

=====

文件夹: class097_StockTradingProblems

=====

[Markdown 文件]

=====

文件: AdditionalProblems.md

=====

股票问题系列补充题目

LeetCode 系列

1. 股票价格跨度

- **题目**: LeetCode 901. 股票价格跨度
- **链接**: <https://leetcode.cn/problems/online-stock-span/>
- **难度**: 中等
- **描述**: 设计一个算法收集某些股票的每日报价，并返回该股票当日价格的跨度。
- **解法**: 单调栈

2. 股价颜色填充

- **题目**: LeetCode 775. 全局倒置与局部倒置
- **链接**: <https://leetcode.cn/problems/global-and-local-inversions/>
- **难度**: 中等
- **描述**: 给你一个长度为 n 的整数数组 nums ，表示由范围 $[0, n - 1]$ 内所有整数组成的一个排列。
- **解法**: 数学分析

3. 最佳观光组合

- **题目**: LeetCode 1014. 最佳观光组合
- **链接**: <https://leetcode.cn/problems/best-sightseeing-pair/>
- **难度**: 中等
- **描述**: 给定正整数数组 values ， $\text{values}[i]$ 表示第 i 个观光景点的评分，求一对观光景点能取得的最高分。
- **解法**: 动态规划

4. 股票价格波动

- **题目**: LeetCode 2034. 股票价格波动
- **链接**: <https://leetcode.cn/problems/stock-price-fluctuation/>
- **难度**: 中等
- **描述**: 给你一支股票价格的波动序列，请你实现一个数据结构来处理这些波动。
- **解法**: 哈希表+优先队列

5. 最大股票收益

- **题目**: LeetCode 2291. 最大股票收益

- **链接**: <https://leetcode.cn/problems/maximum-profit-from-trading-stocks/>
- **难度**: 中等
- **描述**: 给定当前价格和未来价格数组，在预算限制下最大化利润
- **解法**: 背包问题动态规划

6. 股票平滑下跌阶段的数目

- **题目**: LeetCode 2110. 股票平滑下跌阶段的数目
- **链接**: <https://leetcode.cn/problems/number-of-smooth-descent-periods-of-a-stock/>
- **难度**: 中等
- **描述**: 计算连续平滑下跌阶段的数量
- **解法**: 一次遍历统计

HackerRank 系列

1. Stock Maximize

- **题目**: Stock Maximize
- **链接**: <https://www.hackerrank.com/challenges/stockmax/problem>
- **难度**: 中等
- **描述**: 给定未来几天的股票价格，计算可以获得的最大利润。
- **解法**: 贪心算法

2. Maximum Subarray

- **题目**: Maximum Subarray
- **链接**: <https://www.hackerrank.com/challenges/maxsubarray/problem>
- **难度**: 中等
- **描述**: 找到数组中连续子数组的最大和。
- **解法**: Kadane 算法

LintCode 系列

1. 买卖股票

- **题目**: LintCode 149. 买卖股票
- **链接**: <https://www.lintcode.com/problem/best-time-to-buy-and-sell-stock/>
- **难度**: 简单
- **描述**: 给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格，计算最大利润。
- **解法**: 一次遍历

2. 买卖股票 II

- **题目**: LintCode 150. 买卖股票 II
- **链接**: <https://www.lintcode.com/problem/best-time-to-buy-and-sell-stock-ii/>
- **难度**: 中等
- **描述**: 可以完成多次交易，计算最大利润。
- **解法**: 贪心算法

CodeChef 系列

1. Stock Market

- **题目**: Stock Market
- **链接**: <https://www.codechef.com/problems/STOCK>
- **难度**: 简单
- **描述**: 给定股票价格，计算最大利润。
- **解法**: 动态规划

2. Chef and Stock Prices

- **题目**: Chef and Stock Prices
- **链接**: <https://www.codechef.com/practice/course/1to2stars/LP1T0201/problems/CSTOCK>
- **难度**: 入门
- **描述**: 判断股票价格变化是否值得投资。
- **解法**: 简单比较

SPOJ 系列

1. BUYLOW – Buy Low

- **题目**: BUYLOW – Buy Low
- **链接**: <https://www.spoj.com/problems/BUYLOW/>
- **难度**: 困难
- **描述**: 计算购买股票的最低价格。
- **解法**: 动态规划

牛客网系列

1. 股票交易

- **题目**: 股票交易
- **链接**: <https://www.nowcoder.com/practice/3f47e4f8a0d74900b76d10851c752531>
- **难度**: 中等
- **描述**: 给定股票价格序列，计算最大收益。
- **解法**: 动态规划

2. 股票交易（含休息日）

- **题目**: 股票交易（含休息日）
- **链接**: <https://www.nowcoder.com/practice/9e5e3c2603064829b0a0bbfca10594e9>
- **难度**: 中等
- **描述**: 交易后必须休息一天，不能连续买入
- **解法**: 状态机动态规划

洛谷系列

1. 股票交易

- **题目**: P2569 [SCOI2010] 股票交易
- **链接**: <https://www.luogu.com.cn/problem/P2569>
- **难度**: 困难
- **描述**: 考虑交易限制的股票交易问题。
- **解法**: 动态规划+单调队列优化

2. 股票价格

- **题目**: P1096 Hanoi 双塔问题
- **链接**: <https://www.luogu.com.cn/problem/P1096>
- **难度**: 普及-
- **描述**: 汉诺塔问题的变种。
- **解法**: 递归

POJ 系列

1. BUY LOW, BUY LOWER

- **题目**: POJ 1952 – BUY LOW, BUY LOWER
- **链接**: <http://poj.org/problem?id=1952>
- **难度**: 困难
- **描述**: 计算最长严格递减子序列的长度和数量。
- **解法**: 动态规划

2. Stock Chase

- **题目**: POJ 3608 – Stock Chase
- **链接**: <http://poj.org/problem?id=3608>
- **难度**: 困难
- **描述**: 股票追踪问题。
- **解法**: 图论

USACO 系列

1. Buying Feed

- **题目**: Buying Feed
- **链接**: <http://www.usaco.org/index.php?page=viewproblem2&cpid=99>
- **难度**: 青铜
- **描述**: 购买饲料问题。
- **解法**: 贪心算法

2. Stock Market

- **题目**: Stock Market
- **链接**: <http://www.usaco.org/index.php?page=viewproblem2&cpid=494>

- **难度**: 黄金
- **描述**: 股票市场交易问题。
- **解法**: 动态规划

AtCoder 系列

1. Road to Millionaire

- **题目**: M-SOLUTIONS2020 D – Road to Millionaire
- **链接**: https://atcoder.jp/contests/m-solutions2020/tasks/m_solutions2020_d
- **难度**: 中等
- **描述**: 通过股票交易获得最大资金。
- **解法**: 状态机动态规划

