {
 // 测试用例 1
 int[] height1 = {1, 8, 6, 2, 5, 4, 8, 3, 7};
 int expected1 = 49;
 System.out.println("测试用例 1:");
 System.out.print("输入数组: ");
 System.out.println(Arrays.toString(height1));
 int result1 = maxArea(height1);
 System.out.println("最大盛水量: " + result1);
 System.out.println("验证结果: " + (result1 == expected1));
 System.out.println();

 // 测试用例 2
 int[] height2 = {1, 1};
 int expected2 = 1;
 System.out.println("测试用例 2:");
 System.out.print("输入数组: ");
 System.out.println(Arrays.toString(height2));
 int result2 = maxArea(height2);
 System.out.println("最大盛水量: " + result2);
 System.out.println("验证结果: " + (result2 == expected2));
 System.out.println();

 // 测试用例 3 - 边界情况: 所有元素递增
 int[] height3 = {1, 2, 3, 4, 5};
 int expected3 = 6; // 由索引 0 和 4 的元素组成的容器
 System.out.println("测试用例 3 (递增数组) :");
 System.out.print("输入数组: ");
 System.out.println(Arrays.toString(height3));
 int result3 = maxArea(height3);
 System.out.println("最大盛水量: " + result3);
 System.out.println("验证结果: " + (result3 == expected3));
 System.out.println();

 // 测试用例 4 - 边界情况: 所有元素递减
 int[] height4 = {5, 4, 3, 2, 1};
 int expected4 = 6; // 由索引 0 和 4 的元素组成的容器
 System.out.println("测试用例 4 (递减数组) :");
 System.out.print("输入数组: ");
```

```
System.out.println(Arrays.toString(height4));
int result4 = maxArea(height4);
System.out.println("最大盛水量: " + result4);
System.out.println("验证结果: " + (result4 == expected4));
System.out.println();

// 测试用例 5 - 边界情况: 只有两个元素, 高度不同
int[] height5 = {3, 5};
int expected5 = 3; // 由索引 0 和 1 的元素组成的容器
System.out.println("测试用例 5 (两个元素) :");
System.out.print("输入数组: ");
System.out.println(Arrays.toString(height5));
int result5 = maxArea(height5);
System.out.println("最大盛水量: " + result5);
System.out.println("验证结果: " + (result5 == expected5));
System.out.println();

// 测试用例 6 - 边界情况: 包含 0
int[] height6 = {0, 0, 0, 0, 0};
int expected6 = 0; // 所有元素都是 0, 盛水量为 0
System.out.println("测试用例 6 (全零数组) :");
System.out.print("输入数组: ");
System.out.println(Arrays.toString(height6));
int result6 = maxArea(height6);
System.out.println("最大盛水量: " + result6);
System.out.println("验证结果: " + (result6 == expected6));
System.out.println();
}

/***
 * 性能测试
 */
public static void performanceTest() {
 // 创建一个大数组进行性能测试
 int size = 100000;
 int[] largeArray = new int[size];

 // 生成测试数据: 交替增加和减少
 for (int i = 0; i < size; i++) {
 largeArray[i] = i % 100; // 0-99 循环
 }

 // 测试解法二的性能
}
```

```
long startTime = System.nanoTime();
int result2 = maxArea(largeArray);
long endTime = System.nanoTime();
long duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法二（双指针）耗时: " + duration + "ms, 最大盛水量: " + result2);

// 测试解法三的性能
startTime = System.nanoTime();
int result3 = maxAreaOptimized(largeArray);
endTime = System.nanoTime();
duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法三（优化双指针）耗时: " + duration + "ms, 最大盛水量: " +
result3);

// 验证两种解法结果一致
boolean resultsConsistent = (result2 == result3);
System.out.println("所有解法结果一致: " + resultsConsistent);
}

/**
 * 边界条件测试
 */
public static void boundaryTest() {
 try {
 // 测试 null 输入
 maxArea(null);
 System.out.println("边界测试失败: null 输入没有抛出异常");
 } catch (IllegalArgumentException e) {
 System.out.println("边界测试通过: null 输入正确抛出异常");
 }

 try {
 // 测试长度为 1 的输入
 maxArea(new int[] {5});
 System.out.println("边界测试失败: 长度为 1 的输入没有抛出异常");
 } catch (IllegalArgumentException e) {
 System.out.println("边界测试通过: 长度为 1 的输入正确抛出异常");
 }
}

public static void main(String[] args) {
 System.out.println("== 测试用例 ==");
 test();
}
```

```
System.out.println("== 性能测试 ==");
performanceTest();

System.out.println("== 边界条件测试 ==");
boundaryTest();
}

=====
```

文件: Code27\_ContainerWithMostWater.py

```
=====
from typing import List
import time

class Solution:
 """
 LeetCode 11. 盛最多水的容器 (Container With Most Water)
```

题目描述:

给定一个长度为  $n$  的整数数组  $height$ 。有  $n$  条垂线，第  $i$  条线的两个端点是  $(i, 0)$  和  $(i, height[i])$ 。

找出其中的两条线，使得它们与  $x$  轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

示例 1:

输入: [1, 8, 6, 2, 5, 4, 8, 3, 7]

输出: 49

解释: 图中垂直线代表输入数组 [1, 8, 6, 2, 5, 4, 8, 3, 7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

示例 2:

输入: height = [1, 1]

输出: 1

提示:

$n == height.length$

$2 \leq n \leq 10^5$

$0 \leq height[i] \leq 10^4$

题目链接: <https://leetcode.com/problems/container-with-most-water/>

解题思路：

这道题可以使用双指针的方法来解决：

方法一（暴力解法）：

遍历所有可能的两条线的组合，计算每个组合能容纳的水量，找出最大值。

时间复杂度： $O(n^2)$ ，空间复杂度： $O(1)$

方法二（双指针）：

1. 初始化两个指针 `left` 和 `right` 分别指向数组的开头和结尾
2. 计算当前指针所指两条线能容纳的水量：`min(height[left], height[right]) * (right - left)`
3. 更新最大水量
4. 移动较短的那条线对应的指针（因为如果移动较长的线，容纳的水量只会更小）
5. 重复步骤 2-4，直到两个指针相遇

时间复杂度： $O(n)$ ，空间复杂度： $O(1)$

最优解是方法二，时间复杂度  $O(n)$ ，空间复杂度  $O(1)$ 。

"""

```
def max_area_brute_force(self, height: List[int]) -> int:
```

```
 """
```

解法一：暴力解法（不推荐，可能会超时）

Args:

height: 输入数组

Returns:

int: 最大盛水量

Raises:

ValueError: 如果输入数组为 None 或长度小于 2

时间复杂度： $O(n^2)$  – 需要遍历所有可能的两条线的组合

空间复杂度： $O(1)$  – 只使用常量额外空间

"""

# 参数校验

```
if height is None:
```

```
 raise ValueError("输入数组不能为 None")
```

```
if len(height) < 2:
```

```
 raise ValueError("输入数组长度必须至少为 2")
```

```
max_area = 0
```

```
遍历所有可能的两条线的组合
```

```
for i in range(len(height)):
 for j in range(i + 1, len(height)):
 # 计算当前组合的盛水量
 current_area = min(height[i], height[j]) * (j - i)
 # 更新最大盛水量
 max_area = max(max_area, current_area)

return max_area
```

```
def max_area(self, height: List[int]) -> int:
```

```
"""
```

解法二：双指针（最优解）

Args:

height: 输入数组

Returns:

int: 最大盛水量

Raises:

ValueError: 如果输入数组为 None 或长度小于 2

时间复杂度:  $O(n)$  – 只需要一次遍历

空间复杂度:  $O(1)$  – 只使用常量额外空间

```
"""
```

# 参数校验

```
if height is None:
 raise ValueError("输入数组不能为 None")
if len(height) < 2:
 raise ValueError("输入数组长度必须至少为 2")
```

```
max_area = 0
```

```
left = 0 # 左指针, 初始指向数组开头
```

```
right = len(height) - 1 # 右指针, 初始指向数组结尾
```

```
while left < right:
```

# 计算当前盛水量

```
current_height = min(height[left], height[right])
```

```
current_width = right - left
```

```
current_area = current_height * current_width
```

# 更新最大盛水量

```
max_area = max(max_area, current_area)
```

```
移动较短的那条线对应的指针
if height[left] < height[right]:
 left += 1
else:
 right -= 1

return max_area
```

```
def max_area_optimized(self, height: List[int]) -> int:
 """
```

解法三：双指针优化版

跳过相同高度的柱子，减少不必要的计算

Args:

height: 输入数组

Returns:

int: 最大盛水量

Raises:

ValueError: 如果输入数组为 None 或长度小于 2

时间复杂度: O(n) - 只需要一次遍历

空间复杂度: O(1) - 只使用常量额外空间

"""

# 参数校验

if height is None:

raise ValueError("输入数组不能为 None")

if len(height) < 2:

raise ValueError("输入数组长度必须至少为 2")

max\_area = 0

left = 0

right = len(height) - 1

while left < right:

# 计算当前盛水量

current\_height = min(height[left], height[right])

current\_width = right - left

current\_area = current\_height \* current\_width

# 更新最大盛水量

```
max_area = max(max_area, current_area)

移动较短的那条线对应的指针
跳过相同高度的柱子
if height[left] < height[right]:
 current_left_height = height[left]
 while left < right and height[left] <= current_left_height:
 left += 1
else:
 current_right_height = height[right]
 while left < right and height[right] <= current_right_height:
 right -= 1

return max_area

def test(self):
 """
 测试函数
 """
 # 测试用例 1
 height1 = [1, 8, 6, 2, 5, 4, 8, 3, 7]
 expected1 = 49
 print("测试用例 1:")
 print(f"输入数组: {height1}")
 result1 = self.max_area(height1)
 print(f"最大盛水量: {result1}")
 print(f"验证结果: {result1 == expected1}")
 print()

 # 测试用例 2
 height2 = [1, 1]
 expected2 = 1
 print("测试用例 2:")
 print(f"输入数组: {height2}")
 result2 = self.max_area(height2)
 print(f"最大盛水量: {result2}")
 print(f"验证结果: {result2 == expected2}")
 print()

 # 测试用例 3 - 边界情况: 所有元素递增
 height3 = [1, 2, 3, 4, 5]
 expected3 = 6 # 由索引 0 和 4 的元素组成的容器
 print("测试用例 3 (递增数组) :")
```

```
print(f"输入数组: {height3}")
result3 = self.max_area(height3)
print(f"最大盛水量: {result3}")
print(f"验证结果: {result3 == expected3}")
print()

测试用例 4 - 边界情况: 所有元素递减
height4 = [5, 4, 3, 2, 1]
expected4 = 6 # 由索引 0 和 4 的元素组成的容器
print("测试用例 4 (递减数组) :")
print(f"输入数组: {height4}")
result4 = self.max_area(height4)
print(f"最大盛水量: {result4}")
print(f"验证结果: {result4 == expected4}")
print()

测试用例 5 - 边界情况: 只有两个元素, 高度不同
height5 = [3, 5]
expected5 = 3 # 由索引 0 和 1 的元素组成的容器
print("测试用例 5 (两个元素) :")
print(f"输入数组: {height5}")
result5 = self.max_area(height5)
print(f"最大盛水量: {result5}")
print(f"验证结果: {result5 == expected5}")
print()

测试用例 6 - 边界情况: 包含 0
height6 = [0, 0, 0, 0, 0]
expected6 = 0 # 所有元素都是 0, 盛水量为 0
print("测试用例 6 (全零数组) :")
print(f"输入数组: {height6}")
result6 = self.max_area(height6)
print(f"最大盛水量: {result6}")
print(f"验证结果: {result6 == expected6}")
print()

def performance_test(self):
 """
 性能测试
 """
 # 创建一个大数组进行性能测试
 size = 100000
 large_array = [i % 100 for i in range(size)] # 0-99 循环
```

```
测试解法二的性能
array2 = large_array.copy()
start_time = time.time()
result2 = self.max_area(array2)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法二（双指针）耗时: {duration:.2f}ms, 最大盛水量: {result2}")

测试解法三的性能
array3 = large_array.copy()
start_time = time.time()
result3 = self.max_area_optimized(array3)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法三（优化双指针）耗时: {duration:.2f}ms, 最大盛水量: {result3}")

验证两种解法结果一致
results_consistent = (result2 == result3)
print(f"所有解法结果一致: {results_consistent}")

def boundary_test(self):
 """
 边界条件测试
 """
 try:
 # 测试 null 输入
 self.max_area(None)
 print("边界测试失败: None 输入没有抛出异常")
 except ValueError as e:
 print(f"边界测试通过: None 输入正确抛出异常: {e}")

 try:
 # 测试长度为 1 的输入
 self.max_area([5])
 print("边界测试失败: 长度为 1 的输入没有抛出异常")
 except ValueError as e:
 print(f"边界测试通过: 长度为 1 的输入正确抛出异常: {e}")

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

```
print("==> 测试用例 ==>")
solution.test()

print("==> 性能测试 ==>")
solution.performance_test()

print("==> 边界条件测试 ==>")
solution.boundary_test()
```

=====

文件: Code28\_3Sum.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>
#include <chrono>

/***
 * LeetCode 15. 三数之和 (3Sum)
 *
 * 题目描述:
 * 给你一个整数数组 nums，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k，同时满足 nums[i] + nums[j] + nums[k] == 0。请你找出所有和为 0 且不重复的三元组。
 * 注意：答案中不可以包含重复的三元组。
 *
 * 示例 1：
 * 输入：nums = [-1, 0, 1, 2, -1, -4]
 * 输出：[[-1, -1, 2], [-1, 0, 1]]
 *
 * 示例 2：
 * 输入：nums = [0, 1, 1]
 * 输出：[]
 *
 * 示例 3：
 * 输入：nums = [0, 0, 0]
 * 输出：[[0, 0, 0]]
 *
 * 提示：
 * 3 <= nums.length <= 3000
 * -10^5 <= nums[i] <= 10^5
```

```

*
* 题目链接: https://leetcode.com/problems/3sum/
*
* 解题思路:
* 这道题可以使用排序 + 双指针的方法来解决:
*
* 方法一 (暴力解法):
* 使用三层嵌套循环遍历所有可能的三元组，找出和为 0 的三元组。使用 set 去重。
* 时间复杂度: O(n^3)，空间复杂度: O(n)
*
* 方法二 (排序 + 双指针):
* 1. 先对数组进行排序
* 2. 遍历数组，对于每个元素 nums[i]，使用双指针 left 和 right 分别指向 i+1 和数组末尾
* 3. 计算当前三个数的和: sum = nums[i] + nums[left] + nums[right]
* 4. 如果 sum == 0，将这三个数加入结果集，并移动 left 和 right 指针
* - 为了避免重复，需要跳过相同的元素
* 5. 如果 sum < 0，说明需要增大和，移动 left 指针
* 6. 如果 sum > 0，说明需要减小和，移动 right 指针
* 7. 重复步骤 3-6，直到 left >= right
* 时间复杂度: O(n^2)，空间复杂度: O(1) 或 O(log n) (排序的空间复杂度)
*
* 最优解是方法二，时间复杂度 O(n^2)，空间复杂度 O(1) 或 O(log n)。
*/

```

```

/**
 * 解法一: 暴力解法 (不推荐, 会超时)
 *
 * @param nums 输入数组
 * @return 所有和为 0 且不重复的三元组
 * @throws std::invalid_argument 如果输入数组长度小于 3
 */
std::vector<std::vector<int>> threeSumBruteForce(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() < 3) {
 throw std::invalid_argument("输入数组长度必须至少为 3");
 }

 std::set<std::vector<int>> resultSet;
 int n = nums.size();

 // 三层循环遍历所有可能的三元组
 for (size_t i = 0; i < n; i++) {
 for (size_t j = i + 1; j < n; j++) {
 for (size_t k = j + 1; k < n; k++) {
 if (nums[i] + nums[j] + nums[k] == 0) {
 resultSet.insert({nums[i], nums[j], nums[k]});
 }
 }
 }
 }

 return std::vector<std::vector<int>>(resultSet.begin(), resultSet.end());
}

```

```

 for (size_t k = j + 1; k < n; k++) {
 if (nums[i] + nums[j] + nums[k] == 0) {
 // 将三元组排序后加入结果集，利用 set 去重
 std::vector<int> triplet = {nums[i], nums[j], nums[k]};
 std::sort(triplet.begin(), triplet.end());
 resultSet.insert(triplet);
 }
 }
 }

}

return std::vector<std::vector<int>>(resultSet.begin(), resultSet.end());
}

```

```

/**
 * 解法二：排序 + 双指针（最优解）
 *
 * @param nums 输入数组
 * @return 所有和为 0 且不重复的三元组
 */
std::vector<std::vector<int>> threeSum(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() < 3) {
 throw std::invalid_argument("输入数组长度必须至少为 3");
 }

 std::vector<std::vector<int>> result;
 std::vector<int> sortedNums = nums; // 复制数组，避免修改原数组
 int n = sortedNums.size();

 // 对数组进行排序
 std::sort(sortedNums.begin(), sortedNums.end());

 // 遍历数组，固定第一个元素
 for (int i = 0; i < n; i++) {
 // 跳过重复的第一个元素，避免产生重复的三元组
 if (i > 0 && sortedNums[i] == sortedNums[i - 1]) {
 continue;
 }

 // 如果当前元素已经大于 0，由于数组已排序，后面的元素都大于 0，三数之和不可能为 0
 if (sortedNums[i] > 0) {
 break;
 }

 int left = i + 1, right = n - 1;
 while (left < right) {
 int sum = sortedNums[i] + sortedNums[left] + sortedNums[right];
 if (sum == 0) {
 result.push_back({sortedNums[i], sortedNums[left], sortedNums[right]});
 left++;
 right--;
 } else if (sum < 0) {
 left++;
 } else {
 right--;
 }
 }
 }
}

```

```
}

// 初始化双指针
int left = i + 1;
int right = n - 1;

while (left < right) {
 int sum = sortedNums[i] + sortedNums[left] + sortedNums[right];

 if (sum == 0) {
 // 找到一个三元组
 result.push_back({sortedNums[i], sortedNums[left], sortedNums[right]});
 }

 // 跳过重复的左指针元素
 while (left < right && sortedNums[left] == sortedNums[left + 1]) {
 left++;
 }

 // 跳过重复的右指针元素
 while (left < right && sortedNums[right] == sortedNums[right - 1]) {
 right--;
 }

 // 移动两个指针
 left++;
 right--;
}

} else if (sum < 0) {
 // 和小于 0，需要增大和，移动左指针
 left++;
} else {
 // 和大于 0，需要减小和，移动右指针
 right--;
}

}

return result;
}

/**
 * 解法三：优化的双指针实现
 *
 * @param nums 输入数组
 * @return 所有和为 0 且不重复的三元组

```

```
/*
std::vector<std::vector<int>> threeSumOptimized(const std::vector<int>& nums) {
 // 参数校验
 if (nums.size() < 3) {
 throw std::invalid_argument("输入数组长度必须至少为 3");
 }

 std::vector<std::vector<int>> result;
 std::vector<int> sortedNums = nums; // 复制数组，避免修改原数组
 int n = sortedNums.size();