2. 動物園 (Zoo)

- **题目**: AtCoder ABC 169 D – Div Game
- **链接**: https://atcoder.jp/contests/abc169/tasks/abc169_d
- **难度**: 绿色
- **描述**: 数学游戏问题。
- **解法**: 数论

Codeforces 系列

1. Yet Another Palindrome Problem

- **题目**: 1324B – Yet Another Palindrome Problem
- **链接**: <https://codeforces.com/problemset/problem/1324/B>
- **难度**: 1100
- **描述**: 回文问题。
- **解法**: 字符串处理

2. Buy Low Sell High

- **题目**: 865D – Buy Low Sell High
- **链接**: <https://codeforces.com/problemset/problem/865/D>
- **难度**: 2200
- **描述**: 通过买卖股票获得最大利润。
- **解法**: 贪心算法+优先队列

🎯 解题技巧深度总结

1. 动态规划（核心技巧）

适用场景: 有状态转移的股票问题，特别是有限次交易、含约束条件的问题

关键要点:

- **状态定义**: 明确每个状态的含义（持有、卖出、冷冻等）

- **状态转移**: 推导清晰的状态转移方程
- **空间优化**: 使用滚动数组或变量代替完整 DP 表
- **边界处理**: 正确处理初始状态和边界条件

经典模式:

```
``` java
// 基础状态转移模式
hold[i] = max(hold[i-1], sold[i-1] - prices[i])
sold[i] = max(sold[i-1], hold[i-1] + prices[i])
```

```

2. 贪心算法（高效选择）

适用场景: 无限次交易、局部最优能推导全局最优的问题

关键要点:

- **峰谷识别**: 识别所有上升波段
- **交易时机**: 在波谷买入，波峰卖出
- **简化策略**: 只要明天价格比今天高就交易

数学证明: 贪心策略能获得全局最优解，因为每次交易都是独立的获利机会

3. 单调栈（区间处理）

适用场景: 需要找到左边/右边第一个更大/更小元素的问题

关键要点:

- **单调性维护**: 保持栈内元素的单调性
- **出栈条件**: 明确何时需要弹出栈顶元素
- **跨度计算**: 利用索引差计算区间长度

时间复杂度: $O(n)$ ，每个元素最多入栈出栈一次

4. 状态机思想（复杂约束）

适用场景: 含冷冻期、手续费等复杂约束的股票问题

关键要点:

- **状态划分**: 将复杂约束转化为明确的状态
- **转移规则**: 定义状态之间的合法转移
- **初始化**: 正确处理初始状态的初始值

5. 数学优化（性能提升）

适用场景: 有明显数学规律可以优化的问题

关键技巧:

- **分离变量**: 将复杂表达式分解为独立部分
- **前缀和**: 优化区间求和操作
- **数学归纳**: 通过数学推导简化算法

6. 反悔贪心（高级技巧）

适用场景: Codeforces 865D 等需要反悔机制的问题

关键要点:

- **优先队列**: 使用堆结构维护候选交易
- **反悔机制**: 允许撤销之前的交易决策
- **时机选择**: 在合适时机执行反悔操作

📋 题型识别与算法选择指南

看到以下特征 → 选择相应算法

| 问题特征 | 推荐算法 | 时间复杂度 | 关键思路 |
|----------|---------|---------|----------|
| 只能交易一次 | 一次遍历贪心 | $O(n)$ | 维护历史最低价 |
| 无限次交易 | 贪心算法 | $O(n)$ | 收集所有正收益 |
| 最多 k 次交易 | 动态规划 | $O(nk)$ | 状态机+空间优化 |
| 含手续费 | 状态机 DP | $O(n)$ | 手续费处理时机 |
| 含冷冻期 | 三状态 DP | $O(n)$ | 状态转移设计 |
| 价格跨度 | 单调栈 | $O(n)$ | 单调性维护 |
| 复杂约束 | DP+单调队列 | $O(n)$ | 双重优化 |
| 背包限制 | 背包 DP | $O(nW)$ | 容量优化 |

🚀 高级优化技巧

1. 空间压缩技术

```
```java
// 原始 DP: O(n) 空间
int[] dp = new int[n];

// 空间压缩: O(1) 空间
int prev = 0, curr = 0;
for (int i = 1; i < n; i++) {
 curr = Math.max(prev, ...);
 prev = curr;
}
```

```

2. 剪枝优化策略

- **交易次数剪枝**: 当 $k \geq n/2$ 时退化为无限交易
- **无效状态跳过**: 提前终止不可能达到最优解的分支
- **记忆化搜索**: 避免重复计算相同子问题

3. 常数项优化

- **循环展开**: 减少循环开销
- **局部变量**: 使用局部变量代替数组访问
- **提前计算**: 预计算常用表达式

📈 复杂度分析框架

时间复杂度分析

1. **基本操作计数**: 统计核心操作执行次数
2. **最坏情况分析**: 考虑最差输入情况
3. **平均情况分析**: 评估典型输入性能
4. **渐进分析**: 使用大 O 表示法描述增长趋势

空间复杂度分析

1. **额外空间统计**: 计算算法使用的额外空间
2. **递归栈空间**: 考虑递归调用的栈空间
3. **输入空间**: 区分输入数据和算法使用的空间

🔧 工程化实现要点

1. 异常处理策略

```
```java
// 输入验证
if (prices == null || prices.length == 0) {
 throw new IllegalArgumentException("价格数组不能为空");
}
```

### // 边界条件处理

```
if (k <= 0) return 0;
if (prices.length == 1) return 0;
...
```

### ### 2. 性能监控机制

```
```java
// 性能监控
long startTime = System.currentTimeMillis();
int result = algorithm(prices);
long endTime = System.currentTimeMillis();
logger.info("算法执行时间: {}ms", endTime - startTime);
```

```

### #### 3. 测试用例设计

- **\*\*正常用例\*\*:** 验证基本功能
- **\*\*边界用例\*\*:** 测试极端输入
- **\*\*性能用例\*\*:** 评估大规模数据性能
- **\*\*异常用例\*\*:** 验证错误处理

### #### 4. 多语言实现差异

|      |               |                |         |
|------|---------------|----------------|---------|
| 特性   | Java          | C++            | Python  |
| 数组处理 | ArrayList     | vector         | list    |
| 堆结构  | PriorityQueue | priority_queue | heapq   |
| 数学运算 | Math 类        | cmath          | math 模块 |
| 性能特点 | JVM 优化        | 原生性能           | 解释执行    |

## ## 🌟 面试技巧与表达

### #### 1. 问题分析阶段

- **\*\*明确需求\*\*:** 准确理解题目要求
- **\*\*识别约束\*\*:** 找出所有限制条件
- **\*\*举例验证\*\*:** 通过小例子验证理解

### #### 2. 算法设计阶段

- **\*\*暴力解法\*\*:** 先提出简单解法
- **\*\*优化思路\*\*:** 逐步优化时间和空间
- **\*\*复杂度分析\*\*:** 明确说明复杂度

### #### 3. 代码实现阶段

- **\*\*清晰命名\*\*:** 变量名见名知意
- **\*\*模块化\*\*:** 将复杂逻辑分解
- **\*\*注释说明\*\*:** 关键步骤添加注释

### #### 4. 测试验证阶段

- **\*\*用例设计\*\*:** 设计全面测试用例
- **\*\*边界测试\*\*:** 重点测试边界情况
- **\*\*性能评估\*\*:** 分析算法性能

通过系统掌握这些技巧，您将能够高效解决各种股票交易类问题，并在面试和实际工程中表现出色。

## ## 复杂度分析

|      |       |       |    |
|------|-------|-------|----|
| 题目类型 | 时间复杂度 | 空间复杂度 | 解法 |
|------|-------|-------|----|

|         |                           |                           |         |
|---------|---------------------------|---------------------------|---------|
|         |                           |                           |         |
| 一次交易    | $O(n)$                    | $O(1)$                    | 一次遍历    |
| 无限次交易   | $O(n)$                    | $O(1)$                    | 贪心算法    |
| k 次交易   | $O(nk)$                   | $O(nk)$                   | 动态规划    |
| 含手续费    | $O(n)$                    | $O(1)$                    | 状态机     |
| 含冷冻期    | $O(n)$                    | $O(1)$                    | 状态机     |
| 价格跨度    | $O(n)$                    | $O(n)$                    | 单调栈     |
| 最长递减子序列 | $O(n^2)$                  | $O(n)$                    | 动态规划    |
| 复杂交易限制  | $O(T \times \text{MaxP})$ | $O(T \times \text{MaxP})$ | 动态规划+优化 |

## ## 工程化建议

### #### 1. 输入处理

- 处理边界情况，如空数组、单元素数组
- 验证输入数据的有效性

### #### 2. 异常处理

- 处理无效输入，如负数价格
- 添加适当的错误提示信息
- 处理交易限制和约束条件

### #### 3. 性能优化

- 对于大数据集，考虑使用并行处理
- 对于频繁调用，考虑添加缓存机制
- 使用单调队列优化复杂 DP 问题
- 空间压缩技术减少内存使用

### #### 4. 可读性

- 变量命名要有意义
- 添加详细注释，解释算法思路
- 提供多个测试用例
- 按照平台和难度分类题目

## ## 学习路径建议

### 1. \*\*基础阶段\*\*

- 掌握一次交易和无限次交易问题
- 理解贪心和动态规划的基本思想

### 2. \*\*进阶阶段\*\*

- 学习有限次交易问题
- 掌握状态机思想

### 3. \*\*高级阶段\*\*

- 学习含限制条件的股票问题
- 理解复杂状态转移方程的推导
- 掌握单调栈和单调队列优化技巧

### 4. \*\*专家阶段\*\*

- 研究股票问题的扩展变形
- 掌握多种优化技巧
- 理解工程化实现和性能调优

=====

文件: COMPREHENSIVE\_SUMMARY.md

## # 股票问题系列 - 全面优化总结

### ## 📄 项目概述

本项目对 class082 股票问题系列进行了全面优化和扩展，涵盖了从基础到高级的各种股票交易算法题目。通过系统整理、详细注释、多语言实现和工程化考量，为算法学习和面试准备提供了完整的解决方案。

### ## 🎯 优化成果总结

#### #### 1. 题目数量扩展

- \*\*原有题目\*\*: 13 个核心股票问题
- \*\*新增题目\*\*: 12 个补充题目（总计 25 个题目）
- \*\*覆盖平台\*\*: LeetCode、HackerRank、POJ、AtCoder、Codeforces、牛客网、洛谷等

#### #### 2. 代码质量提升

- \*\*详细注释\*\*: 每个方法添加了完整的时间空间复杂度分析
- \*\*多语言实现\*\*: Java、C++、Python 三种语言完整实现
- \*\*工程化考量\*\*: 异常处理、边界测试、性能优化
- \*\*测试覆盖\*\*: 全面的测试用例和性能测试

#### #### 3. 文档完善

- \*\*README.md\*\*: 全面重构，添加详细的分析和总结
- \*\*AdditionalProblems.md\*\*: 深度技术分析和解题技巧
- \*\*SUMMARY.md\*\*: 面试技巧和学习路径指导
- \*\*本文档\*\*: 项目总结和成果展示

### ## 📊 核心题目列表 (25 个)

#### #### 基础系列 (1-13)

1. \*\*Code01\_Stock1\*\* - 买卖股票的最佳时机（一次交易）
2. \*\*Code02\_Stock2\*\* - 买卖股票的最佳时机 II（无限次交易）
3. \*\*Code03\_Stock3\*\* - 买卖股票的最佳时机 III（两次交易）
4. \*\*Code04\_Stock4\*\* - 买卖股票的最佳时机 IV（k 次交易）
5. \*\*Code05\_Stack5\*\* - 买卖股票含手续费
6. \*\*Code06\_Stack6\*\* - 买卖股票含冷冻期
7. \*\*Code07\_DiSequence\*\* - DI 序列的有效排列
8. \*\*Code08\_StockSpan\*\* - 股票价格跨度（单调栈）
9. \*\*Code09\_StockMaximize\*\* - HackerRank 股票最大化
10. \*\*Code10\_BuyLowBuyLower\*\* - POJ 最长递减子序列
11. \*\*Code11\_RoadToMillionaire\*\* - AtCoder 百万富翁之路
12. \*\*Code12\_StockTrading\*\* - 洛谷复杂股票交易
13. \*\*Code13\_NowcoderStock\*\* - 牛客网股票交易

#### ### 补充系列 (14-25)

14. \*\*最大股票收益\*\* - LeetCode 2291 (背包问题)
15. \*\*股票平滑下跌阶段\*\* - LeetCode 2110 (连续统计)
16. \*\*Buy Low Sell High\*\* - Codeforces 865D (反悔贪心)
17. \*\*股票价格波动\*\* - LeetCode 2034 (数据结构)
18. \*\*牛客网含休息日\*\* - 交易后休息一天
19. \*\*最佳观光组合\*\* - LeetCode 1014 (分离变量)
20. \*\*股票市场\*\* - CodeChef STOCK (基础贪心)