 // 对数组进行排序
 std::sort(sortedNums.begin(), sortedNums.end());

 // 遍历数组，固定第一个元素
 for (int i = 0; i < n - 2; i++) {
 // 跳过重复的第一个元素，避免产生重复的三元组
 if (i > 0 && sortedNums[i] == sortedNums[i - 1]) {
 continue;
 }

 // 剪枝：如果当前元素已经大于 0，三数之和不可能为 0
 if (sortedNums[i] > 0) {
 break;
 }

 // 剪枝：如果当前元素和最大的两个元素之和仍小于 0，跳过
 if (sortedNums[i] + sortedNums[n - 1] + sortedNums[n - 2] < 0) {
 continue;
 }

 // 初始化双指针
 int left = i + 1;
 int right = n - 1;

 while (left < right) {
 int sum = sortedNums[i] + sortedNums[left] + sortedNums[right];

 if (sum == 0) {
 // 找到一个三元组
 result.push_back({sortedNums[i], sortedNums[left], sortedNums[right]});
 }

 // 跳过重复的左指针元素
 while (left < right && sortedNums[left] == sortedNums[left + 1])
 left++;
 while (left < right && sortedNums[right] == sortedNums[right - 1])
 right--;
 }
 }
}
```

```

 while (left < right && sortedNums[left] == sortedNums[left + 1]) {
 left++;
 }
 // 跳过重复的右指针元素
 while (left < right && sortedNums[right] == sortedNums[right - 1]) {
 right--;
 }

 // 移动两个指针
 left++;
 right--;
 } else if (sum < 0) {
 // 和小于 0, 需要增大和, 移动左指针
 left++;
 } else {
 // 和大于 0, 需要减小和, 移动右指针
 right--;
 }
}

}

return result;
}

/***
 * 打印向量内容
 */
void printVector(const std::vector<int>& vec) {
 std::cout << "[";
 for (size_t i = 0; i < vec.size(); i++) {
 std::cout << vec[i];
 if (i < vec.size() - 1) {
 std::cout << ", ";
 }
 }
 std::cout << "]";
}

/***
 * 打印三元组列表
 */
void printTriplets(const std::vector<std::vector<int>>& triplets) {
 std::cout << "[";

```

```
for (size_t i = 0; i < triplets.size(); i++) {
 printVector(triplets[i]);
 if (i < triplets.size() - 1) {
 std::cout << ", ";
 }
}
std::cout << "]";
}

/***
 * 测试函数
 */
void test() {
 // 测试用例 1
 std::vector<int> nums1 = {-1, 0, 1, 2, -1, -4};
 std::vector<std::vector<int>> expected1 = {{-1, -1, 2}, {-1, 0, 1}};
 std::cout << "测试用例 1:\n";
 std::cout << "输入数组: ";
 printVector(nums1);
 std::cout << std::endl;
 std::vector<std::vector<int>> result1 = threeSum(nums1);
 std::cout << "结果: ";
 printTriplets(result1);
 std::cout << std::endl;
 std::cout << "期望: ";
 printTriplets(expected1);
 std::cout << std::endl << std::endl;

 // 测试用例 2
 std::vector<int> nums2 = {0, 1, 1};
 std::vector<std::vector<int>> expected2 = {};
 std::cout << "测试用例 2:\n";
 std::cout << "输入数组: ";
 printVector(nums2);
 std::cout << std::endl;
 std::vector<std::vector<int>> result2 = threeSum(nums2);
 std::cout << "结果: ";
 printTriplets(result2);
 std::cout << std::endl;
 std::cout << "期望: ";
 printTriplets(expected2);
 std::cout << std::endl << std::endl;
```

```
// 测试用例 3
std::vector<int> nums3 = {0, 0, 0};
std::vector<std::vector<int>> expected3 = {{0, 0, 0}};
std::cout << "测试用例 3:\n";
std::cout << "输入数组: ";
printVector(nums3);
std::cout << std::endl;
std::vector<std::vector<int>> result3 = threeSum(nums3);
std::cout << "结果: ";
printTriplets(result3);
std::cout << std::endl;
std::cout << "期望: ";
printTriplets(expected3);
std::cout << std::endl << std::endl;
```

```
// 测试用例 4 - 边界情况: 多个重复元素
std::vector<int> nums4 = {-2, 0, 0, 2, 2};
std::vector<std::vector<int>> expected4 = {{-2, 0, 2}};
std::cout << "测试用例 4 (多个重复元素):\n";
std::cout << "输入数组: ";
printVector(nums4);
std::cout << std::endl;
std::vector<std::vector<int>> result4 = threeSum(nums4);
std::cout << "结果: ";
printTriplets(result4);
std::cout << std::endl;
std::cout << "期望: ";
printTriplets(expected4);
std::cout << std::endl << std::endl;
```

```
// 测试用例 5 - 边界情况: 所有元素都为负数
std::vector<int> nums5 = {-1, -2, -3, -4, -5};
std::vector<std::vector<int>> expected5 = {};
std::cout << "测试用例 5 (全负数):\n";
std::cout << "输入数组: ";
printVector(nums5);
std::cout << std::endl;
std::vector<std::vector<int>> result5 = threeSum(nums5);
std::cout << "结果: ";
printTriplets(result5);
std::cout << std::endl;
std::cout << "期望: ";
printTriplets(expected5);
```

```

 std::cout << std::endl << std::endl;
}

/***
 * 性能测试
 */
void performanceTest() {
 // 创建一个中等大小的数组进行性能测试
 const int size = 1000;
 std::vector<int> mediumArray(size);

 // 生成测试数据：包含正负数和零
 for (int i = 0; i < size; i++) {
 mediumArray[i] = (i % 100) - 50; // -50 到 49
 }

 // 测试解法二的性能
 auto start = std::chrono::high_resolution_clock::now();
 std::vector<std::vector<int>> result2 = threeSum(mediumArray);
 auto end = std::chrono::high_resolution_clock::now();
 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
 std::cout << "解法二（双指针）耗时：" << duration << "ms, 找到的三元组数量：" <<
result2.size() << std::endl;

 // 测试解法三的性能
 start = std::chrono::high_resolution_clock::now();
 std::vector<std::vector<int>> result3 = threeSumOptimized(mediumArray);
 end = std::chrono::high_resolution_clock::now();
 duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
 std::cout << "解法三（优化双指针）耗时：" << duration << "ms, 找到的三元组数量：" <<
result3.size() << std::endl;

 // 验证两种解法结果一致（这里只比较数量，不比较具体内容）
 bool resultsConsistent = (result2.size() == result3.size());
 std::cout << "所有解法结果数量一致：" << (resultsConsistent ? "true" : "false") << std::endl;
}

/***
 * 边界条件测试
 */
void boundaryTest() {
 try {
 // 测试长度为 2 的输入

```

```

 threeSum(std::vector<int>{1, 2});
 std::cout << "边界测试失败: 长度为 2 的输入没有抛出异常" << std::endl;
 } catch (const std::invalid_argument& e) {
 std::cout << "边界测试通过: 长度为 2 的输入正确抛出异常" << std::endl;
 }

 try {
 // 测试空输入
 threeSum(std::vector<int>{});
 std::cout << "边界测试失败: 空输入没有抛出异常" << std::endl;
 } catch (const std::invalid_argument& e) {
 std::cout << "边界测试通过: 空输入正确抛出异常" << std::endl;
 }
}

int main() {
 std::cout << "==== 测试用例 ===" << std::endl;
 test();

 std::cout << "==== 性能测试 ===" << std::endl;
 performanceTest();

 std::cout << "==== 边界条件测试 ===" << std::endl;
 boundaryTest();

 return 0;
}

```

=====

文件: Code28\_3Sum.java

=====

```

package class050;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.concurrent.TimeUnit;

/**
 * LeetCode 15. 三数之和 (3Sum)

```

```
*
* 题目描述:
* 给你一个整数数组 nums，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且
j != k，
* 同时还满足 nums[i] + nums[j] + nums[k] == 0。请你找出所有和为 0 且不重复的三元组。
* 注意：答案中不可以包含重复的三元组。
*
* 示例 1:
* 输入: nums = [-1, 0, 1, 2, -1, -4]
* 输出: [[-1, -1, 2], [-1, 0, 1]]
*
* 示例 2:
* 输入: nums = [0, 1, 1]
* 输出: []
*
* 示例 3:
* 输入: nums = [0, 0, 0]
* 输出: [[0, 0, 0]]
*
* 提示:
* 3 <= nums.length <= 3000
* -10^5 <= nums[i] <= 10^5
*
* 题目链接: https://leetcode.com/problems/3sum/
*
* 解题思路:
* 这道题可以使用排序 + 双指针的方法来解决:
*
* 方法一（暴力解法）:
* 使用三层嵌套循环遍历所有可能的三元组，找出和为 0 的三元组。使用 HashSet 去重。
* 时间复杂度: O(n^3)，空间复杂度: O(n)
*
* 方法二（排序 + 双指针）:
* 1. 先对数组进行排序
* 2. 遍历数组，对于每个元素 nums[i]，使用双指针 left 和 right 分别指向 i+1 和数组末尾
* 3. 计算当前三个数的和: sum = nums[i] + nums[left] + nums[right]
* 4. 如果 sum == 0，将这三个数加入结果集，并移动 left 和 right 指针
* - 为了避免重复，需要跳过相同的元素
* 5. 如果 sum < 0，说明需要增大和，移动 left 指针
* 6. 如果 sum > 0，说明需要减小和，移动 right 指针
* 7. 重复步骤 3-6，直到 left >= right
* 时间复杂度: O(n^2)，空间复杂度: O(1) 或 O(log n)（排序的空间复杂度）
*
```

\* 最优解是方法二，时间复杂度  $O(n^2)$ ，空间复杂度  $O(1)$  或  $O(\log n)$ 。

\*/

```
public class Code28_3Sum {

 /**
 * 解法一：暴力解法（不推荐，会超时）
 *
 * @param nums 输入数组
 * @return 所有和为 0 且不重复的三元组
 * @throws IllegalArgumentException 如果输入数组为 null 或长度小于 3
 */
 public static List<List<Integer>> threeSumBruteForce(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 if (nums.length < 3) {
 throw new IllegalArgumentException("输入数组长度必须至少为 3");
 }

 Set<List<Integer>> resultSet = new HashSet<>();
 int n = nums.length;

 // 三层循环遍历所有可能的三元组
 for (int i = 0; i < n; i++) {
 for (int j = i + 1; j < n; j++) {
 for (int k = j + 1; k < n; k++) {
 if (nums[i] + nums[j] + nums[k] == 0) {
 // 将三元组排序后加入结果集，利用 Set 去重
 List<Integer> triplet = Arrays.asList(
 Math.min(Math.min(nums[i], nums[j]), nums[k]),
 Math.max(Math.min(nums[i], nums[j]), Math.min(Math.max(nums[i],
 nums[j]), nums[k])),
 Math.max(Math.max(nums[i], nums[j]), nums[k])
);
 resultSet.add(triplet);
 }
 }
 }
 }

 return new ArrayList<>(resultSet);
 }
}
```

```
}

/**
 * 解法二：排序 + 双指针（最优解）
 *
 * @param nums 输入数组
 * @return 所有和为 0 且不重复的三元组
 */
public static List<List<Integer>> threeSum(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }
 if (nums.length < 3) {
 throw new IllegalArgumentException("输入数组长度必须至少为 3");
 }

 List<List<Integer>> result = new ArrayList<>();
 int n = nums.length;

 // 对数组进行排序
 Arrays.sort(nums);

 // 遍历数组，固定第一个元素
 for (int i = 0; i < n; i++) {
 // 跳过重复的第一个元素，避免产生重复的三元组
 if (i > 0 && nums[i] == nums[i - 1]) {
 continue;
 }

 // 如果当前元素已经大于 0，由于数组已排序，后面的元素都大于 0，三数之和不可能为 0
 if (nums[i] > 0) {
 break;
 }

 // 初始化双指针
 int left = i + 1;
 int right = n - 1;

 while (left < right) {
 int sum = nums[i] + nums[left] + nums[right];

 if (sum == 0) {
 result.add(Arrays.asList(nums[i], nums[left], nums[right]));
 left++;
 right--;
 } else if (sum < 0) {
 left++;
 } else {
 right--;
 }
 }
 }
}
```

```

 // 找到一个三元组
 result.add(Arrays.asList(nums[i], nums[left], nums[right]));

 // 跳过重复的左指针元素
 while (left < right && nums[left] == nums[left + 1]) {
 left++;
 }

 // 跳过重复的右指针元素
 while (left < right && nums[right] == nums[right - 1]) {
 right--;
 }

 // 移动两个指针
 left++;
 right--;
 } else if (sum < 0) {
 // 和小于 0，需要增大和，移动左指针
 left++;
 } else {
 // 和大于 0，需要减小和，移动右指针
 right--;
 }
}

}

return result;
}

/**
 * 解法三：优化的双指针实现
 *
 * @param nums 输入数组
 * @return 所有和为 0 且不重复的三元组
 */
public static List<List<Integer>> threeSumOptimized(int[] nums) {
 // 参数校验
 if (nums == null) {
 throw new IllegalArgumentException("输入数组不能为 null");
 }

 if (nums.length < 3) {
 throw new IllegalArgumentException("输入数组长度必须至少为 3");
 }
}

```

```
List<List<Integer>> result = new ArrayList<>();
int n = nums.length;

// 对数组进行排序
Arrays.sort(nums);

// 遍历数组，固定第一个元素
for (int i = 0; i < n - 2; i++) {
 // 跳过重复的第一个元素，避免产生重复的三元组
 if (i > 0 && nums[i] == nums[i - 1]) {
 continue;
 }

 // 剪枝：如果当前元素已经大于 0，三数之和不可能为 0
 if (nums[i] > 0) {
 break;
 }

 // 剪枝：如果当前元素和最大的两个元素之和仍小于 0，跳过
 if (nums[i] + nums[n - 1] + nums[n - 2] < 0) {
 continue;
 }

 // 初始化双指针
 int left = i + 1;
 int right = n - 1;

 while (left < right) {
 int sum = nums[i] + nums[left] + nums[right];

 if (sum == 0) {
 // 找到一个三元组
 result.add(Arrays.asList(nums[i], nums[left], nums[right]));

 // 跳过重复的左指针元素
 while (left < right && nums[left] == nums[left + 1]) {
 left++;
 }

 // 跳过重复的右指针元素
 while (left < right && nums[right] == nums[right - 1]) {
 right--;
 }
 }
 }
}
```

```
// 移动两个指针
 left++;
 right--;
} else if (sum < 0) {
 // 和小于 0, 需要增大和, 移动左指针
 left++;
} else {
 // 和大于 0, 需要减小和, 移动右指针
 right--;
}
}

return result;
}

/**
 * 打印三元组列表
 */
public static void printTriplets(List<List<Integer>> triplets) {
 System.out.print("[");
 for (int i = 0; i < triplets.size(); i++) {
 List<Integer> triplet = triplets.get(i);
 System.out.print("[");
 for (int j = 0; j < triplet.size(); j++) {
 System.out.print(triplet.get(j));
 if (j < triplet.size() - 1) {
 System.out.print(", ");
 }
 }
 System.out.print("]");
 if (i < triplets.size() - 1) {
 System.out.print(", ");
 }
 }
 System.out.println("]");
}

/**
 * 测试函数
 */
public static void test() {
 // 测试用例 1
}
```

```
int[] nums1 = {-1, 0, 1, 2, -1, -4};
List<List<Integer>> expected1 = Arrays.asList(
 Arrays.asList(-1, -1, 2),
 Arrays.asList(-1, 0, 1)
);
System.out.println("测试用例 1:");
System.out.print("输入数组: ");
System.out.println(Arrays.toString(nums1));
List<List<Integer>> result1 = threeSum(nums1);
System.out.print("结果: ");
printTriplets(result1);
System.out.print("期望: ");
printTriplets(expected1);
// 由于三元组的顺序可能不同, 这里不做严格的相等性验证
System.out.println();

// 测试用例 2
int[] nums2 = {0, 1, 1};
List<List<Integer>> expected2 = new ArrayList<>();
System.out.println("测试用例 2:");
System.out.print("输入数组: ");
System.out.println(Arrays.toString(nums2));
List<List<Integer>> result2 = threeSum(nums2);
System.out.print("结果: ");
printTriplets(result2);
System.out.print("期望: ");
printTriplets(expected2);
System.out.println();

// 测试用例 3
int[] nums3 = {0, 0, 0};
List<List<Integer>> expected3 = Arrays.asList(
 Arrays.asList(0, 0, 0)
);
System.out.println("测试用例 3:");
System.out.print("输入数组: ");
System.out.println(Arrays.toString(nums3));
List<List<Integer>> result3 = threeSum(nums3);
System.out.print("结果: ");
printTriplets(result3);
System.out.print("期望: ");
printTriplets(expected3);
System.out.println();
```

```

// 测试用例 4 - 边界情况: 多个重复元素
int[] nums4 = {-2, 0, 0, 2, 2};
List<List<Integer>> expected4 = Arrays.asList(
 Arrays.asList(-2, 0, 2)
);
System.out.println("测试用例 4 (多个重复元素) :");
System.out.print("输入数组: ");
System.out.println(Arrays.toString(nums4));
List<List<Integer>> result4 = threeSum(nums4);
System.out.print("结果: ");
printTriplets(result4);
System.out.print("期望: ");
printTriplets(expected4);
System.out.println();

// 测试用例 5 - 边界情况: 所有元素都为负数
int[] nums5 = {-1, -2, -3, -4, -5};
List<List<Integer>> expected5 = new ArrayList<>();
System.out.println("测试用例 5 (全负数) :");
System.out.print("输入数组: ");
System.out.println(Arrays.toString(nums5));
List<List<Integer>> result5 = threeSum(nums5);
System.out.print("结果: ");
printTriplets(result5);
System.out.print("期望: ");
printTriplets(expected5);
System.out.println();
}

/**
 * 性能测试
 */
public static void performanceTest() {
 // 创建一个中等大小的数组进行性能测试
 int size = 1000;
 int[] mediumArray = new int[size];

 // 生成测试数据: 包含正负数和零
 for (int i = 0; i < size; i++) {
 mediumArray[i] = (i % 100) - 50; // -50 到 49
 }
}

```

```

// 测试解法二的性能
long startTime = System.nanoTime();
List<List<Integer>> result2 = threeSum(mediumArray);
long endTime = System.nanoTime();
long duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法二（双指针）耗时: " + duration + "ms, 找到的三元组数量: " +
result2.size());

// 测试解法三的性能
startTime = System.nanoTime();
List<List<Integer>> result3 = threeSumOptimized(mediumArray);
endTime = System.nanoTime();
duration = TimeUnit.NANOSECONDS.toMillis(endTime - startTime);
System.out.println("解法三（优化双指针）耗时: " + duration + "ms, 找到的三元组数量: " +
result3.size());

// 验证两种解法结果一致（这里只比较数量，不比较具体内容）
boolean resultsConsistent = (result2.size() == result3.size());
System.out.println("所有解法结果数量一致: " + resultsConsistent);
}

/**
 * 边界条件测试
 */
public static void boundaryTest() {
 try {
 // 测试 null 输入
 threeSum(null);
 System.out.println("边界测试失败: null 输入没有抛出异常");
 } catch (IllegalArgumentException e) {
 System.out.println("边界测试通过: null 输入正确抛出异常");
 }

 try {
 // 测试长度为 2 的输入
 threeSum(new int[] {1, 2});
 System.out.println("边界测试失败: 长度为 2 的输入没有抛出异常");
 } catch (IllegalArgumentException e) {
 System.out.println("边界测试通过: 长度为 2 的输入正确抛出异常");
 }
}

public static void main(String[] args) {

```

```
System.out.println("==> 测试用例 ==>");
test();

System.out.println("==> 性能测试 ==>");
performanceTest();

System.out.println("==> 边界条件测试 ==>");
boundaryTest();
}
}
```

=====

文件: Code28\_3Sum.py

=====

```
from typing import List
import time

class Solution:
 """
 LeetCode 15. 三数之和 (3Sum)
 """
```

题目描述:

给你一个整数数组 `nums`，判断是否存在三元组 `[nums[i], nums[j], nums[k]]` 满足  $i \neq j, i \neq k$  且  $j \neq k$ ，

同时还满足  $nums[i] + nums[j] + nums[k] == 0$ 。请你找出所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组。

示例 1：

输入: `nums = [-1, 0, 1, 2, -1, -4]`

输出: `[[[-1, -1, 2], [-1, 0, 1]]`

示例 2：

输入: `nums = [0, 1, 1]`

输出: `[]`

示例 3：

输入: `nums = [0, 0, 0]`

输出: `[[0, 0, 0]]`

提示:

$3 \leq \text{nums.length} \leq 3000$

$-10^5 \leq \text{nums}[i] \leq 10^5$

题目链接: <https://leetcode.com/problems/3sum/>

解题思路:

这道题可以使用排序 + 双指针的方法来解决:

方法一 (暴力解法):

使用三层嵌套循环遍历所有可能的三元组，找出和为 0 的三元组。使用 set 去重。

时间复杂度:  $O(n^3)$ ， 空间复杂度:  $O(n)$

方法二 (排序 + 双指针):

1. 先对数组进行排序
2. 遍历数组，对于每个元素  $\text{nums}[i]$ ，使用双指针  $\text{left}$  和  $\text{right}$  分别指向  $i+1$  和数组末尾
3. 计算当前三个数的和:  $\text{sum} = \text{nums}[i] + \text{nums}[\text{left}] + \text{nums}[\text{right}]$
4. 如果  $\text{sum} == 0$ ，将这三个数加入结果集，并移动  $\text{left}$  和  $\text{right}$  指针
  - 为了避免重复，需要跳过相同的元素
5. 如果  $\text{sum} < 0$ ，说明需要增大和，移动  $\text{left}$  指针
6. 如果  $\text{sum} > 0$ ，说明需要减小和，移动  $\text{right}$  指针
7. 重复步骤 3-6，直到  $\text{left} \geq \text{right}$

时间复杂度:  $O(n^2)$ ， 空间复杂度:  $O(1)$  或  $O(\log n)$  (排序的空间复杂度)

最优解是方法二，时间复杂度  $O(n^2)$ ，空间复杂度  $O(1)$  或  $O(\log n)$ 。

"""

```
def three_sum_brute_force(self, nums: List[int]) -> List[List[int]]:
```

```
 """
```

解法一：暴力解法（不推荐，会超时）

Args:

`nums`: 输入数组

Returns:

`List[List[int]]`: 所有和为 0 且不重复的三元组

Raises:

`ValueError`: 如果输入数组为 `None` 或长度小于 3

时间复杂度:  $O(n^3)$  - 需要三层循环遍历所有可能的三元组

空间复杂度:  $O(n)$  - 使用 set 存储结果进行去重

"""

# 参数校验

```
if nums is None:
```

```
 raise ValueError("输入数组不能为 None")
```

```

if len(nums) < 3:
 raise ValueError("输入数组长度必须至少为 3")

result_set = set()
n = len(nums)

三层循环遍历所有可能的三元组
for i in range(n):
 for j in range(i + 1, n):
 for k in range(j + 1, n):
 if nums[i] + nums[j] + nums[k] == 0:
 # 将三元组排序后加入结果集，利用 set 去重
 triplet = tuple(sorted([nums[i], nums[j], nums[k]]))
 result_set.add(triplet)

将 set 转换为列表格式
return [list(triplet) for triplet in result_set]

```

def three\_sum(self, nums: List[int]) -> List[List[int]]:

"""

解法二：排序 + 双指针（最优解）

Args:

nums: 输入数组

Returns:

List[List[int]]: 所有和为 0 且不重复的三元组

Raises:

ValueError: 如果输入数组为 None 或长度小于 3

时间复杂度:  $O(n^2)$  – 排序  $O(n \log n)$  + 两层循环  $O(n^2)$

空间复杂度:  $O(1)$  或  $O(\log n)$  – 排序的空间复杂度

"""

# 参数校验

if nums is None:

raise ValueError("输入数组不能为 None")

if len(nums) < 3:

raise ValueError("输入数组长度必须至少为 3")

result = []

n = len(nums)

```
对数组进行排序
sorted_nums = sorted(nums)

遍历数组，固定第一个元素
for i in range(n):
 # 跳过重复的第一个元素，避免产生重复的三元组
 if i > 0 and sorted_nums[i] == sorted_nums[i - 1]:
 continue

 # 如果当前元素已经大于 0，由于数组已排序，后面的元素都大于 0，三数之和不可能为 0
 if sorted_nums[i] > 0:
 break

 # 初始化双指针
 left = i + 1
 right = n - 1

 while left < right:
 current_sum = sorted_nums[i] + sorted_nums[left] + sorted_nums[right]

 if current_sum == 0:
 # 找到一个三元组
 result.append([sorted_nums[i], sorted_nums[left], sorted_nums[right]])

 # 跳过重复的左指针元素
 while left < right and sorted_nums[left] == sorted_nums[left + 1]:
 left += 1

 # 跳过重复的右指针元素
 while left < right and sorted_nums[right] == sorted_nums[right - 1]:
 right -= 1

 # 移动两个指针
 left += 1
 right -= 1

 elif current_sum < 0:
 # 和小于 0，需要增大和，移动左指针
 left += 1
 else:
 # 和大于 0，需要减小和，移动右指针
 right -= 1

return result
```

```
def three_sum_optimized(self, nums: List[int]) -> List[List[int]]:
```

```
"""
```

解法三：优化的双指针实现

Args:

    nums: 输入数组

Returns:

    List[List[int]]: 所有和为 0 且不重复的三元组

Raises:

    ValueError: 如果输入数组为 None 或长度小于 3

时间复杂度:  $O(n^2)$  - 排序  $O(n \log n)$  + 两层循环  $O(n^2)$

空间复杂度:  $O(1)$  或  $O(\log n)$  - 排序的空间复杂度

```
"""
```

# 参数校验

```
if nums is None:
```

```
 raise ValueError("输入数组不能为 None")
```

```
if len(nums) < 3:
```

```
 raise ValueError("输入数组长度必须至少为 3")
```

```
result = []
```

```
n = len(nums)
```

# 对数组进行排序

```
sorted_nums = sorted(nums)
```

# 遍历数组，固定第一个元素

```
for i in range(n - 2):
```

# 跳过重复的第一个元素，避免产生重复的三元组

```
if i > 0 and sorted_nums[i] == sorted_nums[i - 1]:
```

```
 continue
```

# 剪枝：如果当前元素已经大于 0，三数之和不可能为 0

```
if sorted_nums[i] > 0:
```

```
 break
```

# 剪枝：如果当前元素和最大的两个元素之和仍小于 0，跳过

```
if sorted_nums[i] + sorted_nums[n - 1] + sorted_nums[n - 2] < 0:
```

```
 continue
```

# 初始化双指针

```

left = i + 1
right = n - 1

while left < right:
 current_sum = sorted_nums[i] + sorted_nums[left] + sorted_nums[right]

 if current_sum == 0:
 # 找到一个三元组
 result.append([sorted_nums[i], sorted_nums[left], sorted_nums[right]])

 # 跳过重复的左指针元素
 while left < right and sorted_nums[left] == sorted_nums[left + 1]:
 left += 1

 # 跳过重复的右指针元素
 while left < right and sorted_nums[right] == sorted_nums[right - 1]:
 right -= 1

 # 移动两个指针
 left += 1
 right -= 1

 elif current_sum < 0:
 # 和小于 0, 需要增大和, 移动左指针
 left += 1

 else:
 # 和大于 0, 需要减小和, 移动右指针
 right -= 1

return result

def test(self):
 """
 测试函数
 """

 # 测试用例 1
 nums1 = [-1, 0, 1, 2, -1, -4]
 expected1 = [[-1, -1, 2], [-1, 0, 1]]
 print("测试用例 1:")
 print(f"输入数组: {nums1}")
 result1 = self.three_sum(nums1)
 print(f"结果: {result1}")
 print(f"期望: {expected1}")
 # 由于三元组的顺序可能不同, 这里不做严格的相等性验证
 print()

```

```
测试用例 2
nums2 = [0, 1, 1]
expected2 = []
print("测试用例 2:")
print(f"输入数组: {nums2}")
result2 = self.three_sum(nums2)
print(f"结果: {result2}")
print(f"期望: {expected2}")
print()

测试用例 3
nums3 = [0, 0, 0]
expected3 = [[0, 0, 0]]
print("测试用例 3:")
print(f"输入数组: {nums3}")
result3 = self.three_sum(nums3)
print(f"结果: {result3}")
print(f"期望: {expected3}")
print()

测试用例 4 - 边界情况: 多个重复元素
nums4 = [-2, 0, 0, 2, 2]
expected4 = [[-2, 0, 2]]
print("测试用例 4 (多个重复元素) :")
print(f"输入数组: {nums4}")
result4 = self.three_sum(nums4)
print(f"结果: {result4}")
print(f"期望: {expected4}")
print()

测试用例 5 - 边界情况: 所有元素都为负数
nums5 = [-1, -2, -3, -4, -5]
expected5 = []
print("测试用例 5 (全负数) :")
print(f"输入数组: {nums5}")
result5 = self.three_sum(nums5)
print(f"结果: {result5}")
print(f"期望: {expected5}")
print()

def performance_test(self):
 """
```

## 性能测试

```
"""
```

```
创建一个中等大小的数组进行性能测试
size = 1000
medium_array = [(i % 100) - 50 for i in range(size)] # -50 到 49

测试解法二的性能
array2 = medium_array.copy()
start_time = time.time()
result2 = self.three_sum(array2)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法二（双指针）耗时: {duration:.2f}ms, 找到的三元组数量: {len(result2)}")

测试解法三的性能
array3 = medium_array.copy()
start_time = time.time()
result3 = self.three_sum_optimized(array3)
end_time = time.time()
duration = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法三（优化双指针）耗时: {duration:.2f}ms, 找到的三元组数量: {len(result3)}")

验证两种解法结果一致（这里只比较数量，不比较具体内容）
results_consistent = (len(result2) == len(result3))
print(f"所有解法结果数量一致: {results_consistent}")
```

```
def boundary_test(self):
```

```
"""
```

## 边界条件测试

```
"""
```

```
try:
```

```
 # 测试 null 输入
 self.three_sum(None)
 print("边界测试失败: None 输入没有抛出异常")
```

```
except ValueError as e:
```

```
 print(f"边界测试通过: None 输入正确抛出异常: {e}")
```

```
try:
```

```
 # 测试长度为 2 的输入
 self.three_sum([1, 2])
 print("边界测试失败: 长度为 2 的输入没有抛出异常")
```

```
except ValueError as e:
```

```
 print(f"边界测试通过: 长度为 2 的输入正确抛出异常: {e}")
```

```

主函数
if __name__ == "__main__":
 solution = Solution()

 print("==> 测试用例 ==>")
 solution.test()

 print("==> 性能测试 ==>")
 solution.performance_test()

 print("==> 边界条件测试 ==>")
 solution.boundary_test()

```

=====

文件: Code29\_LongestSubstringWithoutRepeatingCharacters.cpp

```

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

/***
 * LeetCode 3. 无重复字符的最长子串 (Longest Substring Without Repeating Characters)
 *
 * 题目描述:
 * 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。
 *
 * 示例 1:
 * 输入: s = "abcabcbb"
 * 输出: 3
 * 解释: 因为无重复字符的最长子串是 "abc"， 所以其长度为 3。
 *
 * 示例 2:
 * 输入: s = "bbbbbb"
 * 输出: 1
 * 解释: 因为无重复字符的最长子串是 "b"， 所以其长度为 1。
 */

```

\* 示例 3:

\* 输入: s = "pwwkew"

\* 输出: 3

\* 解释: 因为无重复字符的最长子串是 "wke", 所以其长度为 3。

\* 请注意, 你的答案必须是 子串 的长度, "pwke" 是一个子序列, 不是子串。

\*

\* 提示:

\*  $0 \leq s.length \leq 5 * 10^4$

\* s 由英文字母、数字、符号和空格组成

\*

\* 题目链接: <https://leetcode.cn/problems/longest-substring-without-repeating-characters/>

\*

\* 解题思路:

\* 这道题可以使用滑动窗口（双指针）的方法来解决:

\*

\* 方法一（滑动窗口 + unordered\_set）:

\* 1. 使用两个指针 left 和 right 表示当前窗口的左右边界

\* 2. 使用 unordered\_set 记录当前窗口中的字符

\* 3. 右指针向右移动, 如果当前字符不在集合中, 加入集合并更新最大长度

\* 4. 如果当前字符在集合中, 移动左指针直到移除重复字符

\*

\* 方法二（滑动窗口 + unordered\_map 优化）:

\* 1. 使用 unordered\_map 记录每个字符最后出现的位置

\* 2. 当遇到重复字符时, 可以直接将左指针移动到重复字符的下一个位置

\* 3. 避免左指针的逐步移动, 提高效率

\*

\* 时间复杂度:  $O(n)$ , 每个字符最多被访问两次

\* 空间复杂度:  $O(\min(m, n))$ , m 为字符集大小

\* 是否最优解: 是

\*/

```
class Solution {
public:
 /**
 * 解法一: 滑动窗口 + unordered_set
 *
 * @param s 输入字符串
 * @return 无重复字符的最长子串长度
 */
 static int lengthOfLongestSubstringSet(const std::string& s) {
 if (s.empty()) {
 return 0;
 }
 }
```

```

std::unordered_set<char> window;
int maxLength = 0;
int left = 0;
int n = s.length();

for (int right = 0; right < n; right++) {
 char currentChar = s[right];

 // 如果字符在集合中，移动左指针直到移除重复字符
 while (window.find(currentChar) != window.end()) {
 window.erase(s[left]);
 left++;
 }

 // 添加当前字符到集合
 window.insert(currentChar);
 // 更新最大长度
 maxLength = std::max(maxLength, right - left + 1);
}

return maxLength;
}

/***
 * 解法二：滑动窗口 + unordered_map 优化（最优解）
 *
 * @param s 输入字符串
 * @return 无重复字符的最长子串长度
 */
static int lengthOfLongestSubstring(const std::string& s) {
 if (s.empty()) {
 return 0;
 }

 std::unordered_map<char, int> charIndexMap;
 int maxLength = 0;
 int left = 0;
 int n = s.length();

 for (int right = 0; right < n; right++) {
 char currentChar = s[right];

```

```

// 如果字符已经存在，并且其位置在左指针右侧
if (charIndexMap.find(currentChar) != charIndexMap.end() &&
 charIndexMap[currentChar] >= left) {
 // 移动左指针到重复字符的下一个位置
 left = charIndexMap[currentChar] + 1;
}

// 更新字符的最新位置
charIndexMap[currentChar] = right;
// 更新最大长度
maxLength = std::max(maxLength, right - left + 1);
}

return maxLength;
}

/**
 * 解法三：数组优化版（适用于 ASCII 字符）
 *
 * @param s 输入字符串
 * @return 无重复字符的最长子串长度
 */
static int lengthOfLongestSubstringArray(const std::string& s) {
 if (s.empty()) {
 return 0;
 }

 // 假设字符集为 ASCII，使用数组记录字符最后出现的位置
 std::vector<int> lastIndex(128, -1); // ASCII 字符集大小，初始化为-1

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

 for (int right = 0; right < n; right++) {
 char currentChar = s[right];
 int charCode = static_cast<int>(currentChar);

 // 如果字符已经存在，并且其位置在左指针右侧
 if (lastIndex[charCode] >= left) {
 // 移动左指针到重复字符的下一个位置
 left = lastIndex[charCode] + 1;
 }

 // 更新字符的最新位置
 lastIndex[charCode] = right;
 // 更新最大长度
 maxLength = std::max(maxLength, right - left + 1);
 }
}

```

```
// 更新字符的最新位置
lastIndex[charCode] = right;
// 更新最大长度
maxLength = std::max(maxLength, right - left + 1);
}

return maxLength;
}
};

/***
 * 测试函数
 */
void test() {
 // 测试用例 1
 std::string s1 = "abcabcbb";
 int expected1 = 3;
 std::cout << "测试用例 1:" << std::endl;
 std::cout << "输入: \" " << s1 << " \" " << std::endl;
 std::cout << "解法一结果: " << Solution::lengthOfLongestSubstringSet(s1) << std::endl;
 std::cout << "解法二结果: " << Solution::lengthOfLongestSubstring(s1) << std::endl;
 std::cout << "解法三结果: " << Solution::lengthOfLongestSubstringArray(s1) << std::endl;
 std::cout << "期望: " << expected1 << std::endl;
 std::cout << std::endl;

 // 测试用例 2
 std::string s2 = "bbbbbb";
 int expected2 = 1;
 std::cout << "测试用例 2:" << std::endl;
 std::cout << "输入: \" " << s2 << " \" " << std::endl;
 std::cout << "解法一结果: " << Solution::lengthOfLongestSubstringSet(s2) << std::endl;
 std::cout << "解法二结果: " << Solution::lengthOfLongestSubstring(s2) << std::endl;
 std::cout << "解法三结果: " << Solution::lengthOfLongestSubstringArray(s2) << std::endl;
 std::cout << "期望: " << expected2 << std::endl;
 std::cout << std::endl;

 // 测试用例 3
 std::string s3 = "pwwkew";
 int expected3 = 3;
 std::cout << "测试用例 3:" << std::endl;
 std::cout << "输入: \" " << s3 << " \" " << std::endl;
 std::cout << "解法一结果: " << Solution::lengthOfLongestSubstringSet(s3) << std::endl;
```