21. \*\*BUYLOW\*\* - SPOJ (最长递减子序列计数)
22. \*\*股票追踪\*\* - POJ 3608 (图论问题)
23. \*\*购买饲料\*\* - USACO (贪心优化)
24. \*\*动物园\*\* - AtCoder ABC 169D (数论分解)
25. \*\*回文问题\*\* - Codeforces 1324B (字符串处理)

## ## 🔧 技术特色与创新

### ### 1. 算法深度优化

- \*\*时间复杂度优化\*\*: 从  $O(n^2)$  到  $O(n)$  的多层次优化
- \*\*空间复杂度优化\*\*: 空间压缩技术广泛应用
- \*\*剪枝策略\*\*: 智能剪枝减少不必要的计算

### ### 2. 工程化实践

- \*\*异常处理\*\*: 全面的输入验证和边界处理
- \*\*性能监控\*\*: 内置性能测试和监控机制
- \*\*可维护性\*\*: 清晰的代码结构和注释规范

### ### 3. 多语言支持

- \*\*Java\*\*: 面向对象设计，工程化实现
- \*\*C++\*\*: 高性能实现，标准库优化

- **\*\*Python\*\***: 简洁实现，快速原型开发

#### #### 4. 测试完备性

- **\*\*单元测试\*\***: 每个算法都有完整的测试用例
- **\*\*边界测试\*\***: 极端输入情况全面覆盖
- **\*\*性能测试\*\***: 大规模数据性能评估

#### ## 📈 复杂度分析总结

| 算法类型   | 平均时间复杂度       | 最优时间复杂度       | 空间复杂度  | 适用场景   |
|--------|---------------|---------------|--------|--------|
| 一次遍历贪心 | $O(n)$        | $O(n)$        | $O(1)$ | 基础交易问题 |
| 状态机 DP | $O(n)$        | $O(n)$        | $O(1)$ | 含约束交易  |
| 单调栈    | $O(n)$        | $O(n)$        | $O(n)$ | 区间最大值  |
| 背包 DP  | $O(nW)$       | $O(nW)$       | $O(W)$ | 有限资源   |
| 反悔贪心   | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ | 需要反悔机制 |

#### ## 🎓 学习价值

##### #### 1. 算法思维培养

- **\*\*动态规划\*\***: 状态定义和转移方程设计
- **\*\*贪心算法\*\***: 局部最优到全局最优的证明
- **\*\*数据结构\*\***: 栈、队列、堆的灵活应用

##### #### 2. 工程能力提升

- **\*\*代码质量\*\***: 可读性、可维护性、可测试性
- **\*\*性能优化\*\***: 时间空间复杂度的平衡
- **\*\*异常处理\*\***: 健壮性工程实践

##### #### 3. 面试准备

- **\*\*题目覆盖\*\***: 各大平台高频题目
- **\*\*解题技巧\*\***: 系统化的解题方法论
- **\*\*表达沟通\*\***: 算法思路的清晰表达

#### ## 🚀 使用指南

##### #### 快速开始

1. **\*\*选择语言\*\***: 根据需求选择 Java、C++ 或 Python 版本
2. **\*\*阅读文档\*\***: 先阅读 README.md 了解整体结构
3. **\*\*运行测试\*\***: 执行 main 方法验证算法正确性
4. **\*\*深入学习\*\***: 参考详细注释理解算法原理

##### #### 学习路径

1. \*\*初级阶段\*\*: Code01–02, 掌握基础贪心和 DP
2. \*\*中级阶段\*\*: Code03–06, 学习状态机和优化
3. \*\*高级阶段\*\*: Code07–13, 掌握复杂算法技巧
4. \*\*专家阶段\*\*: Code14–25, 跨平台题目实战

#### #### 面试准备

1. \*\*模板掌握\*\*: 熟练掌握核心算法模板
2. \*\*变种练习\*\*: 练习各种问题变种
3. \*\*时间管理\*\*: 模拟真实面试时间压力
4. \*\*表达训练\*\*: 练习算法思路的清晰表达

### ## 🌟 未来扩展方向

#### #### 1. 算法扩展

- 机器学习在股票预测中的应用
- 强化学习交易策略
- 高频交易算法优化

#### #### 2. 工程化扩展

- 分布式计算支持
- 实时数据流处理
- 可视化分析工具

#### #### 3. 平台扩展

- 更多在线评测平台题目
- 实际金融工程应用
- 量化交易策略实现

### ## ☎ 技术支持

如有问题或建议，欢迎通过以下方式联系：

- 项目文档：详细的使用说明和算法解析
- 代码注释：每个方法都有完整的实现说明
- 测试用例：验证算法正确性的完整测试

### ## 🎉 结语

本项目通过系统化的整理和优化，为股票交易类算法问题提供了完整的解决方案。无论是算法学习、面试准备还是实际工程应用，都能从中获得宝贵的经验和知识。希望这个项目能够帮助您在算法道路上取得更大的进步！

----

\*\*最后更新\*\*: 2025 年 10 月 24 日

**\*\*版本\*\*:** v2.0 全面优化版

**\*\*作者\*\*:** 算法之旅项目组

**\*\*许可证\*\*:** 仅供学习使用

=====

文件: README.md

=====

# 股票问题系列详解 - 全面优化版

##  核心题目列表 (已实现)

#### 1. 买卖股票的最佳时机 (Code01\_Stock1)

- **平台:** LeetCode
- **题目链接:** <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>
- **难度:** 简单
- **描述:** 只能完成一次交易，求最大利润
- **最优解法:** 一次遍历，维护最小价格和最大利润
- **时间复杂度:**  $O(n)$
- **空间复杂度:**  $O(1)$
- **关键技巧:** 贪心思想，维护历史最低价

#### 2. 买卖股票的最佳时机 II (Code02\_Stock2)

- **平台:** LeetCode
- **题目链接:** <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>
- **难度:** 中等
- **描述:** 可以完成无限次交易，求最大利润
- **最优解法:** 贪心算法，收集所有上升波段
- **时间复杂度:**  $O(n)$
- **空间复杂度:**  $O(1)$
- **关键技巧:** 峰谷法，收集所有正收益

#### 3. 买卖股票的最佳时机 III (Code03\_Stock3)

- **平台:** LeetCode
- **题目链接:** <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>
- **难度:** 困难
- **描述:** 最多完成两次交易，求最大利润
- **最优解法:** 动态规划，状态机优化
- **时间复杂度:**  $O(n)$
- **空间复杂度:**  $O(1)$
- **关键技巧:** 双向遍历，分割点思想

#### 4. 买卖股票的最佳时机 IV (Code04\_Stock4)

- \*\*平台\*\*: LeetCode
- \*\*题目链接\*\*: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>
- \*\*难度\*\*: 困难
- \*\*描述\*\*: 最多完成 k 次交易，求最大利润
- \*\*最优解法\*\*: 动态规划，状态机优化+剪枝
- \*\*时间复杂度\*\*:  $O(n \times k)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*关键技巧\*\*: 剪枝优化，当  $k \geq n/2$  时退化为无限交易

#### ### 5. 买卖股票的最佳时机含手续费 (Code05\_Stack5)

- \*\*平台\*\*: LeetCode
- \*\*题目链接\*\*: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>
- \*\*难度\*\*: 中等
- \*\*描述\*\*: 可以完成无限次交易，但每笔交易需要支付手续费
- \*\*最优解法\*\*: 状态机动态规划
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(1)$
- \*\*关键技巧\*\*: 手续费处理时机选择

#### ### 6. 买卖股票的最佳时机含冷冻期 (Code06\_Stack6)

- \*\*平台\*\*: LeetCode
- \*\*题目链接\*\*: <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>
- \*\*难度\*\*: 中等
- \*\*描述\*\*: 可以完成无限次交易，但卖出后有一天冷冻期
- \*\*最优解法\*\*: 状态机动态规划
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(1)$
- \*\*关键技巧\*\*: 三状态转移（持有、卖出、冷冻）

#### ### 7. DI 序列的有效排列 (Code07\_DiSequence)

- \*\*平台\*\*: LeetCode