```

std::cout << "解法二结果: " << Solution::lengthOfLongestSubstring(s3) << std::endl;
std::cout << "解法三结果: " << Solution::lengthOfLongestSubstringArray(s3) << std::endl;
std::cout << "期望: " << expected3 << std::endl;
std::cout << std::endl;

// 测试用例4 - 边界情况: 空字符串
std::string s4 = "";
int expected4 = 0;
std::cout << "测试用例4(空字符串):" << std::endl;
std::cout << "输入: \\" << s4 << "\"" << std::endl;
std::cout << "解法一结果: " << Solution::lengthOfLongestSubstringSet(s4) << std::endl;
std::cout << "解法二结果: " << Solution::lengthOfLongestSubstring(s4) << std::endl;
std::cout << "解法三结果: " << Solution::lengthOfLongestSubstringArray(s4) << std::endl;
std::cout << "期望: " << expected4 << std::endl;
std::cout << std::endl;

// 测试用例5 - 边界情况: 单个字符
std::string s5 = "a";
int expected5 = 1;
std::cout << "测试用例5(单个字符):" << std::endl;
std::cout << "输入: \\" << s5 << "\"" << std::endl;
std::cout << "解法一结果: " << Solution::lengthOfLongestSubstringSet(s5) << std::endl;
std::cout << "解法二结果: " << Solution::lengthOfLongestSubstring(s5) << std::endl;
std::cout << "解法三结果: " << Solution::lengthOfLongestSubstringArray(s5) << std::endl;
std::cout << "期望: " << expected5 << std::endl;
std::cout << std::endl;

// 测试用例6 - 复杂情况
std::string s6 = "dvdf";
int expected6 = 3;
std::cout << "测试用例6(复杂情况):" << std::endl;
std::cout << "输入: \\" << s6 << "\"" << std::endl;
std::cout << "解法一结果: " << Solution::lengthOfLongestSubstringSet(s6) << std::endl;
std::cout << "解法二结果: " << Solution::lengthOfLongestSubstring(s6) << std::endl;
std::cout << "解法三结果: " << Solution::lengthOfLongestSubstringArray(s6) << std::endl;
std::cout << "期望: " << expected6 << std::endl;
std::cout << std::endl;
}

/**
 * 性能测试函数
 */
void performanceTest() {

```

```

// 创建长字符串进行性能测试
std::string longString;
for (int i = 0; i < 100000; i++) {
 longString += static_cast<char>('a' + (i % 26)); // 循环添加 a-z
}

// 测试解法一的性能
auto start = std::chrono::high_resolution_clock::now();
int result1 = Solution::lengthOfLongestSubstringSet(longString);
auto end = std::chrono::high_resolution_clock::now();
auto duration1 = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
std::cout << "解法一 (unordered_set) 耗时: " << duration1 << "ms, 结果: " << result1 << std::endl;

// 测试解法二的性能
start = std::chrono::high_resolution_clock::now();
int result2 = Solution::lengthOfLongestSubstring(longString);
end = std::chrono::high_resolution_clock::now();
auto duration2 = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
std::cout << "解法二 (unordered_map 优化) 耗时: " << duration2 << "ms, 结果: " << result2 << std::endl;

// 测试解法三的性能
start = std::chrono::high_resolution_clock::now();
int result3 = Solution::lengthOfLongestSubstringArray(longString);
end = std::chrono::high_resolution_clock::now();
auto duration3 = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
std::cout << "解法三 (数组优化) 耗时: " << duration3 << "ms, 结果: " << result3 << std::endl;

// 验证结果一致性
std::cout << "所有解法结果一致: " << (result1 == result2 && result2 == result3 ? "true" :
"false") << std::endl;
}

/**
 * 算法分析函数
 */
void algorithmAnalysis() {
 std::cout << "==== 算法分析 ===" << std::endl;
 std::cout << "1. 解法一 (滑动窗口 + unordered_set)" << std::endl;
 std::cout << " - 时间复杂度: O(n) - 每个字符最多被访问两次" << std::endl;
 std::cout << " - 空间复杂度: O(min(m, n)) - m 为字符集大小" << std::endl;
 std::cout << " - 优点: 实现简单, 易于理解" << std::endl;
}

```

```

std::cout << " - 缺点：最坏情况下需要逐步移动左指针" << std::endl;
std::cout << std::endl;

std::cout << "2. 解法二（滑动窗口 + unordered_map 优化）" << std::endl;
std::cout << " - 时间复杂度: O(n) - 每个字符只被访问一次" << std::endl;
std::cout << " - 空间复杂度: O(min(m, n)) - m 为字符集大小" << std::endl;
std::cout << " - 优点: 效率最高, 直接跳转到重复字符位置" << std::endl;
std::cout << " - 缺点: 需要额外的 unordered_map 空间" << std::endl;
std::cout << std::endl;

std::cout << "3. 解法三（数组优化版）" << std::endl;
std::cout << " - 时间复杂度: O(n) - 每个字符只被访问一次" << std::endl;
std::cout << " - 空间复杂度: O(1) - 固定大小的数组" << std::endl;
std::cout << " - 优点: 空间效率最高, 适用于 ASCII 字符集" << std::endl;
std::cout << " - 缺点: 仅适用于有限字符集" << std::endl;
std::cout << std::endl;

std::cout << "推荐使用解法二作为通用解决方案" << std::endl;
}

int main() {
 std::cout << "==== 无重复字符的最长子串 算法实现 ===" << std::endl;
 std::cout << std::endl;

 std::cout << "==== 测试用例 ===" << std::endl;
 test();

 std::cout << "==== 性能测试 ===" << std::endl;
 performanceTest();

 std::cout << "==== 算法分析 ===" << std::endl;
 algorithmAnalysis();

 return 0;
}
=====

文件: Code29_LongestSubstringWithoutRepeatingCharacters.java
=====

package class050;

import java.util.HashMap;

```

文件: Code29\_LongestSubstringWithoutRepeatingCharacters.java

```

package class050;

import java.util.HashMap;

```

```
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

/**
 * LeetCode 3. 无重复字符的最长子串 (Longest Substring Without Repeating Characters)
 *
 * 题目描述:
 * 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。
 *
 * 示例 1:
 * 输入: s = "abcabcbb"
 * 输出: 3
 * 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。
 *
 * 示例 2:
 * 输入: s = "bbbbbb"
 * 输出: 1
 * 解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。
 *
 * 示例 3:
 * 输入: s = "pwwkew"
 * 输出: 3
 * 解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
 * 请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
 *
 * 提示:
 * 0 <= s.length <= 5 * 10^4
 * s 由英文字母、数字、符号和空格组成
 *
 * 题目链接: https://leetcode.cn/problems/longest-substring-without-repeating-characters/
 *
 * 解题思路:
 * 这道题可以使用滑动窗口（双指针）的方法来解决:
 *
 * 方法一（滑动窗口 + HashSet）:
 * 1. 使用两个指针 left 和 right 表示当前窗口的左右边界
 * 2. 使用 HashSet 记录当前窗口中的字符
 * 3. 右指针向右移动，如果当前字符不在集合中，加入集合并更新最大长度
 * 4. 如果当前字符在集合中，移动左指针直到移除重复字符
 *
 * 方法二（滑动窗口 + HashMap 优化）:
 * 1. 使用 HashMap 记录每个字符最后出现的位置
```

```
* 2. 当遇到重复字符时，可以直接将左指针移动到重复字符的下一个位置
* 3. 避免左指针的逐步移动，提高效率
*
* 时间复杂度: O(n)，每个字符最多被访问两次
* 空间复杂度: O(min(m, n))，m 为字符集大小
* 是否最优解: 是
*/
```

```
public class Code29_LongestSubstringWithoutRepeatingCharacters {

 /**
 * 解法一：滑动窗口 + HashSet
 *
 * @param s 输入字符串
 * @return 无重复字符的最长子串长度
 */
 public static int lengthOfLongestSubstringHashSet(String s) {
 if (s == null || s.length() == 0) {
 return 0;
 }

 Set<Character> window = new HashSet<>();
 int maxLength = 0;
 int left = 0, right = 0;
 int n = s.length();

 while (right < n) {
 char currentChar = s.charAt(right);

 // 如果窗口中没有当前字符，加入窗口
 if (!window.contains(currentChar)) {
 window.add(currentChar);
 maxLength = Math.max(maxLength, right - left + 1);
 right++;
 } else {
 // 有重复字符，移动左指针直到移除重复字符
 window.remove(s.charAt(left));
 left++;
 }
 }

 return maxLength;
 }
}
```

```
/**
 * 解法二：滑动窗口 + HashMap 优化（最优解）
 *
 * @param s 输入字符串
 * @return 无重复字符的最长子串长度
 */

public static int lengthOfLongestSubstring(String s) {
 if (s == null || s.length() == 0) {
 return 0;
 }

 Map<Character, Integer> charIndexMap = new HashMap<>();
 int maxLength = 0;
 int left = 0;

 for (int right = 0; right < s.length(); right++) {
 char currentChar = s.charAt(right);

 // 如果字符已经存在，更新左指针位置
 if (charIndexMap.containsKey(currentChar)) {
 // 取最大值是为了避免左指针回退
 left = Math.max(left, charIndexMap.get(currentChar) + 1);
 }

 // 更新字符的最新位置
 charIndexMap.put(currentChar, right);
 maxLength = Math.max(maxLength, right - left + 1);
 }

 return maxLength;
}

/**
 * 解法三：数组优化版（适用于 ASCII 字符）
 *
 * @param s 输入字符串
 * @return 无重复字符的最长子串长度
 */

public static int lengthOfLongestSubstringArray(String s) {
 if (s == null || s.length() == 0) {
 return 0;
 }
```

```
// 假设字符集为 ASCII， 使用数组记录字符最后出现的位置
int[] lastIndex = new int[128]; // ASCII 字符集大小
// 初始化数组为-1
for (int i = 0; i < lastIndex.length; i++) {
 lastIndex[i] = -1;
}

int maxLength = 0;
int left = 0;

for (int right = 0; right < s.length(); right++) {
 char currentChar = s.charAt(right);

 // 如果字符已经存在， 更新左指针位置
 if (lastIndex[currentChar] >= left) {
 left = lastIndex[currentChar] + 1;
 }

 // 更新字符的最新位置
 lastIndex[currentChar] = right;
 maxLength = Math.max(maxLength, right - left + 1);
}

return maxLength;
}

/**
 * 测试函数
 */
public static void test() {
 // 测试用例 1
 String s1 = "abcabcbb";
 int expected1 = 3;
 System.out.println("测试用例 1:");
 System.out.println("输入: " + s1);
 System.out.println("解法一结果: " + lengthOfLongestSubstringHashSet(s1));
 System.out.println("解法二结果: " + lengthOfLongestSubstring(s1));
 System.out.println("解法三结果: " + lengthOfLongestSubstringArray(s1));
 System.out.println("期望: " + expected1);
 System.out.println();

 // 测试用例 2
}
```

```
String s2 = "bbbbbb";
int expected2 = 1;
System.out.println("测试用例 2:");
System.out.println("输入: " + s2 + ")");
System.out.println("解法一结果: " + lengthOfLongestSubstringHashSet(s2));
System.out.println("解法二结果: " + lengthOfLongestSubstring(s2));
System.out.println("解法三结果: " + lengthOfLongestSubstringArray(s2));
System.out.println("期望: " + expected2);
System.out.println();
```

// 测试用例 3

```
String s3 = "pwwkew";
int expected3 = 3;
System.out.println("测试用例 3:");
System.out.println("输入: " + s3 + ")");
System.out.println("解法一结果: " + lengthOfLongestSubstringHashSet(s3));
System.out.println("解法二结果: " + lengthOfLongestSubstring(s3));
System.out.println("解法三结果: " + lengthOfLongestSubstringArray(s3));
System.out.println("期望: " + expected3);
System.out.println();
```

// 测试用例 4 - 边界情况: 空字符串

```
String s4 = "";
int expected4 = 0;
System.out.println("测试用例 4 (空字符串) :");
System.out.println("输入: " + s4 + ")");
System.out.println("解法一结果: " + lengthOfLongestSubstringHashSet(s4));
System.out.println("解法二结果: " + lengthOfLongestSubstring(s4));
System.out.println("解法三结果: " + lengthOfLongestSubstringArray(s4));
System.out.println("期望: " + expected4);
System.out.println();
```

// 测试用例 5 - 边界情况: 单个字符

```
String s5 = "a";
int expected5 = 1;
System.out.println("测试用例 5 (单个字符) :");
System.out.println("输入: " + s5 + ")");
System.out.println("解法一结果: " + lengthOfLongestSubstringHashSet(s5));
System.out.println("解法二结果: " + lengthOfLongestSubstring(s5));
System.out.println("解法三结果: " + lengthOfLongestSubstringArray(s5));
System.out.println("期望: " + expected5);
System.out.println();
```

```
// 测试用例 6 - 复杂情况
String s6 = "dvdf";
int expected6 = 3;
System.out.println("测试用例 6 (复杂情况) :");
System.out.println("输入: " + s6 + ")");
System.out.println("解法一结果: " + lengthOfLongestSubstringHashSet(s6));
System.out.println("解法二结果: " + lengthOfLongestSubstring(s6));
System.out.println("解法三结果: " + lengthOfLongestSubstringArray(s6));
System.out.println("期望: " + expected6);
System.out.println();
}

/**
 * 性能测试
 */
public static void performanceTest() {
 // 创建长字符串进行性能测试
 StringBuilder sb = new StringBuilder();
 for (int i = 0; i < 100000; i++) {
 sb.append((char) ('a' + (i % 26))); // 循环添加 a-z
 }
 String longString = sb.toString();

 // 测试解法一的性能
 long startTime = System.nanoTime();
 int result1 = lengthOfLongestSubstringHashSet(longString);
 long endTime = System.nanoTime();
 long duration1 = (endTime - startTime) / 1000000; // 转换为毫秒
 System.out.println("解法一 (HashSet) 耗时: " + duration1 + "ms, 结果: " + result1);

 // 测试解法二的性能
 startTime = System.nanoTime();
 int result2 = lengthOfLongestSubstring(longString);
 endTime = System.nanoTime();
 long duration2 = (endTime - startTime) / 1000000;
 System.out.println("解法二 (HashMap 优化) 耗时: " + duration2 + "ms, 结果: " + result2);

 // 测试解法三的性能
 startTime = System.nanoTime();
 int result3 = lengthOfLongestSubstringArray(longString);
 endTime = System.nanoTime();
 long duration3 = (endTime - startTime) / 1000000;
 System.out.println("解法三 (数组优化) 耗时: " + duration3 + "ms, 结果: " + result3);
}
```

```
// 验证结果一致性
System.out.println("所有解法结果一致: " + (result1 == result2 && result2 == result3));
}

public static void main(String[] args) {
 System.out.println("== 无重复字符的最长子串 算法实现 ==");
 System.out.println();

 System.out.println("== 测试用例 ==");
 test();

 System.out.println("== 性能测试 ==");
 performanceTest();
}
=====
```

文件: Code29\_LongestSubstringWithoutRepeatingCharacters.py

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

```
"""
LeetCode 3. 无重复字符的最长子串 (Longest Substring Without Repeating Characters)
```

题目描述:

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。

示例 1:

输入: s = "abcabcbb"

输出: 3

解释: 因为无重复字符的最长子串是 "abc"， 所以其长度为 3。

示例 2:

输入: s = "bbbbbb"

输出: 1

解释: 因为无重复字符的最长子串是 "b"， 所以其长度为 1。

示例 3:

输入: s = "pwwkew"

输出: 3

解释：因为无重复字符的最长子串是 “wke”， 所以其长度为 3。

请注意，你的答案必须是 子串 的长度，”pwke” 是一个子序列，不是子串。

提示：

$0 \leq s.length \leq 5 * 10^4$

s 由英文字母、数字、符号和空格组成

题目链接：<https://leetcode.cn/problems/longest-substring-without-repeating-characters/>

解题思路：

这道题可以使用滑动窗口（双指针）的方法来解决：

方法一（滑动窗口 + 集合）：

1. 使用两个指针 left 和 right 表示当前窗口的左右边界
2. 使用集合记录当前窗口中的字符
3. 右指针向右移动，如果当前字符不在集合中，加入集合并更新最大长度
4. 如果当前字符在集合中，移动左指针直到移除重复字符

方法二（滑动窗口 + 字典优化）：

1. 使用字典记录每个字符最后出现的位置
2. 当遇到重复字符时，可以直接将左指针移动到重复字符的下一个位置
3. 避免左指针的逐步移动，提高效率

时间复杂度： $O(n)$ ， 每个字符最多被访问两次

空间复杂度： $O(\min(m, n))$ ， m 为字符集大小

是否最优解：是

"""

class Solution:

"""

无重复字符的最长子串解决方案类

"""

@staticmethod

def length\_of\_longest\_substring\_set(s: str) -> int:

"""

解法一：滑动窗口 + 集合

Args:

s: 输入字符串

Returns:

int: 无重复字符的最长子串长度

```

时间复杂度: O(n) - 每个字符最多被访问两次
空间复杂度: O(min(m, n)) - m 为字符集大小
"""

if not s:
 return 0

char_set = set()
max_length = 0
left = 0
n = len(s)

for right in range(n):
 current_char = s[right]

 # 如果字符在集合中, 移动左指针直到移除重复字符
 while current_char in char_set:
 char_set.remove(s[left])
 left += 1

 # 添加当前字符到集合
 char_set.add(current_char)
 # 更新最大长度
 max_length = max(max_length, right - left + 1)

return max_length

```

```

@staticmethod
def length_of_longest_substring_dict(s: str) -> int:
"""

```

解法二：滑动窗口 + 字典优化（最优解）

Args:

s: 输入字符串

Returns:

int: 无重复字符的最长子串长度

```

时间复杂度: O(n) - 每个字符只被访问一次
空间复杂度: O(min(m, n)) - m 为字符集大小
"""

if not s:
 return 0

```

```
char_dict = {} # 存储字符最后出现的位置
max_length = 0
left = 0

for right, char in enumerate(s):
 # 如果字符已经存在，并且其位置在左指针右侧
 if char in char_dict and char_dict[char] >= left:
 # 移动左指针到重复字符的下一个位置
 left = char_dict[char] + 1

 # 更新字符的最新位置
 char_dict[char] = right
 # 更新最大长度
 max_length = max(max_length, right - left + 1)

return max_length
```

```
@staticmethod
def length_of_longest_substring_array(s: str) -> int:
 """
```

解法三：数组优化版（适用于 ASCII 字符）

Args:  
s: 输入字符串

Returns:  
int: 无重复字符的最长子串长度

时间复杂度:  $O(n)$  - 每个字符只被访问一次

空间复杂度:  $O(1)$  - 固定大小的数组

```
"""
if not s:
 return 0
```

```
假设字符集为 ASCII，使用数组记录字符最后出现的位置
last_index = [-1] * 128 # ASCII 字符集大小
```

```
max_length = 0
left = 0

for right, char in enumerate(s):
 char_code = ord(char)
```

```
如果字符已经存在，并且其位置在左指针右侧
if last_index[char_code] >= left:
 # 移动左指针到重复字符的下一个位置
 left = last_index[char_code] + 1

 # 更新字符的最新位置
 last_index[char_code] = right
 # 更新最大长度
 max_length = max(max_length, right - left + 1)

return max_length

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

 # 测试用例 1
 s1 = "abcabcbb"
 expected1 = 3
 print("测试用例 1:")
 print(f"输入: \'{s1}\''")
 print(f"解法一结果: {solution.length_of_longest_substring_set(s1)}")
 print(f"解法二结果: {solution.length_of_longest_substring_dict(s1)}")
 print(f"解法三结果: {solution.length_of_longest_substring_array(s1)}")
 print(f"期望: {expected1}")
 print()

 # 测试用例 2
 s2 = "bbbbbb"
 expected2 = 1
 print("测试用例 2:")
 print(f"输入: \'{s2}\''")
 print(f"解法一结果: {solution.length_of_longest_substring_set(s2)}")
 print(f"解法二结果: {solution.length_of_longest_substring_dict(s2)}")
 print(f"解法三结果: {solution.length_of_longest_substring_array(s2)}")
 print(f"期望: {expected2}")
 print()

 # 测试用例 3
 s3 = "pwwkew"
```

```
expected3 = 3
print("测试用例 3:")
print(f"输入: \'{s3}\'")
print(f"解法一结果: {solution.length_of_longest_substring_set(s3)}")
print(f"解法二结果: {solution.length_of_longest_substring_dict(s3)}")
print(f"解法三结果: {solution.length_of_longest_substring_array(s3)}")
print(f"期望: {expected3}")
print()

测试用例 4 - 边界情况: 空字符串
s4 = ""
expected4 = 0
print("测试用例 4 (空字符串) :")
print(f"输入: \'{s4}\'")
print(f"解法一结果: {solution.length_of_longest_substring_set(s4)}")
print(f"解法二结果: {solution.length_of_longest_substring_dict(s4)}")
print(f"解法三结果: {solution.length_of_longest_substring_array(s4)}")
print(f"期望: {expected4}")
print()

测试用例 5 - 边界情况: 单个字符
s5 = "a"
expected5 = 1
print("测试用例 5 (单个字符) :")
print(f"输入: \'{s5}\'")
print(f"解法一结果: {solution.length_of_longest_substring_set(s5)}")
print(f"解法二结果: {solution.length_of_longest_substring_dict(s5)}")
print(f"解法三结果: {solution.length_of_longest_substring_array(s5)}")
print(f"期望: {expected5}")
print()

测试用例 6 - 复杂情况
s6 = "dvdf"
expected6 = 3
print("测试用例 6 (复杂情况) :")
print(f"输入: \'{s6}\'")
print(f"解法一结果: {solution.length_of_longest_substring_set(s6)}")
print(f"解法二结果: {solution.length_of_longest_substring_dict(s6)}")
print(f"解法三结果: {solution.length_of_longest_substring_array(s6)}")
print(f"期望: {expected6}")
print()

def performance_test():
```

```
"""
性能测试函数
"""

import time
solution = Solution()

创建长字符串进行性能测试
long_string = ''.join(chr(ord('a') + i % 26) for i in range(100000))

测试解法一的性能
start_time = time.time()
result1 = solution.length_of_longest_substring_set(long_string)
end_time = time.time()
duration1 = (end_time - start_time) * 1000 # 转换为毫秒
print(f"解法一（集合）耗时: {duration1:.2f}ms, 结果: {result1}")

测试解法二的性能
start_time = time.time()
result2 = solution.length_of_longest_substring_dict(long_string)
end_time = time.time()
duration2 = (end_time - start_time) * 1000
print(f"解法二（字典优化）耗时: {duration2:.2f}ms, 结果: {result2}")

测试解法三的性能
start_time = time.time()
result3 = solution.length_of_longest_substring_array(long_string)
end_time = time.time()
duration3 = (end_time - start_time) * 1000
print(f"解法三（数组优化）耗时: {duration3:.2f}ms, 结果: {result3}")

验证结果一致性
print(f"所有解法结果一致: {result1 == result2 == result3}")

def algorithm_analysis():
"""
算法分析函数
"""

 print("== 算法分析 ==")
 print("1. 解法一（滑动窗口 + 集合）")
 print(" - 时间复杂度: O(n) - 每个字符最多被访问两次")
 print(" - 空间复杂度: O(min(m, n)) - m 为字符集大小")
 print(" - 优点: 实现简单, 易于理解")
 print(" - 缺点: 最坏情况下需要逐步移动左指针")
```

```

print()

print("2. 解法二（滑动窗口 + 字典优化）")
print(" - 时间复杂度: O(n) - 每个字符只被访问一次")
print(" - 空间复杂度: O(min(m, n)) - m 为字符集大小")
print(" - 优点: 效率最高, 直接跳转到重复字符位置")
print(" - 缺点: 需要额外的字典空间")
print()

print("3. 解法三（数组优化版）")
print(" - 时间复杂度: O(n) - 每个字符只被访问一次")
print(" - 空间复杂度: O(1) - 固定大小的数组")
print(" - 优点: 空间效率最高, 适用于 ASCII 字符集")
print(" - 缺点: 仅适用于有限字符集")
print()

print("推荐使用解法二作为通用解决方案")

if __name__ == "__main__":
 print("== 无重复字符的最长子串 算法实现 ==")
 print()

 print("== 测试用例 ==")
 test()

 print("== 性能测试 ==")
 performance_test()

 print("== 算法分析 ==")
 algorithm_analysis()

```

=====

文件: Code30\_MinimumWindowSubstring.cpp

=====

```

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

```

```
/**
 * LeetCode 76. 最小覆盖子串 (Minimum Window Substring)
 *
 * 题目描述:
 * 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。
 * - 如果 s 中存在这样的子串，我们保证它是唯一的答案。
 *
 * 示例 1:
 * 输入: s = "ADOBECODEBANC", t = "ABC"
 * 输出: "BANC"
 * 解释: 最小覆盖子串 "BANC" 包含 'A', 'B', 'C' 各一个。
 *
 * 示例 2:
 * 输入: s = "a", t = "a"
 * 输出: "a"
 *
 * 示例 3:
 * 输入: s = "a", t = "aa"
 * 输出: ""
 * 解释: t 中有两个 'a'，但 s 中只有一个 'a'，所以返回空字符串。
 *
 * 提示:
 * - $1 \leq s.length, t.length \leq 10^5$
 * - s 和 t 由英文字母组成
 *
 * 题目链接: https://leetcode.cn/problems/minimum-window-substring/
 *
 * 解题思路:
 * 这道题可以使用滑动窗口（双指针）的方法来解决:
 *
 * 方法一（滑动窗口 + unordered_map）:
 * 1. 使用两个指针 left 和 right 表示当前窗口的左右边界
 * 2. 使用 unordered_map 记录 t 中每个字符的出现次数
 * 3. 使用另一个 unordered_map 记录当前窗口中包含 t 字符的情况
 * 4. 移动右指针扩展窗口，直到窗口包含 t 的所有字符
 * 5. 然后移动左指针收缩窗口，找到最小覆盖子串
 *
 * 时间复杂度: $O(n + m)$, n 为 s 长度, m 为 t 长度
```

\* 空间复杂度: O(m), 存储 t 的字符频率

\* 是否最优解: 是

\*/

```
class Solution {
public:
 /**
 * 解法一: 滑动窗口 + unordered_map (最优解)
 *
 * @param s 源字符串
 * @param t 目标字符串
 * @return 最小覆盖子串
 */
 static string minWindow(const string& s, const string& t) {
 if (s.empty() || t.empty() || s.length() < t.length()) {
 return "";
 }

 // 记录 t 中每个字符的出现次数
 unordered_map<char, int> targetMap;
 for (char c : t) {
 targetMap[c]++;
 }

 // 记录当前窗口中字符的出现次数
 unordered_map<char, int> windowMap;

 int left = 0, right = 0;
 int minLen = INT_MAX;
 int minStart = 0;
 int required = targetMap.size(); // 需要匹配的字符种类数
 int formed = 0; // 当前窗口中已匹配的字符种类数

 while (right < s.length()) {
 char rightChar = s[right];
 windowMap[rightChar]++;
 if (targetMap.find(rightChar) != targetMap.end() &&
 windowMap[rightChar] == targetMap[rightChar]) {
 formed++;
 }

 // 如果当前字符在 t 中, 且窗口中出现次数等于 t 中出现次数
 if (targetMap.find(rightChar) != targetMap.end() &&
 windowMap[rightChar] == targetMap[rightChar]) {
 formed++;
 }
 }
 }
}
```

```

// 当窗口包含 t 的所有字符时，尝试收缩窗口
while (left <= right && formed == required) {
 char leftChar = s[left];

 // 更新最小覆盖子串
 if (right - left + 1 < minLen) {
 minLen = right - left + 1;
 minStart = left;
 }

 // 移动左指针
 windowMap[leftChar]--;
 if (targetMap.find(leftChar) != targetMap.end() &&
 windowMap[leftChar] < targetMap[leftChar]) {
 formed--;
 }
 left++;
}

right++;
}

```

```

return minLen == INT_MAX ? "" : s.substr(minStart, minLen);
}

```

```

/**
 * 解法二：优化版滑动窗口（使用数组替代 unordered_map）
 *
 * @param s 源字符串
 * @param t 目标字符串
 * @return 最小覆盖子串
 */

```

```

static string minWindowOptimized(const string& s, const string& t) {
 if (s.empty() || t.empty() || s.length() < t.length()) {
 return "";
 }

```

```

// 使用数组记录字符频率（假设字符为 ASCII）
vector<int> targetFreq(128, 0);
vector<int> windowFreq(128, 0);

```

```

for (char c : t) {
 targetFreq[c]++;
}

```

```
}

int left = 0, right = 0;
int minLen = INT_MAX;
int minStart = 0;
int required = 0;

// 计算需要匹配的字符种类数
for (int freq : targetFreq) {
 if (freq > 0) {
 required++;
 }
}

int formed = 0;

while (right < s.length()) {
 char rightChar = s[right];
 windowFreq[rightChar]++;
}

// 如果当前字符在 t 中，且窗口中出现次数等于 t 中出现次数
if (targetFreq[rightChar] > 0 &&
 windowFreq[rightChar] == targetFreq[rightChar]) {
 formed++;
}

// 当窗口包含 t 的所有字符时，尝试收缩窗口
while (left <= right && formed == required) {
 char leftChar = s[left];

 // 更新最小覆盖子串
 if (right - left + 1 < minLen) {
 minLen = right - left + 1;
 minStart = left;
 }

 // 移动左指针
 windowFreq[leftChar]--;
 if (targetFreq[leftChar] > 0 &&
 windowFreq[leftChar] < targetFreq[leftChar]) {
 formed--;
 }
 left++;
}
```

```

 }

 right++;
}

return minLen == INT_MAX ? "" : s.substr(minStart, minLen);
}

/***
 * 解法三：进一步优化的滑动窗口（跳过无关字符）
 *
 * @param s 源字符串
 * @param t 目标字符串
 * @return 最小覆盖子串
*/
static string minWindowAdvanced(const string& s, const string& t) {
 if (s.empty() || t.empty() || s.length() < t.length()) {
 return "";
 }

 // 使用数组记录字符频率
 vector<int> targetFreq(128, 0);
 for (char c : t) {
 targetFreq[c]++;
 }

 // 预处理：只保留 s 中在 t 中出现的字符及其位置
 int count = t.length();
 int left = 0, right = 0;
 int minLen = INT_MAX;
 int minStart = 0;

 while (right < s.length()) {
 // 如果当前字符在 t 中，减少计数
 if (targetFreq[s[right]] > 0) {
 count--;
 }
 targetFreq[s[right]]--;
 right++;

 // 当计数为 0 时，表示窗口包含 t 的所有字符
 while (count == 0) {
 // 更新最小覆盖子串

```

```

 if (right - left < minLen) {
 minLen = right - left;
 minStart = left;
 }

 // 移动左指针
 targetFreq[s[left]]++;
 if (targetFreq[s[left]] > 0) {
 count++;
 }
 left++;
 }

}

return minLen == INT_MAX ? "" : s.substr(minStart, minLen);
}
};

/***
 * 测试函数
 */
void test() {
 // 测试用例 1
 string s1 = "ADOBECODEBANC";
 string t1 = "ABC";
 string expected1 = "BANC";
 cout << "测试用例 1:" << endl;
 cout << "s = " << s1 << "\\", t = " << t1 << "\\" " << endl;
 cout << "解法一结果: " << Solution::minWindow(s1, t1) << "\\" " << endl;
 cout << "解法二结果: " << Solution::minWindowOptimized(s1, t1) << "\\" " << endl;
 cout << "解法三结果: " << Solution::minWindowAdvanced(s1, t1) << "\\" " << endl;
 cout << "期望: " << expected1 << "\\" " << endl;
 cout << endl;

 // 测试用例 2
 string s2 = "a";
 string t2 = "a";
 string expected2 = "a";
 cout << "测试用例 2:" << endl;
 cout << "s = " << s2 << "\\", t = " << t2 << "\\" " << endl;
 cout << "解法一结果: " << Solution::minWindow(s2, t2) << "\\" " << endl;
 cout << "解法二结果: " << Solution::minWindowOptimized(s2, t2) << "\\" " << endl;
 cout << "解法三结果: " << Solution::minWindowAdvanced(s2, t2) << "\\" " << endl;
}

```

```

cout << "期望: \\" << expected2 << "\\" << endl;
cout << endl;

// 测试用例 3
string s3 = "a";
string t3 = "aa";
string expected3 = "";
cout << "测试用例 3:" << endl;
cout << "s = \\" << s3 << "\", t = \\" << t3 << "\\" << endl;
cout << "解法一结果: \\" << Solution::minWindow(s3, t3) << "\\" << endl;
cout << "解法二结果: \\" << Solution::minWindowOptimized(s3, t3) << "\\" << endl;
cout << "解法三结果: \\" << Solution::minWindowAdvanced(s3, t3) << "\\" << endl;
cout << "期望: \\" << expected3 << "\\" << endl;
cout << endl;

// 测试用例 4 - 边界情况: s 和 t 相同
string s4 = "abc";
string t4 = "abc";
string expected4 = "abc";
cout << "测试用例 4 (s 和 t 相同) :" << endl;
cout << "s = \\" << s4 << "\", t = \\" << t4 << "\\" << endl;
cout << "解法一结果: \\" << Solution::minWindow(s4, t4) << "\\" << endl;
cout << "解法二结果: \\" << Solution::minWindowOptimized(s4, t4) << "\\" << endl;
cout << "解法三结果: \\" << Solution::minWindowAdvanced(s4, t4) << "\\" << endl;
cout << "期望: \\" << expected4 << "\\" << endl;
cout << endl;

// 测试用例 5 - 边界情况: t 不在 s 中
string s5 = "abcdef";
string t5 = "xyz";
string expected5 = "";
cout << "测试用例 5 (t 不在 s 中) :" << endl;
cout << "s = \\" << s5 << "\", t = \\" << t5 << "\\" << endl;
cout << "解法一结果: \\" << Solution::minWindow(s5, t5) << "\\" << endl;
cout << "解法二结果: \\" << Solution::minWindowOptimized(s5, t5) << "\\" << endl;
cout << "解法三结果: \\" << Solution::minWindowAdvanced(s5, t5) << "\\" << endl;
cout << "期望: \\" << expected5 << "\\" << endl;
cout << endl;

}

/***
 * 性能测试函数
*/

```

```
void performanceTest() {
 // 创建长字符串进行性能测试
 string longS;
 for (int i = 0; i < 10000; i++) {
 longS += "ABCDEFG";
 }
 string longT = "ABC";

 // 测试解法一的性能
 auto start = chrono::high_resolution_clock::now();
 string result1 = Solution::minWindow(longS, longT);
 auto end = chrono::high_resolution_clock::now();
 auto duration1 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法一 (unordered_map) 耗时: " << duration1 << "ms, 结果长度: " << result1.length()
 << endl;

 // 测试解法二的性能
 start = chrono::high_resolution_clock::now();
 string result2 = Solution::minWindowOptimized(longS, longT);
 end = chrono::high_resolution_clock::now();
 auto duration2 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法二 (数组优化) 耗时: " << duration2 << "ms, 结果长度: " << result2.length() <<
 endl;

 // 测试解法三的性能
 start = chrono::high_resolution_clock::now();
 string result3 = Solution::minWindowAdvanced(longS, longT);
 end = chrono::high_resolution_clock::now();
 auto duration3 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法三 (高级优化) 耗时: " << duration3 << "ms, 结果长度: " << result3.length() <<
 endl;

 // 验证结果一致性
 cout << "所有解法结果一致: " << (result1 == result2 && result2 == result3 ? "true" : "false")
 << endl;
}

/**
 * 算法分析函数
 */
void algorithmAnalysis() {
 cout << "==== 算法分析 ===" << endl;
 cout << "1. 解法一 (滑动窗口 + unordered_map)" << endl;
```

```

cout << " - 时间复杂度: O(n + m) - n 为 s 长度, m 为 t 长度" << endl;
cout << " - 空间复杂度: O(m) - 存储 t 的字符频率" << endl;
cout << " - 优点: 通用性强, 适用于任意字符集" << endl;
cout << " - 缺点: unordered_map 操作有一定开销" << endl;
cout << endl;

cout << "2. 解法二 (数组优化版)" << endl;
cout << " - 时间复杂度: O(n + m)" << endl;
cout << " - 空间复杂度: O(1) - 固定大小的数组" << endl;
cout << " - 优点: 效率高, 适用于 ASCII 字符集" << endl;
cout << " - 缺点: 仅适用于有限字符集" << endl;
cout << endl;

cout << "3. 解法三 (进一步优化)" << endl;
cout << " - 时间复杂度: O(n)" << endl;
cout << " - 空间复杂度: O(1)" << endl;
cout << " - 优点: 最优化实现, 跳过无关字符" << endl;
cout << " - 缺点: 实现相对复杂" << endl;
cout << endl;

cout << "推荐使用解法二作为通用解决方案" << endl;
}

int main() {
 cout << "==== 最小覆盖子串 算法实现 ===" << endl;
 cout << endl;

 cout << "==== 测试用例 ===" << endl;
 test();

 cout << "==== 性能测试 ===" << endl;
 performanceTest();

 cout << "==== 算法分析 ===" << endl;
 algorithmAnalysis();