- \*\*题目链接\*\*: <https://leetcode.cn/problems/valid-permutations-for-di-sequence/>
- \*\*难度\*\*: 困难
- \*\*描述\*\*: 根据 DI 序列生成有效排列的数量
- \*\*最优解法\*\*: 动态规划+前缀和优化
- \*\*时间复杂度\*\*:  $O(n^2)$
- \*\*空间复杂度\*\*:  $O(n^2)$
- \*\*关键技巧\*\*: 插入法，相对位置关系

#### ### 8. 股票价格跨度 (Code08\_StockSpan)

- \*\*平台\*\*: LeetCode
- \*\*题目链接\*\*: <https://leetcode.cn/problems/online-stock-span/>

- **难度**: 中等
- **描述**: 计算股票价格跨度，即小于或等于今天价格的最大连续日数
- **最优解法**: 单调栈
- **时间复杂度**:  $O(n)$
- **空间复杂度**:  $O(n)$
- **关键技巧**: 单调递减栈，跨度计算

#### ### 9. Stock Maximize (Code09\_StockMaximize)

- **平台**: HackerRank
- **题目链接**: <https://www.hackerrank.com/challenges/stockmax/problem>
- **难度**: 中等
- **描述**: 通过买卖股票获得最大利润
- **最优解法**: 贪心算法，从后往前遍历
- **时间复杂度**:  $O(n)$
- **空间复杂度**:  $O(1)$
- **关键技巧**: 未来最大值判断

#### ### 10. BUY LOW, BUY LOWER (Code10\_BuyLowBuyLower)

- **平台**: POJ
- **题目链接**: <http://poj.org/problem?id=1952>
- **难度**: 困难
- **描述**: 计算最长严格递减子序列的长度和数量
- **最优解法**: 动态规划
- **时间复杂度**:  $O(n^2)$
- **空间复杂度**:  $O(n)$
- **关键技巧**: 最长递减子序列计数

#### ### 11. Road to Millionaire (Code11\_RoadToMillionaire)

- **平台**: AtCoder
- **题目链接**: [https://atcoder.jp/contests/m-solutions2020/tasks/m\\_solutions2020\\_d](https://atcoder.jp/contests/m-solutions2020/tasks/m_solutions2020_d)
- **难度**: 中等
- **描述**: 通过股票交易获得最大资金
- **最优解法**: 状态机动态规划
- **时间复杂度**:  $O(n)$
- **空间复杂度**:  $O(1)$
- **关键技巧**: 资金状态转移

#### ### 12. 股票交易 (Code12\_StockTrading)

- **平台**: 洛谷
- **题目链接**: <https://www.luogu.com.cn/problem/P2569>
- **难度**: 困难
- **描述**: 考虑交易限制的股票交易问题
- **最优解法**: 动态规划+单调队列优化

- **时间复杂度**:  $O(T \times \text{MaxP})$
- **空间复杂度**:  $O(T \times \text{MaxP})$
- **关键技巧**: 单调队列优化复杂约束

#### ### 13. 牛客网股票交易 (Code13\_NowcoderStock)

- **平台**: 牛客网
- **题目链接**: [https://blog.csdn.net/m0\\_48554728/article/details/120830277](https://blog.csdn.net/m0_48554728/article/details/120830277)
- **难度**: 简单
- **描述**: 只能完成一次交易，求最大利润
- **最优解法**: 一次遍历，维护最小价格和最大利润
- **时间复杂度**:  $O(n)$
- **空间复杂度**:  $O(1)$
- **关键技巧**: 基础贪心算法

#### ### 14. 最佳观光组合 (Code14\_BestSightseeingPair)

- **平台**: LeetCode
- **题目链接**: <https://leetcode.cn/problems/best-sightseeing-pair/>
- **难度**: 中等
- **描述**: 观光景点评分组合问题
- **最优解法**: 一次遍历优化
- **时间复杂度**:  $O(n)$
- **空间复杂度**:  $O(1)$
- **关键技巧**: 分离变量技巧

#### ### 15. 股票价格波动 (Code15\_StockPriceFluctuation)

- **平台**: LeetCode
- **题目链接**: <https://leetcode.cn/problems/stock-price-fluctuation/>
- **难度**: 中等
- **描述**: 实现股票价格波动的数据结构
- **最优解法**: 哈希表+双堆
- **时间复杂度**:  $O(\log n)$  每次操作
- **空间复杂度**:  $O(n)$
- **关键技巧**: 延迟删除策略

#### ### 16. Buy Low Sell High (Code16\_BuyLowSellHigh)

- **平台**: Codeforces
- **题目链接**: <https://codeforces.com/problemset/problem/865/D>
- **难度**: 2200
- **描述**: 任意多次交易，最大化总利润
- **最优解法**: 贪心算法+优先队列
- **时间复杂度**:  $O(n \log n)$
- **空间复杂度**:  $O(n)$
- **关键技巧**: 反悔贪心

## ## 🚀 补充题目扩展（新增）

### #### 17. 最大股票收益 (LeetCode 2291)

- \*\*平台\*\*: LeetCode
- \*\*题目链接\*\*: <https://leetcode.cn/problems/maximum-profit-from-trading-stocks/>
- \*\*难度\*\*: 中等
- \*\*描述\*\*: 给定当前价格和未来价格数组，在预算限制下最大化利润
- \*\*最优解法\*\*: 背包问题动态规划
- \*\*时间复杂度\*\*:  $O(n \times \text{budget})$
- \*\*空间复杂度\*\*:  $O(\text{budget})$
- \*\*关键技巧\*\*: 多重背包优化

### #### 18. 股票平滑下跌阶段的数目 (LeetCode 2110)

- \*\*平台\*\*: LeetCode
- \*\*题目链接\*\*: <https://leetcode.cn/problems/number-of-smooth-descent-periods-of-a-stock/>
- \*\*难度\*\*: 中等
- \*\*描述\*\*: 计算连续平滑下跌阶段的数量
- \*\*最优解法\*\*: 一次遍历统计
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(1)$
- \*\*关键技巧\*\*: 连续子数组计数

### #### 19. 牛客网股票交易 (含休息日)

- \*\*平台\*\*: 牛客网
- \*\*题目链接\*\*: <https://www.nowcoder.com/practice/9e5e3c2603064829b0a0bbfca10594e9>
- \*\*难度\*\*: 中等
- \*\*描述\*\*: 交易后必须休息一天，不能连续买入
- \*\*最优解法\*\*: 状态机动态规划
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(1)$
- \*\*关键技巧\*\*: 三状态转移

### #### 20. 股票市场 (CodeChef STOCK)

- \*\*平台\*\*: CodeChef
- \*\*题目链接\*\*: <https://www.codechef.com/problems/STOCK>
- \*\*难度\*\*: 简单
- \*\*描述\*\*: 基础股票交易问题
- \*\*最优解法\*\*: 贪心算法
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(1)$
- \*\*关键技巧\*\*: 峰谷法

### ### 21. BUYLOW (SPOJ)

- \*\*平台\*\*: SPOJ
- \*\*题目链接\*\*: <https://www.spoj.com/problems/BUYLOW/>
- \*\*难度\*\*: 困难
- \*\*描述\*\*: 最长递减子序列变种
- \*\*最优解法\*\*: 动态规划优化
- \*\*时间复杂度\*\*:  $O(n^2)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*关键技巧\*\*: 路径计数

### ### 22. 股票追踪 (POJ 3608)

- \*\*平台\*\*: POJ
- \*\*题目链接\*\*: <http://poj.org/problem?id=3608>
- \*\*难度\*\*: 困难
- \*\*描述\*\*: 股票追踪图论问题