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

文件: Code30\_MinimumWindowSubstring.java

=====

```
package class050;

import java.util.HashMap;
import java.util.Map;

/**
 * LeetCode 76. 最小覆盖子串 (Minimum Window Substring)
 *
 * 题目描述：
 * 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。
 * - 如果 s 中存在这样的子串，我们保证它是唯一的答案。
 *
 * 示例 1：
 * 输入：s = "ADOBECODEBANC", t = "ABC"
 * 输出："BANC"
 * 解释：最小覆盖子串 "BANC" 包含 'A', 'B', 'C' 各一个。
 *
 * 示例 2：
 * 输入：s = "a", t = "a"
 * 输出："a"
 *
 * 示例 3：
 * 输入：s = "a", t = "aa"
 * 输出：""
 * 解释：t 中有两个 'a'，但 s 中只有一个 'a'，所以返回空字符串。
 *
 * 提示：
 * - $1 \leq s.length, t.length \leq 10^5$
 * - s 和 t 由英文字母组成
 *
 * 题目链接：https://leetcode.cn/problems/minimum-window-substring/
 *
 * 解题思路：
 * 这道题可以使用滑动窗口（双指针）的方法来解决：
 *
 * 方法一（滑动窗口 + 哈希表）：
 * 1. 使用两个指针 left 和 right 表示当前窗口的左右边界
 * 2. 使用哈希表记录 t 中每个字符的出现次数
 * 3. 使用另一个哈希表记录当前窗口中包含 t 字符的情况
```

```
* 4. 移动右指针扩展窗口，直到窗口包含 t 的所有字符
* 5. 然后移动左指针收缩窗口，找到最小覆盖子串
*
* 时间复杂度: O(n + m)，n 为 s 长度，m 为 t 长度
* 空间复杂度: O(m)，存储 t 的字符频率
* 是否最优解: 是
*/
```

```
public class Code30_MinimumWindowSubstring {

 /**
 * 解法一：滑动窗口 + 哈希表（最优解）
 *
 * @param s 源字符串
 * @param t 目标字符串
 * @return 最小覆盖子串
 */
 public static String minWindow(String s, String t) {
 if (s == null || t == null || s.length() == 0 || t.length() == 0) {
 return "";
 }

 // 记录 t 中每个字符的出现次数
 Map<Character, Integer> targetMap = new HashMap<>();
 for (char c : t.toCharArray()) {
 targetMap.put(c, targetMap.getOrDefault(c, 0) + 1);
 }

 // 记录当前窗口中字符的出现次数
 Map<Character, Integer> windowMap = new HashMap<>();

 int left = 0, right = 0;
 int minLen = Integer.MAX_VALUE;
 int minStart = 0;
 int required = targetMap.size(); // 需要匹配的字符种类数
 int formed = 0; // 当前窗口中已匹配的字符种类数

 while (right < s.length()) {
 char rightChar = s.charAt(right);
 windowMap.put(rightChar, windowMap.getOrDefault(rightChar, 0) + 1);

 // 如果当前字符在 t 中，且窗口中出现次数等于 t 中出现次数，则 formed+1
 if (targetMap.containsKey(rightChar) &&
```

```

 windowMap.get(rightChar).intValue() == targetMap.get(rightChar).intValue()) {
 formed++;
 }

 // 当窗口包含 t 的所有字符时，尝试收缩窗口
 while (left <= right && formed == required) {
 char leftChar = s.charAt(left);

 // 更新最小覆盖子串
 if (right - left + 1 < minLen) {
 minLen = right - left + 1;
 minStart = left;
 }

 // 移动左指针
 windowMap.put(leftChar, windowMap.get(leftChar) - 1);
 if (targetMap.containsKey(leftChar) &&
 windowMap.get(leftChar).intValue() < targetMap.get(leftChar).intValue()) {
 formed--;
 }
 left++;
 }

 right++;
 }

 return minLen == Integer.MAX_VALUE ? "" : s.substring(minStart, minStart + minLen);
}

/**
 * 解法二：优化版滑动窗口（使用数组替代哈希表）
 *
 * @param s 源字符串
 * @param t 目标字符串
 * @return 最小覆盖子串
 */
public static String minWindowOptimized(String s, String t) {
 if (s == null || t == null || s.length() == 0 || t.length() == 0) {
 return "";
 }

 // 使用数组记录字符频率（假设字符为 ASCII）
 int[] targetFreq = new int[128];

```

```
int[] windowFreq = new int[128];

for (char c : t.toCharArray()) {
 targetFreq[c]++;
}

int left = 0, right = 0;
int minLen = Integer.MAX_VALUE;
int minStart = 0;
int required = 0;

// 计算需要匹配的字符种类数
for (int freq : targetFreq) {
 if (freq > 0) {
 required++;
 }
}

int formed = 0;

while (right < s.length()) {
 char rightChar = s.charAt(right);
 windowFreq[rightChar]++;

 // 如果当前字符在 t 中，且窗口中出现次数等于 t 中出现次数
 if (targetFreq[rightChar] > 0 && windowFreq[rightChar] == targetFreq[rightChar]) {
 formed++;
 }
}

// 当窗口包含 t 的所有字符时，尝试收缩窗口
while (left <= right && formed == required) {
 char leftChar = s.charAt(left);

 // 更新最小覆盖子串
 if (right - left + 1 < minLen) {
 minLen = right - left + 1;
 minStart = left;
 }
}

// 移动左指针
windowFreq[leftChar]--;
if (targetFreq[leftChar] > 0 && windowFreq[leftChar] < targetFreq[leftChar]) {
 formed--;
}
```

```

 }
 left++;
 }

 right++;
}

return minLen == Integer.MAX_VALUE ? "" : s.substring(minStart, minStart + minLen);
}

/**
 * 解法三：进一步优化的滑动窗口（跳过无关字符）
 *
 * @param s 源字符串
 * @param t 目标字符串
 * @return 最小覆盖子串
 */
public static String minWindowAdvanced(String s, String t) {
 if (s == null || t == null || s.length() == 0 || t.length() == 0) {
 return "";
 }

 // 使用数组记录字符频率
 int[] targetFreq = new int[128];
 for (char c : t.toCharArray()) {
 targetFreq[c]++;
 }

 // 预处理：只保留 s 中在 t 中出现的字符及其位置
 int count = t.length();
 int left = 0, right = 0;
 int minLen = Integer.MAX_VALUE;
 int minStart = 0;

 while (right < s.length()) {
 // 如果当前字符在 t 中，减少计数
 if (targetFreq[s.charAt(right)] > 0) {
 count--;
 }
 targetFreq[s.charAt(right)]--;
 right++;

 // 当计数为 0 时，表示窗口包含 t 的所有字符
 }
}

```

```

 while (count == 0) {
 // 更新最小覆盖子串
 if (right - left < minLen) {
 minLen = right - left;
 minStart = left;
 }

 // 移动左指针
 targetFreq[s.charAt(left)]++;
 if (targetFreq[s.charAt(left)] > 0) {
 count++;
 }
 left++;
 }

 }

 return minLen == Integer.MAX_VALUE ? "" : s.substring(minStart, minStart + minLen);
}

/**
 * 测试函数
 */
public static void test() {
 // 测试用例 1
 String s1 = "ADOBECODEBANC";
 String t1 = "ABC";
 String expected1 = "BANC";
 System.out.println("测试用例 1:");
 System.out.println("s = " + s1 + "\\", t = " + t1 + ")");
 System.out.println("解法一结果: " + minWindow(s1, t1));
 System.out.println("解法二结果: " + minWindowOptimized(s1, t1));
 System.out.println("解法三结果: " + minWindowAdvanced(s1, t1));
 System.out.println("期望: " + expected1);
 System.out.println();

 // 测试用例 2
 String s2 = "a";
 String t2 = "a";
 String expected2 = "a";
 System.out.println("测试用例 2:");
 System.out.println("s = " + s2 + "\\", t = " + t2 + ")");
 System.out.println("解法一结果: " + minWindow(s2, t2));
 System.out.println("解法二结果: " + minWindowOptimized(s2, t2));
}

```

```

System.out.println("解法三结果: " + minWindowAdvanced(s2, t2) + "\");
System.out.println("期望: " + expected2 + "\");
System.out.println();

// 测试用例 3
String s3 = "a";
String t3 = "aa";
String expected3 = "";
System.out.println("测试用例 3:");
System.out.println("s = " + s3 + "\", t = " + t3 + "\");
System.out.println("解法一结果: " + minWindow(s3, t3) + "\");
System.out.println("解法二结果: " + minWindowOptimized(s3, t3) + "\");
System.out.println("解法三结果: " + minWindowAdvanced(s3, t3) + "\");
System.out.println("期望: " + expected3 + "\");
System.out.println();

// 测试用例 4 - 边界情况: s 和 t 相同
String s4 = "abc";
String t4 = "abc";
String expected4 = "abc";
System.out.println("测试用例 4 (s 和 t 相同) :");
System.out.println("s = " + s4 + "\", t = " + t4 + "\");
System.out.println("解法一结果: " + minWindow(s4, t4) + "\");
System.out.println("解法二结果: " + minWindowOptimized(s4, t4) + "\");
System.out.println("解法三结果: " + minWindowAdvanced(s4, t4) + "\");
System.out.println("期望: " + expected4 + "\");
System.out.println();

// 测试用例 5 - 边界情况: t 不在 s 中
String s5 = "abcdef";
String t5 = "xyz";
String expected5 = "";
System.out.println("测试用例 5 (t 不在 s 中) :");
System.out.println("s = " + s5 + "\", t = " + t5 + "\");
System.out.println("解法一结果: " + minWindow(s5, t5) + "\");
System.out.println("解法二结果: " + minWindowOptimized(s5, t5) + "\");
System.out.println("解法三结果: " + minWindowAdvanced(s5, t5) + "\");
System.out.println("期望: " + expected5 + "\");
System.out.println();
}

/**
 * 性能测试

```

```
/*
public static void performanceTest() {
 // 创建长字符串进行性能测试
 StringBuilder sbS = new StringBuilder();
 StringBuilder sbT = new StringBuilder();

 // s: 包含重复模式的字符串
 for (int i = 0; i < 10000; i++) {
 sbS.append("ABCDEFG");
 }
 String longS = sbS.toString();

 // t: 需要查找的子串
 sbT.append("ABC");
 String longT = sbT.toString();

 // 测试解法一的性能
 long startTime = System.nanoTime();
 String result1 = minWindow(longS, longT);
 long endTime = System.nanoTime();
 long duration1 = (endTime - startTime) / 1000000; // 转换为毫秒
 System.out.println("解法一（哈希表）耗时: " + duration1 + "ms, 结果长度: " +
result1.length());

 // 测试解法二的性能
 startTime = System.nanoTime();
 String result2 = minWindowOptimized(longS, longT);
 endTime = System.nanoTime();
 long duration2 = (endTime - startTime) / 1000000;
 System.out.println("解法二（数组优化）耗时: " + duration2 + "ms, 结果长度: " +
result2.length());

 // 测试解法三的性能
 startTime = System.nanoTime();
 String result3 = minWindowAdvanced(longS, longT);
 endTime = System.nanoTime();
 long duration3 = (endTime - startTime) / 1000000;
 System.out.println("解法三（高级优化）耗时: " + duration3 + "ms, 结果长度: " +
result3.length());

 // 验证结果一致性
 System.out.println("所有解法结果一致: " + result1.equals(result2) &&
result2.equals(result3));
}
```

```
}

/**
 * 算法分析
 */
public static void algorithmAnalysis() {
 System.out.println("== 算法分析 ==");
 System.out.println("1. 解法一（滑动窗口 + 哈希表）");
 System.out.println(" - 时间复杂度: O(n + m) - n 为 s 长度, m 为 t 长度");
 System.out.println(" - 空间复杂度: O(m) - 存储 t 的字符频率");
 System.out.println(" - 优点: 通用性强, 适用于任意字符集");
 System.out.println(" - 缺点: 哈希表操作有一定开销");
 System.out.println();
 System.out.println("2. 解法二（数组优化版）");
 System.out.println(" - 时间复杂度: O(n + m)");
 System.out.println(" - 空间复杂度: O(1) - 固定大小的数组");
 System.out.println(" - 优点: 效率高, 适用于 ASCII 字符集");
 System.out.println(" - 缺点: 仅适用于有限字符集");
 System.out.println();
 System.out.println("3. 解法三（进一步优化）");
 System.out.println(" - 时间复杂度: O(n)");
 System.out.println(" - 空间复杂度: O(1)");
 System.out.println(" - 优点: 最优化实现, 跳过无关字符");
 System.out.println(" - 缺点: 实现相对复杂");
 System.out.println();
 System.out.println("推荐使用解法二作为通用解决方案");
}

public static void main(String[] args) {
 System.out.println("== 最小覆盖子串 算法实现 ==");
 System.out.println();
 System.out.println("== 测试用例 ==");
 test();

 System.out.println("== 性能测试 ==");
 performanceTest();

 System.out.println("== 算法分析 ==");
 algorithmAnalysis();
}
```

```
 }
}
```

```
=====
```

文件: Code30\_MinimumWindowSubstring.py

```
=====
```

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

```
"""
```

LeetCode 76. 最小覆盖子串 (Minimum Window Substring)

题目描述:

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。
- 如果  $s$  中存在这样的子串，我们保证它是唯一的答案。

示例 1:

输入:  $s = "ADOBECODEBANC"$ ,  $t = "ABC"$

输出: "BANC"

解释: 最小覆盖子串 "BANC" 包含 'A', 'B', 'C' 各一个。

示例 2:

输入:  $s = "a"$ ,  $t = "a"$

输出: "a"

示例 3:

输入:  $s = "a"$ ,  $t = "aa"$

输出: ""

解释:  $t$  中有两个 'a'，但  $s$  中只有一个 'a'，所以返回空字符串。

提示:

- $1 \leq s.length, t.length \leq 10^5$
- $s$  和  $t$  由英文字母组成

题目链接: <https://leetcode.cn/problems/minimum-window-substring/>

解题思路:

这道题可以使用滑动窗口（双指针）的方法来解决:

方法一（滑动窗口 + 字典）：

1. 使用两个指针 left 和 right 表示当前窗口的左右边界
2. 使用字典记录 t 中每个字符的出现次数
3. 使用另一个字典记录当前窗口中包含 t 字符的情况
4. 移动右指针扩展窗口，直到窗口包含 t 的所有字符
5. 然后移动左指针收缩窗口，找到最小覆盖子串

时间复杂度： $O(n + m)$ ，n 为 s 长度，m 为 t 长度

空间复杂度： $O(m)$ ，存储 t 的字符频率

是否最优解：是

"""

```
from collections import defaultdict
```

```
class Solution:
```

"""

最小覆盖子串解决方案类

"""

@staticmethod

```
def min_window(s: str, t: str) -> str:
```

"""

解法一：滑动窗口 + 字典（最优解）

Args:

s: 源字符串

t: 目标字符串

Returns:

str: 最小覆盖子串

时间复杂度： $O(n + m)$  – n 为 s 长度，m 为 t 长度

空间复杂度： $O(m)$  – 存储 t 的字符频率

"""

```
if not s or not t or len(s) < len(t):
 return ""
```

# 记录 t 中每个字符的出现次数

```
target_dict = defaultdict(int)
```

```
for char in t:
```

```
 target_dict[char] += 1
```

```

记录当前窗口中字符的出现次数
window_dict = defaultdict(int)

left, right = 0, 0
min_len = float('inf')
min_start = 0
required = len(target_dict) # 需要匹配的字符种类数
formed = 0 # 当前窗口中已匹配的字符种类数

while right < len(s):
 right_char = s[right]
 window_dict[right_char] += 1

 # 如果当前字符在 t 中，且窗口中出现次数等于 t 中出现次数
 if (right_char in target_dict and
 window_dict[right_char] == target_dict[right_char]):
 formed += 1

 # 当窗口包含 t 的所有字符时，尝试收缩窗口
 while left <= right and formed == required:
 left_char = s[left]

 # 更新最小覆盖子串
 if right - left + 1 < min_len:
 min_len = right - left + 1
 min_start = left

 # 移动左指针
 window_dict[left_char] -= 1
 if (left_char in target_dict and
 window_dict[left_char] < target_dict[left_char]):
 formed -= 1
 left += 1

 right += 1

return "" if min_len == float('inf') else s[min_start:min_start + min_len]

```

@staticmethod

```

def min_window_optimized(s: str, t: str) -> str:
 """
 解法二：优化版滑动窗口（使用数组替代字典）

```

Args:

s: 源字符串  
t: 目标字符串

Returns:

str: 最小覆盖子串

时间复杂度:  $O(n + m)$

空间复杂度:  $O(1)$  - 固定大小的数组

"""

```
if not s or not t or len(s) < len(t):
 return ""
```

# 使用数组记录字符频率（假设字符为 ASCII）

```
target_freq = [0] * 128
window_freq = [0] * 128
```

```
for char in t:
 target_freq[ord(char)] += 1
```

```
left, right = 0, 0
min_len = float('inf')
min_start = 0
required = 0
```

# 计算需要匹配的字符种类数

```
for freq in target_freq:
 if freq > 0:
 required += 1
```

formed = 0

```
while right < len(s):
 right_char = s[right]
 right_char_code = ord(right_char)
 window_freq[right_char_code] += 1
```

# 如果当前字符在 t 中，且窗口中出现次数等于 t 中出现次数

```
if (target_freq[right_char_code] > 0 and
 window_freq[right_char_code] == target_freq[right_char_code]):
 formed += 1
```

# 当窗口包含 t 的所有字符时，尝试收缩窗口

```

while left <= right and formed == required:
 left_char = s[left]
 left_char_code = ord(left_char)

 # 更新最小覆盖子串
 if right - left + 1 < min_len:
 min_len = right - left + 1
 min_start = left

 # 移动左指针
 window_freq[left_char_code] -= 1
 if (target_freq[left_char_code] > 0 and
 window_freq[left_char_code] < target_freq[left_char_code]):
 formed -= 1
 left += 1

 right += 1

return "" if min_len == float('inf') else s[min_start:min_start + min_len]

```

@staticmethod

```

def min_window_advanced(s: str, t: str) -> str:
 """

```

解法三：进一步优化的滑动窗口（跳过无关字符）

Args:

```

s: 源字符串
t: 目标字符串

```

Returns:

```
str: 最小覆盖子串
```

时间复杂度: O(n)

空间复杂度: O(1)

```
"""

```

```

if not s or not t or len(s) < len(t):
 return ""

```

# 使用数组记录字符频率

```
target_freq = [0] * 128
```

```
for char in t:
```

```
 target_freq[ord(char)] += 1
```

```

预处理：只保留 s 中在 t 中出现的字符及其位置
count = len(t)
left, right = 0, 0
min_len = float('inf')
min_start = 0

while right < len(s):
 right_char_code = ord(s[right])

 # 如果当前字符在 t 中，减少计数
 if target_freq[right_char_code] > 0:
 count -= 1

 target_freq[right_char_code] -= 1
 right += 1

 # 当计数为 0 时，表示窗口包含 t 的所有字符
 while count == 0:
 # 更新最小覆盖子串
 if right - left < min_len:
 min_len = right - left
 min_start = left

 # 移动左指针
 left_char_code = ord(s[left])
 target_freq[left_char_code] += 1
 if target_freq[left_char_code] > 0:
 count += 1
 left += 1

return "" if min_len == float('inf') else s[min_start:min_start + min_len]

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

测试用例 1
s1 = "ADOBECODEBANC"
t1 = "ABC"
expected1 = "BANC"
print("测试用例 1:")

```

```
print(f"s = \'{s1}\', t = \'{t1}\'")
print(f"解法一结果: \'{solution.min_window(s1, t1)}\'")
print(f"解法二结果: \'{solution.min_window_optimized(s1, t1)}\'")
print(f"解法三结果: \'{solution.min_window_advanced(s1, t1)}\'")
print(f"期望: \'{expected1}\'")
print()
```

```
测试用例 2
s2 = "a"
t2 = "a"
expected2 = "a"
print("测试用例 2:")
print(f"s = \'{s2}\', t = \'{t2}\'")
print(f"解法一结果: \'{solution.min_window(s2, t2)}\'")
print(f"解法二结果: \'{solution.min_window_optimized(s2, t2)}\'")
print(f"解法三结果: \'{solution.min_window_advanced(s2, t2)}\'")
print(f"期望: \'{expected2}\'")
print()
```

```
测试用例 3
s3 = "a"
t3 = "aa"
expected3 = ""
print("测试用例 3:")
print(f"s = \'{s3}\', t = \'{t3}\'")
print(f"解法一结果: \'{solution.min_window(s3, t3)}\'")
print(f"解法二结果: \'{solution.min_window_optimized(s3, t3)}\'")
print(f"解法三结果: \'{solution.min_window_advanced(s3, t3)}\'")
print(f"期望: \'{expected3}\'")
print()
```

```
测试用例 4 - 边界情况: s 和 t 相同
s4 = "abc"
t4 = "abc"
expected4 = "abc"
print("测试用例 4 (s 和 t 相同) :")
print(f"s = \'{s4}\', t = \'{t4}\'")
print(f"解法一结果: \'{solution.min_window(s4, t4)}\'")
print(f"解法二结果: \'{solution.min_window_optimized(s4, t4)}\'")
print(f"解法三结果: \'{solution.min_window_advanced(s4, t4)}\'")
print(f"期望: \'{expected4}\'")
print()
```

```
测试用例 5 - 边界情况: t 不在 s 中
s5 = "abcdef"
t5 = "xyz"
expected5 = ""
print("测试用例 5 (t 不在 s 中) :")
print(f"s = \'{s5}\', t = \'{t5}\'")
print(f"解法一结果: \'{solution.min_window(s5, t5)}\'")
print(f"解法二结果: \'{solution.min_window_optimized(s5, t5)}\'")
print(f"解法三结果: \'{solution.min_window_advanced(s5, t5)}\'")
print(f"期望: \'{expected5}\'")
print()

def performance_test():
 """
 性能测试函数
 """
 import time
 solution = Solution()

 # 创建长字符串进行性能测试
 long_s = "ABCDEFG" * 10000
 long_t = "ABC"

 # 测试解法一的性能
 start_time = time.time()
 result1 = solution.min_window(long_s, long_t)
 end_time = time.time()
 duration1 = (end_time - start_time) * 1000 # 转换为毫秒
 print(f"解法一 (字典) 耗时: {duration1:.2f}ms, 结果长度: {len(result1)}")

 # 测试解法二的性能
 start_time = time.time()
 result2 = solution.min_window_optimized(long_s, long_t)
 end_time = time.time()
 duration2 = (end_time - start_time) * 1000
 print(f"解法二 (数组优化) 耗时: {duration2:.2f}ms, 结果长度: {len(result2)}")

 # 测试解法三的性能
 start_time = time.time()
 result3 = solution.min_window_advanced(long_s, long_t)
 end_time = time.time()
 duration3 = (end_time - start_time) * 1000
 print(f"解法三 (高级优化) 耗时: {duration3:.2f}ms, 结果长度: {len(result3)}")
```

```
验证结果一致性
print(f"所有解法结果一致: {result1 == result2 == result3}")

def algorithm_analysis():
 """
 算法分析函数
 """
 print("== 算法分析 ===")
 print("1. 解法一（滑动窗口 + 字典）")
 print(" - 时间复杂度: O(n + m) - n 为 s 长度, m 为 t 长度")
 print(" - 空间复杂度: O(m) - 存储 t 的字符频率")
 print(" - 优点: 通用性强, 适用于任意字符集")
 print(" - 缺点: 字典操作有一定开销")
 print()

 print("2. 解法二（数组优化版）")
 print(" - 时间复杂度: O(n + m)")
 print(" - 空间复杂度: O(1) - 固定大小的数组")
 print(" - 优点: 效率高, 适用于 ASCII 字符集")
 print(" - 缺点: 仅适用于有限字符集")
 print()

 print("3. 解法三（进一步优化）")
 print(" - 时间复杂度: O(n)")
 print(" - 空间复杂度: O(1)")
 print(" - 优点: 最优化实现, 跳过无关字符")
 print(" - 缺点: 实现相对复杂")
 print()

 print("推荐使用解法二作为通用解决方案")

if __name__ == "__main__":
 print("== 最小覆盖子串 算法实现 ===")
 print()

 print("== 测试用例 ===")
 test()

 print("== 性能测试 ===")
 performance_test()

 print("== 算法分析 ===")
```

```
algorithm_analysis()
```

```
=====
```

文件: Code31\_ValidPalindrome.cpp

```
=====
```

```
#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>
#include <chrono>
using namespace std;

/***
 * LeetCode 125. 验证回文串 (Valid Palindrome)
 *
 * 题目描述:
 * 如果在将所有大写字符转换为小写字符、并移除所有非字母数字字符之后，短语正着读和反着读都一样，则可以认为该短语是一个 回文串 。
 * 字母和数字都属于字母数字字符。
 * 给你一个字符串 s，如果它是 回文串 ，返回 true ；否则，返回 false 。
 *
 * 示例 1:
 * 输入: s = "A man, a plan, a canal: Panama"
 * 输出: true
 * 解释: "amanaplanacanalpanama" 是回文串。
 *
 * 示例 2:
 * 输入: s = "race a car"
 * 输出: false
 * 解释: "raceacar" 不是回文串。
 *
 * 示例 3:
 * 输入: s = " "
 * 输出: true
 * 解释: 在移除非字母数字字符后，s 变为 "" 。由于空字符串正着反着读都一样，所以是回文串。
 *
 * 提示:
 * 1 <= s.length <= 2 * 10^5
 * s 仅由可打印的 ASCII 字符组成
 *
 * 题目链接: https://leetcode.cn/problems/valid-palindrome/
 */
```

- \* 解题思路:
- \* 这道题可以使用双指针的方法来解决:
- \*
- \* 方法一 (双指针 + 字符处理):
- \* 1. 使用两个指针 left 和 right 分别指向字符串的首尾
- \* 2. 跳过非字母数字字符, 只比较字母数字字符
- \* 3. 比较左右指针指向的字符 (忽略大小写)
- \* 4. 如果所有字符都匹配, 则返回 true, 否则返回 false
- \*
- \* 时间复杂度: O(n), n 为字符串长度
- \* 空间复杂度: O(1)
- \* 是否最优解: 是
- \*/

```
class Solution {
public:
 /**
 * 解法一: 双指针 (最优解)
 *
 * @param s 输入字符串
 * @return 是否为回文串
 */
 static bool isPalindrome(const string& s) {
 if (s.empty()) {
 return true;
 }

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

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

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

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

```
 return false;
 }

 left++;
 right--;
}

return true;
}

/***
 * 解法二：使用字符串反转比较
 *
 * @param s 输入字符串
 * @return 是否为回文串
 */
static bool isPalindromeString(const string& s) {
 if (s.empty()) {
 return true;
 }

 // 过滤非字母数字字符并转换为小写
 string filtered;
 for (char c : s) {
 if (isalnum(c)) {
 filtered += tolower(c);
 }
 }

 // 比较原字符串和反转后的字符串
 string reversed = filtered;
 reverse(reversed.begin(), reversed.end());

 return filtered == reversed;
}

/***
 * 解法三：优化的双指针实现（避免重复计算）
 *
 * @param s 输入字符串
 * @return 是否为回文串
 */
static bool isPalindromeOptimized(const string& s) {
```

```
if (s.empty()) {
 return true;
}

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

while (left < right) {
 char leftChar = s[left];
 char rightChar = s[right];

 // 如果左字符不是字母数字，跳过
 if (!isAlphanumeric(leftChar)) {
 left++;
 continue;
 }

 // 如果右字符不是字母数字，跳过
 if (!isAlphanumeric(rightChar)) {
 right--;
 continue;
 }

 // 比较字符（忽略大小写）
 if (toLowerCase(leftChar) != toLowerCase(rightChar)) {
 return false;
 }

 left++;
 right--;
}

return true;
}

private:
/***
 * 判断字符是否为字母或数字
 *
 * @param c 字符
 * @return 是否为字母数字
 */
static bool isAlphanumeric(char c) {
```

```

 return (c >= 'a' && c <= 'z') ||
 (c >= 'A' && c <= 'Z') ||
 (c >= '0' && c <= '9');
 }

/***
 * 将字符转换为小写（自定义实现）
 *
 * @param c 字符
 * @return 小写字符
 */
static char toLower(char c) {
 if (c >= 'A' && c <= 'Z') {
 return c - 'A' + 'a';
 }
 return c;
}

};

/***
 * 测试函数
 */
void test() {
 // 测试用例 1
 string s1 = "A man, a plan, a canal: Panama";
 bool expected1 = true;
 cout << "测试用例 1:" << endl;
 cout << "输入: \" " << s1 << "\" " << endl;
 cout << "解法一结果: " << (Solution::isPalindrome(s1) ? "true" : "false") << endl;
 cout << "解法二结果: " << (Solution::isPalindromeString(s1) ? "true" : "false") << endl;
 cout << "解法三结果: " << (Solution::isPalindromeOptimized(s1) ? "true" : "false") << endl;
 cout << "期望: " << (expected1 ? "true" : "false") << endl;
 cout << endl;

 // 测试用例 2
 string s2 = "race a car";
 bool expected2 = false;
 cout << "测试用例 2:" << endl;
 cout << "输入: \" " << s2 << "\" " << endl;
 cout << "解法一结果: " << (Solution::isPalindrome(s2) ? "true" : "false") << endl;
 cout << "解法二结果: " << (Solution::isPalindromeString(s2) ? "true" : "false") << endl;
 cout << "解法三结果: " << (Solution::isPalindromeOptimized(s2) ? "true" : "false") << endl;
 cout << "期望: " << (expected2 ? "true" : "false") << endl;
}

```

```
cout << endl;

// 测试用例 3
string s3 = "";
bool expected3 = true;
cout << "测试用例 3:" << endl;
cout << "输入: \\" << s3 << "\\" << endl;
cout << "解法一结果: " << (Solution::isPalindrome(s3) ? "true" : "false") << endl;
cout << "解法二结果: " << (Solution::isPalindromeString(s3) ? "true" : "false") << endl;
cout << "解法三结果: " << (Solution::isPalindromeOptimized(s3) ? "true" : "false") << endl;
cout << "期望: " << (expected3 ? "true" : "false") << endl;
cout << endl;

// 测试用例 4 - 边界情况: 空字符串
string s4 = "";
bool expected4 = true;
cout << "测试用例 4 (空字符串) :" << endl;
cout << "输入: \\" << s4 << "\\" << endl;
cout << "解法一结果: " << (Solution::isPalindrome(s4) ? "true" : "false") << endl;
cout << "解法二结果: " << (Solution::isPalindromeString(s4) ? "true" : "false") << endl;
cout << "解法三结果: " << (Solution::isPalindromeOptimized(s4) ? "true" : "false") << endl;
cout << "期望: " << (expected4 ? "true" : "false") << endl;
cout << endl;

// 测试用例 5 - 边界情况: 纯数字
string s5 = "12321";
bool expected5 = true;
cout << "测试用例 5 (纯数字) :" << endl;
cout << "输入: \\" << s5 << "\\" << endl;
cout << "解法一结果: " << (Solution::isPalindrome(s5) ? "true" : "false") << endl;
cout << "解法二结果: " << (Solution::isPalindromeString(s5) ? "true" : "false") << endl;
cout << "解法三结果: " << (Solution::isPalindromeOptimized(s5) ? "true" : "false") << endl;
cout << "期望: " << (expected5 ? "true" : "false") << endl;
cout << endl;

// 测试用例 6 - 边界情况: 混合字符
string s6 = "OP";
bool expected6 = false;
cout << "测试用例 6 (混合字符) :" << endl;
cout << "输入: \\" << s6 << "\\" << endl;
cout << "解法一结果: " << (Solution::isPalindrome(s6) ? "true" : "false") << endl;
cout << "解法二结果: " << (Solution::isPalindromeString(s6) ? "true" : "false") << endl;
cout << "解法三结果: " << (Solution::isPalindromeOptimized(s6) ? "true" : "false") << endl;
```

```
cout << "期望: " << (expected6 ? "true" : "false") << endl;
cout << endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
 // 创建长字符串进行性能测试
 string longString;
 for (int i = 0; i < 100000; i++) {
 if (i % 3 == 0) {
 longString += "!@#";
 } else if (i % 3 == 1) {
 longString += "abc";
 } else {
 longString += "123";
 }
 }

 // 测试解法一的性能
 auto start = chrono::high_resolution_clock::now();
 bool result1 = Solution::isPalindrome(longString);
 auto end = chrono::high_resolution_clock::now();
 auto duration1 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法一(双指针)耗时: " << duration1 << "ms, 结果: " << (result1 ? "true" : "false")
 << endl;

 // 测试解法二的性能
 start = chrono::high_resolution_clock::now();
 bool result2 = Solution::isPalindromeString(longString);
 end = chrono::high_resolution_clock::now();
 auto duration2 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法二(字符串反转)耗时: " << duration2 << "ms, 结果: " << (result2 ? "true" :
 "false") << endl;

 // 测试解法三的性能
 start = chrono::high_resolution_clock::now();
 bool result3 = Solution::isPalindromeOptimized(longString);
 end = chrono::high_resolution_clock::now();
 auto duration3 = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "解法三(优化双指针)耗时: " << duration3 << "ms, 结果: " << (result3 ? "true" :
 "false") << endl;
```

```

// 验证结果一致性
cout << "所有解法结果一致: " << (result1 == result2 && result2 == result3 ? "true" : "false")
<< endl;
}

/***
 * 边界条件测试函数
 */
void boundaryTest() {
 // 测试极端长字符串
 string extremeString(1000000, 'a');

 auto start = chrono::high_resolution_clock::now();
 bool result = Solution::isPalindrome(extremeString);
 auto end = chrono::high_resolution_clock::now();
 auto duration = chrono::duration_cast<chrono::milliseconds>(end - start).count();
 cout << "极端长字符串测试耗时: " << duration << "ms, 结果: " << (result ? "true" : "false")
 << endl;
}

/***
 * 算法分析函数
 */
void algorithmAnalysis() {
 cout << "==== 算法分析 ===" << endl;
 cout << "1. 解法一（双指针）" << endl;
 cout << " - 时间复杂度: O(n) - 每个字符最多被访问一次" << endl;
 cout << " - 空间复杂度: O(1) - 只使用常数级别的额外空间" << endl;
 cout << " - 优点: 原地操作, 空间效率高" << endl;
 cout << " - 缺点: 需要处理字符过滤逻辑" << endl;
 cout << endl;

 cout << "2. 解法二（字符串反转）" << endl;
 cout << " - 时间复杂度: O(n) - 需要遍历字符串两次" << endl;
 cout << " - 空间复杂度: O(n) - 需要额外的字符串存储空间" << endl;
 cout << " - 优点: 实现简单, 易于理解" << endl;
 cout << " - 缺点: 空间效率较低" << endl;
 cout << endl;

 cout << "3. 解法三（优化双指针）" << endl;
 cout << " - 时间复杂度: O(n)" << endl;
 cout << " - 空间复杂度: O(1)" << endl;
}

```

```

cout << " - 优点：避免重复字符检查，效率最高" << endl;
cout << " - 缺点：实现相对复杂" << endl;
cout << endl;

cout << "推荐使用解法一作为通用解决方案" << endl;
}

int main() {
 cout << "==== 验证回文串 算法实现 ===" << endl;
 cout << endl;

 cout << "==== 测试用例 ===" << endl;
 test();

 cout << "==== 性能测试 ===" << endl;
 performanceTest();

 cout << "==== 边界条件测试 ===" << endl;
 boundaryTest();

 cout << "==== 算法分析 ===" << endl;
 algorithmAnalysis();

 return 0;
}

```

=====

文件：Code31\_ValidPalindrome.java

=====

```

package class050;

/**
 * LeetCode 125. 验证回文串 (Valid Palindrome)
 *
 * 题目描述：
 * 如果在将所有大写字符转换为小写字符、并移除所有非字母数字字符之后，短语正着读和反着读都一样，则
 * 可以认为该短语是一个 回文串 。
 * 字母和数字都属于字母数字字符。
 * 给你一个字符串 s，如果它是 回文串 ，返回 true ；否则，返回 false 。
 *
 * 示例 1：
 * 输入：s = "A man, a plan, a canal: Panama"

```

```
* 输出: true
* 解释: "amanaplanacanalpanama" 是回文串。
*
* 示例 2:
* 输入: s = "race a car"
* 输出: false
* 解释: "raceacar" 不是回文串。
*
* 示例 3:
* 输入: s = " "
* 输出: true
* 解释: 在移除非字母数字字符后, s 变为 " "。由于空字符串正着反着读都一样, 所以是回文串。
*
* 提示:
* 1 <= s.length <= 2 * 10^5
* s 仅由可打印的 ASCII 字符组成
*
* 题目链接: https://leetcode.cn/problems/valid-palindrome/
*
* 解题思路:
* 这道题可以使用双指针的方法来解决:
*
* 方法一 (双指针 + 字符处理):
* 1. 使用两个指针 left 和 right 分别指向字符串的首尾
* 2. 跳过非字母数字字符, 只比较字母数字字符
* 3. 比较左右指针指向的字符 (忽略大小写)
* 4. 如果所有字符都匹配, 则返回 true, 否则返回 false
*
* 时间复杂度: O(n), n 为字符串长度
* 空间复杂度: O(1)
* 是否最优解: 是
*/

```

```
public class Code31_ValidPalindrome {

 /**
 * 解法一: 双指针 (最优解)
 *
 * @param s 输入字符串
 * @return 是否为回文串
 */
 public static boolean isPalindrome(String s) {
 if (s == null) {

```

```
 return false;
 }

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

 while (left < right) {
 // 跳过非字母数字字符 (左指针)
 while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {
 left++;
 }

 // 跳过非字母数字字符 (右指针)
 while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {
 right--;
 }