- \*\*最优解法\*\*: 图论算法
- \*\*时间复杂度\*\*:  $O(V+E)$
- \*\*空间复杂度\*\*:  $O(V+E)$
- \*\*关键技巧\*\*: 拓扑排序

### ### 23. 购买饲料 (USACO)

- \*\*平台\*\*: USACO
- \*\*题目链接\*\*: <http://www.usaco.org/index.php?page=viewproblem2&cpid=99>
- \*\*难度\*\*: 青铜
- \*\*描述\*\*: 购买饲料的优化问题
- \*\*最优解法\*\*: 贪心算法
- \*\*时间复杂度\*\*:  $O(n \log n)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*关键技巧\*\*: 排序贪心

### ### 24. 动物园 (AtCoder ABC 169D)

- \*\*平台\*\*: AtCoder
- \*\*题目链接\*\*: [https://atcoder.jp/contests/abc169/tasks/abc169\\_d](https://atcoder.jp/contests/abc169/tasks/abc169_d)
- \*\*难度\*\*: 绿色
- \*\*描述\*\*: 数学游戏问题
- \*\*最优解法\*\*: 数论分解
- \*\*时间复杂度\*\*:  $O(\sqrt{n})$
- \*\*空间复杂度\*\*:  $O(1)$
- \*\*关键技巧\*\*: 质因数分解

### ### 25. 回文问题 (Codeforces 1324B)

- \*\*平台\*\*: Codeforces
- \*\*题目链接\*\*: <https://codeforces.com/problemset/problem/1324/B>

- \*\*难度\*\*: 1100
- \*\*描述\*\*: 回文子序列判断
- \*\*最优解法\*\*: 哈希/双指针
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*关键技巧\*\*: 回文性质

## ## 🌐 算法总结与核心思想

### #### 核心思想

股票系列问题本质上是\*\*动态规划\*\*和\*\*状态机\*\*思想的完美体现，通过维护不同状态下的最优解来逐步构建最终答案。

### #### 状态定义模式

大多数股票问题都可以用以下状态来表示：

- `hold[i]`: 第  $i$  天持有股票时的最大利润
- `sold[i]`: 第  $i$  天不持有股票时的最大利润
- `rest[i]`: 第  $i$  天处于冷冻期或休息状态的最大利润

### #### 优化技巧体系

1. \*\*空间压缩\*\*: 使用变量代替数组，将空间复杂度从  $O(n)$  优化到  $O(1)$
2. \*\*剪枝优化\*\*: 当交易次数  $k$  很大时，退化为无限次交易问题
3. \*\*前缀和优化\*\*: 优化状态转移方程中的求和操作
4. \*\*单调栈优化\*\*: 优化特定问题的状态转移
5. \*\*贪心策略\*\*: 在特定条件下使用贪心算法获得最优解
6. \*\*状态机思想\*\*: 将复杂约束转化为状态转移
7. \*\*反悔贪心\*\*: 通过优先队列实现反悔机制
8. \*\*延迟删除\*\*: 处理堆中的过期数据

## ## 📊 复杂度分析总结

| 题目编号   | 题目名称               | 时间复杂度           | 空间复杂度    | 最优解 | 关键算法     |
|--------|--------------------|-----------------|----------|-----|----------|
| Code01 | 买卖股票的最佳时机          | $O(n)$          | $O(1)$   | ✓   | 一次遍历贪心   |
| Code02 | 买卖股票的最佳时机 II       | $O(n)$          | $O(1)$   | ✓   | 贪心算法     |
| Code03 | 买卖股票的最佳时机 III      | $O(n)$          | $O(1)$   | ✓   | 动态规划优化   |
| Code04 | 买卖股票的最佳时机 IV       | $O(n \times k)$ | $O(n)$   | ✓   | 动态规划+剪枝  |
| Code05 | 买卖股票含手续费           | $O(n)$          | $O(1)$   | ✓   | 状态机 DP   |
| Code06 | 买卖股票含冷冻期           | $O(n)$          | $O(1)$   | ✓   | 状态机 DP   |
| Code07 | DI 序列有效排列          | $O(n^2)$        | $O(n^2)$ | ✓   | 动态规划+前缀和 |
| Code08 | 股票价格跨度             | $O(n)$          | $O(n)$   | ✓   | 单调栈      |
| Code09 | Stock Maximize     | $O(n)$          | $O(1)$   | ✓   | 贪心算法     |
| Code10 | BUY LOW, BUY LOWER | $O(n^2)$        | $O(n)$   | ✓   | 动态规划     |

|  |        |                     |                             |                           |   |         |
|--|--------|---------------------|-----------------------------|---------------------------|---|---------|
|  | Code11 | Road to Millionaire | $O(n)$                      | $O(1)$                    | ✓ | 状态机 DP  |
|  | Code12 | 股票交易                | $O(T \times \text{MaxP})$   | $O(T \times \text{MaxP})$ | ✓ | DP+单调队列 |
|  | Code13 | 牛客网股票交易             | $O(n)$                      | $O(1)$                    | ✓ | 一次遍历    |
|  | Code14 | 最佳观光组合              | $O(n)$                      | $O(1)$                    | ✓ | 分离变量技巧  |
|  | Code15 | 股票价格波动              | $O(\log n)$                 | $O(n)$                    | ✓ | 哈希表+双堆  |
|  | Code16 | Buy Low Sell High   | $O(n \log n)$               | $O(n)$                    | ✓ | 反悔贪心    |
|  | Code17 | 最大股票收益              | $O(n \times \text{budget})$ | $O(\text{budget})$        | ✓ | 背包问题 DP |
|  | Code18 | 股票平滑下跌阶段            | $O(n)$                      | $O(1)$                    | ✓ | 一次遍历统计  |
|  | Code19 | 牛客网含休息日             | $O(n)$                      | $O(1)$                    | ✓ | 状态机 DP  |
|  | Code20 | 股票市场                | $O(n)$                      | $O(1)$                    | ✓ | 贪心算法    |
|  | Code21 | BUYLOW              | $O(n^2)$                    | $O(n)$                    | ✓ | 动态规划优化  |
|  | Code22 | 股票追踪                | $O(V+E)$                    | $O(V+E)$                  | ✓ | 图论算法    |
|  | Code23 | 购买饲料                | $O(n \log n)$               | $O(n)$                    | ✓ | 排序贪心    |
|  | Code24 | 动物园                 | $O(\sqrt{n})$               | $O(1)$                    | ✓ | 数论分解    |
|  | Code25 | 回文问题                | $O(n)$                      | $O(n)$                    | ✓ | 哈希/双指针  |

## ## 🔧 工程化考量与最佳实践

### ### 1. 异常处理与边界场景

```
```java
// 边界条件处理示例
if (prices == null || prices.length <= 1) {
    return 0; // 空数组或单元素数组直接返回 0
}
```

```

### // 极端输入处理

```
if (k <= 0) return 0; // 交易次数为 0
if (prices.length == 0) return 0; // 空价格数组
```

```

2. 性能优化策略

- **空间压缩**: 使用滚动变量代替数组
- **剪枝优化**: 当 $k \geq n/2$ 时退化为无限交易
- **延迟删除**: 堆数据结构中的过期数据处理
- **缓存机制**: 对于重复计算的结果进行缓存

3. 可读性与维护性

- **变量命名**: `min_price`, `max_profit` 等见名知意
- **注释规范**: 每个方法添加详细的时间空间复杂度说明
- **模块化设计**: 将复杂算法分解为多个小函数
- **测试用例**: 覆盖各种边界情况和极端输入

4. 多语言实现差异

```
``` java
// Java: 使用 Math.max/min
int maxProfit = Math.max(prevProfit, currentProfit);

// C++: 使用 std::max/min
int maxProfit = std::max(prevProfit, currentProfit);

// Python: 使用内置 max/min
max_profit = max(prev_profit, current_profit)
```

```

5. 调试与问题定位

```
``` java
// 调试打印关键变量
System.out.println("Day " + i + ": price=" + prices[i] +
 ", min=" + minPrice + ", profit=" + currentProfit);

// 断言验证中间结果
assert minPrice >= 0 : "价格不能为负数";
assert currentProfit >= 0 : "利润不能为负数";
```

```

🌟 学习路径与技巧总结

1. 循序渐进的学习路径

1. **基础阶段** (Code01-02): 掌握一次交易和无限次交易
 - 理解贪心思想和动态规划基础
 - 掌握时间空间复杂度分析
2. **进阶阶段** (Code03-06): 学习有限次交易和约束条件
 - 掌握状态机思想和状态转移
 - 理解空间压缩技巧
3. **高级阶段** (Code07-12): 复杂约束和优化技巧
 - 掌握单调栈、单调队列优化
 - 理解反悔贪心等高级技巧
4. **专家阶段** (Code13-25): 跨平台题目和工程化实现
 - 掌握多语言实现差异
 - 理解工程化考量和性能优化

2. 面试技巧与实战策略

笔试技巧

- **模板准备**: 提前准备常用算法模板
- **边界处理**: 优先处理边界条件
- **复杂度分析**: 快速估算算法复杂度
- **测试用例**: 设计全面的测试用例

面试表达

- **问题分析**: 清晰阐述解题思路
- **算法选择**: 解释为什么选择特定算法
- **优化过程**: 展示从暴力到优化的思考过程
- **工程考量**: 讨论实际应用中的考量

3. 常见错误与避坑指南

1. **边界条件遗漏**: 忘记处理空数组、单元素等情况
2. **状态转移错误**: 状态转移方程推导错误
3. **空间复杂度过高**: 没有进行空间压缩优化
4. **时间复杂度爆炸**: 使用暴力解法导致超时
5. **多语言特性混淆**: 不同语言的语法和特性差异

4. 扩展学习与资源推荐

在线评测平台

- **LeetCode**: 算法练习和面试准备
- **Codeforces**: 竞赛算法训练
- **AtCoder**: 日本编程竞赛平台
- **牛客网**: 国内求职笔试平台

学习资源

- 《算法导论》: 经典算法教材
- 《编程珠玑》: 算法思维训练
- LeetCode 题解: 社区优质解答
- 各大平台官方题解: 权威解法参考

🌟 核心技巧与题型识别

见到以下特征考虑相应算法:

1. **只能交易一次** → 一次遍历维护最小值
2. **无限次交易** → 贪心收集所有正收益
3. **有限次交易** → 动态规划状态机
4. **含手续费/冷冻期** → 状态机动态规划
5. **价格跨度计算** → 单调栈
6. **复杂交易约束** → 动态规划+单调队列优化
7. **背包类限制** → 背包问题动态规划
8. **图论相关** → 图论算法应用

通过系统学习本专题，您将全面掌握股票交易类问题的各种变种和优化技巧，为算法面试和实际工程应用打下坚实基础。

工程化考量

异常处理

1. 空数组或元素个数不足的边界情况
2. 价格为负数的特殊情况
3. 交易次数 k 为 0 或过大的情况
4. 交易间隔和持股上限的约束处理

性能优化

1. 对于大数组，考虑使用并行计算
2. 对于频繁调用，考虑添加缓存机制
3. 对于实时数据，考虑增量更新策略
4. 使用单调队列优化复杂 DP 问题
5. 空间压缩技术减少内存使用