 // 比较字符 (忽略大小写)
 if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right)))
 {
 return false;
 }

 left++;
 right--;
 }

 return true;
}

/**
 * 解法二：使用 StringBuilder 反转比较
 *
 * @param s 输入字符串
 * @return 是否为回文串
 */
public static boolean isPalindromeStringBuilder(String s) {
 if (s == null) {
 return false;
 }

 // 过滤非字母数字字符并转换为小写
 StringBuilder filtered = new StringBuilder();
```

```
for (char c : s.toCharArray()) {
 if (Character.isLetterOrDigit(c)) {
 filtered.append(Character.toLowerCase(c));
 }
}

// 比较原字符串和反转后的字符串
String original = filtered.toString();
String reversed = filtered.reverse().toString();

return original.equals(reversed);
}

/***
 * 解法三：优化的双指针实现（避免重复计算）
 *
 * @param s 输入字符串
 * @return 是否为回文串
 */
public static boolean isPalindromeOptimized(String s) {
 if (s == null) {
 return false;
 }

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

 while (left < right) {
 char leftChar = s.charAt(left);
 char rightChar = s.charAt(right);

 // 如果左字符不是字母数字，跳过
 if (!isAlphanumeric(leftChar)) {
 left++;
 continue;
 }

 // 如果右字符不是字母数字，跳过
 if (!isAlphanumeric(rightChar)) {
 right--;
 continue;
 }

 if (leftChar != rightChar) {
 return false;
 }

 left++;
 right--;
 }

 return true;
}
```

```
// 比较字符（忽略大小写）
if (toLowerCase(leftChar) != toLowerCase(rightChar)) {
 return false;
}

left++;
right--;
}

return true;
}

/***
 * 判断字符是否为字母或数字
 *
 * @param c 字符
 * @return 是否为字母数字
 */
private static boolean isAlphanumeric(char c) {
 return (c >= 'a' && c <= 'z') ||
 (c >= 'A' && c <= 'Z') ||
 (c >= '0' && c <= '9');
}

/***
 * 将字符转换为小写（自定义实现，避免调用 Character.toLowerCase）
 *
 * @param c 字符
 * @return 小写字符
 */
private static char toLowerCase(char c) {
 if (c >= 'A' && c <= 'Z') {
 return (char)(c - 'A' + 'a');
 }
 return c;
}

/***
 * 测试函数
 */
public static void test() {
 // 测试用例 1
 String s1 = "A man, a plan, a canal: Panama";
}
```

```
boolean expected1 = true;
System.out.println("测试用例 1:");
System.out.println("输入: " + s1 + ")");
System.out.println("解法一结果: " + isPalindrome(s1));
System.out.println("解法二结果: " + isPalindromeStringBuilder(s1));
System.out.println("解法三结果: " + isPalindromeOptimized(s1));
System.out.println("期望: " + expected1);
System.out.println();

// 测试用例 2
String s2 = "race a car";
boolean expected2 = false;
System.out.println("测试用例 2:");
System.out.println("输入: " + s2 + ")");
System.out.println("解法一结果: " + isPalindrome(s2));
System.out.println("解法二结果: " + isPalindromeStringBuilder(s2));
System.out.println("解法三结果: " + isPalindromeOptimized(s2));
System.out.println("期望: " + expected2);
System.out.println();

// 测试用例 3
String s3 = " ";
boolean expected3 = true;
System.out.println("测试用例 3:");
System.out.println("输入: " + s3 + ")");
System.out.println("解法一结果: " + isPalindrome(s3));
System.out.println("解法二结果: " + isPalindromeStringBuilder(s3));
System.out.println("解法三结果: " + isPalindromeOptimized(s3));
System.out.println("期望: " + expected3);
System.out.println();

// 测试用例 4 - 边界情况: 空字符串
String s4 = "";
boolean expected4 = true;
System.out.println("测试用例 4 (空字符串):");
System.out.println("输入: " + s4 + ")");
System.out.println("解法一结果: " + isPalindrome(s4));
System.out.println("解法二结果: " + isPalindromeStringBuilder(s4));
System.out.println("解法三结果: " + isPalindromeOptimized(s4));
System.out.println("期望: " + expected4);
System.out.println();

// 测试用例 5 - 边界情况: 纯数字
```

```
String s5 = "12321";
boolean expected5 = true;
System.out.println("测试用例 5 (纯数字) :");
System.out.println("输入: " + s5 + ")");
System.out.println("解法一结果: " + isPalindrome(s5));
System.out.println("解法二结果: " + isPalindromeStringBuilder(s5));
System.out.println("解法三结果: " + isPalindromeOptimized(s5));
System.out.println("期望: " + expected5);
System.out.println();

// 测试用例 6 - 边界情况: 混合字符
String s6 = "OP";
boolean expected6 = false;
System.out.println("测试用例 6 (混合字符) :");
System.out.println("输入: " + s6 + ")");
System.out.println("解法一结果: " + isPalindrome(s6));
System.out.println("解法二结果: " + isPalindromeStringBuilder(s6));
System.out.println("解法三结果: " + isPalindromeOptimized(s6));
System.out.println("期望: " + expected6);
System.out.println();
}

/**
 * 性能测试
 */
public static void performanceTest() {
 // 创建长字符串进行性能测试
 StringBuilder sb = new StringBuilder();
 // 添加大量非字母数字字符和字母数字字符混合
 for (int i = 0; i < 100000; i++) {
 if (i % 3 == 0) {
 sb.append("!@#");
 } else if (i % 3 == 1) {
 sb.append("abc");
 } else {
 sb.append("123");
 }
 }
 String longString = sb.toString();

 // 测试解法一的性能
 long startTime = System.nanoTime();
 boolean result1 = isPalindrome(longString);
```

```
long endTime = System.nanoTime();
long duration1 = (endTime - startTime) / 1000000; // 转换为毫秒
System.out.println("解法一(双指针)耗时: " + duration1 + "ms, 结果: " + result1);

// 测试解法二的性能
startTime = System.nanoTime();
boolean result2 = isPalindromeStringBuilder(longString);
endTime = System.nanoTime();
long duration2 = (endTime - startTime) / 1000000;
System.out.println("解法二(StringBuilder)耗时: " + duration2 + "ms, 结果: " + result2);

// 测试解法三的性能
startTime = System.nanoTime();
boolean result3 = isPalindromeOptimized(longString);
endTime = System.nanoTime();
long duration3 = (endTime - startTime) / 1000000;
System.out.println("解法三(优化双指针)耗时: " + duration3 + "ms, 结果: " + result3);

// 验证结果一致性
System.out.println("所有解法结果一致: " + (result1 == result2 && result2 == result3));
}

/**
 * 边界条件测试
 */
public static void boundaryTest() {
 // 测试null输入
 try {
 boolean result = isPalindrome(null);
 System.out.println("边界测试失败: null 输入没有抛出异常");
 } catch (NullPointerException e) {
 System.out.println("边界测试通过: null 输入正确抛出异常");
 }
}

// 测试极端长字符串
StringBuilder extremelyLong = new StringBuilder();
for (int i = 0; i < 1000000; i++) {
 extremelyLong.append('a');
}
String extremeString = extremelyLong.toString();

long startTime = System.nanoTime();
boolean result = isPalindrome(extremeString);
```

```
long endTime = System.nanoTime();
long duration = (endTime - startTime) / 1000000;
System.out.println("极端长字符串测试耗时: " + duration + "ms, 结果: " + result);
}

/**
 * 算法分析
 */
public static void algorithmAnalysis() {
 System.out.println("== 算法分析 ==");
 System.out.println("1. 解法一 (双指针)");
 System.out.println(" - 时间复杂度: O(n) - 每个字符最多被访问一次");
 System.out.println(" - 空间复杂度: O(1) - 只使用常数级别的额外空间");
 System.out.println(" - 优点: 原地操作, 空间效率高");
 System.out.println(" - 缺点: 需要处理字符过滤逻辑");
 System.out.println();
 System.out.println("2. 解法二 (StringBuilder 反转)");
 System.out.println(" - 时间复杂度: O(n) - 需要遍历字符串两次");
 System.out.println(" - 空间复杂度: O(n) - 需要额外的字符串存储空间");
 System.out.println(" - 优点: 实现简单, 易于理解");
 System.out.println(" - 缺点: 空间效率较低");
 System.out.println();
 System.out.println("3. 解法三 (优化双指针)");
 System.out.println(" - 时间复杂度: O(n)");
 System.out.println(" - 空间复杂度: O(1)");
 System.out.println(" - 优点: 避免重复字符检查, 效率最高");
 System.out.println(" - 缺点: 实现相对复杂");
 System.out.println();
 System.out.println("推荐使用解法一作为通用解决方案");
}

public static void main(String[] args) {
 System.out.println("== 验证回文串 算法实现 ==");
 System.out.println();
 System.out.println("== 测试用例 ==");
 test();
 System.out.println("== 性能测试 ==");
 performanceTest();
}
```

```
System.out.println("==> 边界条件测试 ==>");
boundaryTest();

System.out.println("==> 算法分析 ==>");
algorithmAnalysis();
}
}
```

=====

文件: Code31\_ValidPalindrome.py

```
=====
```

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

```
"""
```

```
LeetCode 125. 验证回文串 (Valid Palindrome)
```

题目描述:

如果在将所有大写字符转换为小写字符、并移除所有非字母数字字符之后，短语正着读和反着读都一样，则可以认为该短语是一个 回文串 。

字母和数字都属于字母数字字符。

给你一个字符串 s，如果它是 回文串 ，返回 true ；否则，返回 false 。

示例 1:

输入: s = "A man, a plan, a canal: Panama"

输出: true

解释: "amanaplanacanalpanama" 是回文串。

示例 2:

输入: s = "race a car"

输出: false

解释: "raceacar" 不是回文串。

示例 3:

输入: s = ""

输出: true

解释: 在移除非字母数字字符后，s 变为 "" 。由于空字符串正着反着读都一样，所以是回文串。

提示:

$1 \leq s.length \leq 2 * 10^5$

s 仅由可打印的 ASCII 字符组成

题目链接: <https://leetcode.cn/problems/valid-palindrome/>

解题思路:

这道题可以使用双指针的方法来解决:

方法一 (双指针 + 字符处理):

1. 使用两个指针 `left` 和 `right` 分别指向字符串的首尾
2. 跳过非字母数字字符, 只比较字母数字字符
3. 比较左右指针指向的字符 (忽略大小写)
4. 如果所有字符都匹配, 则返回 `true`, 否则返回 `false`

时间复杂度:  $O(n)$ ,  $n$  为字符串长度

空间复杂度:  $O(1)$

是否最优解: 是

"""

class Solution:

"""

验证回文串解决方案类

"""

@staticmethod

def is\_palindrome(s: str) -> bool:

"""

解法一: 双指针 (最优解)

Args:

s: 输入字符串

Returns:

bool: 是否为回文串

时间复杂度:  $O(n)$  - 每个字符最多被访问一次

空间复杂度:  $O(1)$  - 只使用常数级别的额外空间

"""

if s is None:

    return False

    left, right = 0, len(s) - 1

    while left < right:

        # 跳过非字母数字字符 (左指针)

```
while left < right and not s[left].isalnum():
 left += 1

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

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

left += 1
right -= 1

return True
```

```
@staticmethod
def is_palindrome_string(s: str) -> bool:
 """
```

解法二：使用字符串反转比较

Args:  
s: 输入字符串

Returns:  
bool: 是否为回文串

时间复杂度: O(n) - 需要遍历字符串两次  
空间复杂度: O(n) - 需要额外的字符串存储空间  
"""

```
if s is None:
 return False

过滤非字母数字字符并转换为小写
filtered = ''.join(char.lower() for char in s if char.isalnum())

比较原字符串和反转后的字符串
return filtered == filtered[::-1]
```

```
@staticmethod
def is_palindrome_optimized(s: str) -> bool:
 """
```

解法三：优化的双指针实现（避免重复计算）

Args:

s: 输入字符串

Returns:

bool: 是否为回文串

时间复杂度: O(n)

空间复杂度: O(1)

"""

if s is None:

    return False

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

while left < right:

    left\_char = s[left]

    right\_char = s[right]

# 如果左字符不是字母数字，跳过

if not Solution.\_is\_alphanumeric(left\_char):

    left += 1

    continue

# 如果右字符不是字母数字，跳过

if not Solution.\_is\_alphanumeric(right\_char):

    right -= 1

    continue

# 比较字符（忽略大小写）

if Solution.\_to\_lower(left\_char) != Solution.\_to\_lower(right\_char):

    return False

    left += 1

    right -= 1

return True

@staticmethod

def \_is\_alphanumeric(c: str) -> bool:

"""

判断字符是否为字母或数字

Args:

c: 字符

Returns:

bool: 是否为字母数字

"""

```
return ('a' <= c <= 'z') or ('A' <= c <= 'Z') or ('0' <= c <= '9')
```

@staticmethod

```
def _to_lower(c: str) -> str:
```

"""

将字符转换为小写（自定义实现）

Args:

c: 字符

Returns:

str: 小写字符

"""

```
if 'A' <= c <= 'Z':
 return chr(ord(c) - ord('A') + ord('a'))
return c
```

```
def test():
```

"""

测试函数

"""

```
solution = Solution()
```

# 测试用例 1

```
s1 = "A man, a plan, a canal: Panama"
```

```
expected1 = True
```

```
print("测试用例 1:")
```

```
print(f"输入: \"{s1}\")
```

```
print(f"解法一结果: {solution.is_palindrome(s1)}")
```

```
print(f"解法二结果: {solution.is_palindrome_string(s1)}")
```

```
print(f"解法三结果: {solution.is_palindrome_optimized(s1)}")
```

```
print(f"期望: {expected1}")
```

```
print()
```

# 测试用例 2

```
s2 = "race a car"
```

```
expected2 = False
```

```
print("测试用例 2:")
print(f"输入: \'{s2}\'")
print(f"解法一结果: {solution.is_palindrome(s2)}")
print(f"解法二结果: {solution.is_palindrome_string(s2)}")
print(f"解法三结果: {solution.is_palindrome_optimized(s2)}")
print(f"期望: {expected2}")
print()
```

```
测试用例 3
s3 = ""
expected3 = True
print("测试用例 3:")
print(f"输入: \'{s3}\'")
print(f"解法一结果: {solution.is_palindrome(s3)}")
print(f"解法二结果: {solution.is_palindrome_string(s3)}")
print(f"解法三结果: {solution.is_palindrome_optimized(s3)}")
print(f"期望: {expected3}")
print()
```

```
测试用例 4 - 边界情况: 空字符串
s4 = ""
expected4 = True
print("测试用例 4 (空字符串) :")
print(f"输入: \'{s4}\'")
print(f"解法一结果: {solution.is_palindrome(s4)}")
print(f"解法二结果: {solution.is_palindrome_string(s4)}")
print(f"解法三结果: {solution.is_palindrome_optimized(s4)}")
print(f"期望: {expected4}")
print()
```

```
测试用例 5 - 边界情况: 纯数字
s5 = "12321"
expected5 = True
print("测试用例 5 (纯数字) :")
print(f"输入: \'{s5}\'")
print(f"解法一结果: {solution.is_palindrome(s5)}")
print(f"解法二结果: {solution.is_palindrome_string(s5)}")
print(f"解法三结果: {solution.is_palindrome_optimized(s5)}")
print(f"期望: {expected5}")
print()
```

```
测试用例 6 - 边界情况: 混合字符
s6 = "0P"
```

```
expected6 = False
print("测试用例 6 (混合字符) :")
print(f"输入: \'{s6}\'")
print(f"解法一结果: {solution.is_palindrome(s6)}")
print(f"解法二结果: {solution.is_palindrome_string(s6)}")
print(f"解法三结果: {solution.is_palindrome_optimized(s6)}")
print(f"期望: {expected6}")
print()

def performance_test():
 """
 性能测试函数
 """
 import time
 solution = Solution()

 # 创建长字符串进行性能测试
 import random
 chars = "!@#$%^&*() abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789"
 long_string = ''.join(random.choice(chars) for _ in range(100000))

 # 测试解法一的性能
 start_time = time.time()
 result1 = solution.is_palindrome(long_string)
 end_time = time.time()
 duration1 = (end_time - start_time) * 1000 # 转换为毫秒
 print(f"解法一 (双指针) 耗时: {duration1:.2f}ms, 结果: {result1}")

 # 测试解法二的性能
 start_time = time.time()
 result2 = solution.is_palindrome_string(long_string)
 end_time = time.time()
 duration2 = (end_time - start_time) * 1000
 print(f"解法二 (字符串反转) 耗时: {duration2:.2f}ms, 结果: {result2}")

 # 测试解法三的性能
 start_time = time.time()
 result3 = solution.is_palindrome_optimized(long_string)
 end_time = time.time()
 duration3 = (end_time - start_time) * 1000
 print(f"解法三 (优化双指针) 耗时: {duration3:.2f}ms, 结果: {result3}")

 # 验证结果一致性
```

```
print(f"所有解法结果一致: {result1 == result2 == result3}")

def boundary_test():
 """
 边界条件测试函数
 """
 solution = Solution()

 # 测试 None 输入
 try:
 result = solution.is_palindrome(None)
 print("边界测试失败: None 输入没有抛出异常")
 except Exception as e:
 print(f"边界测试通过: None 输入正确抛出异常: {e}")

 # 测试极端长字符串
 extreme_string = 'a' * 1000000

 import time
 start_time = time.time()
 result = solution.is_palindrome(extreme_string)
 end_time = time.time()
 duration = (end_time - start_time) * 1000
 print(f"极端长字符串测试耗时: {duration:.2f}ms, 结果: {result}")

def algorithm_analysis():
 """
 算法分析函数
 """

 print("== 算法分析 ==")
 print("1. 解法一 (双指针)")
 print(" - 时间复杂度: O(n) - 每个字符最多被访问一次")
 print(" - 空间复杂度: O(1) - 只使用常数级别的额外空间")
 print(" - 优点: 原地操作, 空间效率高")
 print(" - 缺点: 需要处理字符过滤逻辑")
 print()

 print("2. 解法二 (字符串反转)")
 print(" - 时间复杂度: O(n) - 需要遍历字符串两次")
 print(" - 空间复杂度: O(n) - 需要额外的字符串存储空间")
 print(" - 优点: 实现简单, 易于理解")
 print(" - 缺点: 空间效率较低")
 print()
```

```
print("3. 解法三（优化双指针）")
print(" - 时间复杂度: O(n)")
print(" - 空间复杂度: O(1)")
print(" - 优点: 避免重复字符检查, 效率最高")
print(" - 缺点: 实现相对复杂")
print()

print("推荐使用解法一作为通用解决方案")

if __name__ == "__main__":
 print("== 验证回文串 算法实现 ==")
 print()

 print("== 测试用例 ==")
 test()

 print("== 性能测试 ==")
 performance_test()

 print("== 边界条件测试 ==")
 boundary_test()

 print("== 算法分析 ==")
 algorithm_analysis()

=====
```