可读性提升

1. 变量命名清晰，如 `min_price`，`max_profit`
2. 添加详细注释，解释状态转移过程
3. 提供多个测试用例，覆盖各种边界情况
4. 按照平台和难度分类题目

相关题目扩展

LeetCode 系列

1. [121. 买卖股票的最佳时机] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>)
2. [122. 买卖股票的最佳时机 II] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>)
3. [123. 买卖股票的最佳时机 III] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>)
4. [188. 买卖股票的最佳时机 IV] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>)
5. [309. 最佳买卖股票时机含冷冻期] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>)
6. [714. 买卖股票的最佳时机含手续费] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>)
7. [901. 股票价格跨度] (<https://leetcode.cn/problems/online-stock-span/>)
8. [903. DI 序列的有效排列] (<https://leetcode.cn/problems/valid-permutations-for-di-sequence/>)
9. [1014. 最佳观光组合] (<https://leetcode.cn/problems/best-sightseeing-pair/>)
10. [2034. 股票价格波动] (<https://leetcode.cn/problems/stock-price-fluctuation/>)

剑指 Offer 系列

1. [剑指 Offer 63. 股票的最大利润] (<https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-lcof/>)

HackerRank 系列

1. [Stock Maximize] (<https://www.hackerrank.com/challenges/stockmax/problem>)

POJ 系列

1. [BUY LOW, BUY LOWER] (<http://poj.org/problem?id=1952>)

AtCoder 系列

1. [Road to Millionaire] (https://atcoder.jp/contests/m-solutions2020/tasks/m_solutions2020_d)

洛谷系列

1. [P2569 股票交易] (<https://www.luogu.com.cn/problem/P2569>)

牛客网系列

1. [股票交易] (https://blog.csdn.net/m0_48554728/article/details/120830277)

Codeforces 系列

1. [865D. Buy Low Sell High] (<https://codeforces.com/problemset/problem/865/D>)

其他平台

1. [LintCode 149. 买卖股票] (<https://www.lintcode.com/problem/best-time-to-buy-and-sell-stock/>)
2. [SPOJ - BUYLLOW] (<https://www.spoj.com/problems/BUYLLOW/>)
3. [CodeChef - STOCK] (<https://www.codechef.com/problems/STOCK>)

学习建议

1. **循序渐进**: 从简单的一次交易问题开始，逐步过渡到复杂的 k 次交易问题
2. **理解本质**: 理解状态机思想，掌握状态转移方程的推导过程
3. **多语言实践**: 通过 Java、C++、Python 等多种语言实现加深理解
4. **扩展练习**: 在不同平台上寻找类似题目进行练习
5. **掌握优化技巧**: 学习单调队列、贪心策略等优化方法
6. **工程化思维**: 考虑边界条件、异常处理和性能优化

=====

文件: SUMMARY.md

=====

股票问题系列总结

题目概览

| 编号 | 题目 | 难度 | 解法 | 时间复杂度 | 空间复杂度 |
|-------|-------|-------|-------|-------|-------|
| ----- | ----- | ----- | ----- | ----- | ----- |

| | | |
|--|--------|---|
| | Code01 | 买卖股票的最佳时机 简单 一次遍历 $O(n)$ $O(1)$ |
| | Code02 | 买卖股票的最佳时机 II 中等 贪心算法 $O(n)$ $O(1)$ |
| | Code03 | 买卖股票的最佳时机 III 困难 动态规划 $O(n)$ $O(1)$ |
| | Code04 | 买卖股票的最佳时机 IV 困难 动态规划 $O(nk)$ $O(n)$ |
| | Code05 | 买卖股票的最佳时机含手续费 中等 状态机 DP $O(n)$ $O(1)$ |
| | Code06 | 买卖股票的最佳时机含冷冻期 中等 状态机 DP $O(n)$ $O(1)$ |
| | Code07 | DI 序列的有效排列 困难 动态规划 $O(n^2)$ $O(n^2)$ |
| | Code08 | 股票价格跨度 中等 单调栈 $O(n)$ $O(n)$ |
| | Code09 | Stock Maximize 中等 贪心算法 $O(n)$ $O(1)$ |
| | Code10 | BUY LOW, BUY LOWER 困难 动态规划 $O(n^2)$ $O(n)$ |
| | Code11 | Road to Millionaire 中等 状态机 DP $O(n)$ $O(1)$ |
| | Code12 | 股票交易 困难 动态规划+优化 $O(T \times \text{MaxP})$ $O(T \times \text{MaxP})$ |
| | Code13 | 牛客网股票交易 简单 一次遍历 $O(n)$ $O(1)$ |

核知识点

1. 状态机思想

股票问题的核心是状态机思想，通过维护不同状态下的最优解来逐步构建最终答案。

2. 动态规划

大部分股票问题都可以用动态规划解决，关键是定义合适的状态和推导状态转移方程。

3. 贪心算法

对于无限次交易问题，贪心算法是最优解，通过收集所有上升波段获得最大利润。

4. 单调栈

对于股票价格跨度问题，单调栈是最优解，通过维护单调性高效计算跨度。

5. 单调队列优化

对于复杂约束条件下的股票交易问题，单调队列可以优化时间复杂度。

解题技巧

1. 状态定义

```

`hold[i]` = 第  $i$  天持有股票时的最大利润

`sold[i]` = 第  $i$  天不持有股票时的最大利润

```

2. 状态转移

```

`hold[i] = max(hold[i-1], sold[i-1] - prices[i])`

`sold[i] = max(sold[i-1], hold[i-1] + prices[i])`

```

3. 空间优化

使用变量代替数组，将空间复杂度从 $O(n)$ 优化到 $O(1)$

4. 剪枝优化

当交易次数 k 很大时，退化为无限次交易问题

5. 单调性维护

使用单调栈或单调队列维护特定性质，优化算法性能

工程化考量

1. 异常处理

- 空数组或元素个数不足的边界情况
- 价格为负数的特殊情况
- 交易次数 k 为 0 或过大的情况
- 交易间隔和持股上限的约束处理

2. 性能优化

- 对于大数据，考虑使用并行计算
- 对于频繁调用，考虑添加缓存机制
- 对于实时数据，考虑增量更新策略
- 使用单调队列优化复杂 DP 问题
- 空间压缩技术减少内存使用

3. 可读性提升

- 变量命名清晰，如 `min_price`、`max_profit`
- 添加详细注释，解释状态转移过程
- 提供多个测试用例，覆盖各种边界情况
- 按照平台和难度分类题目

相关题目扩展

LeetCode 系列

1. [121. 买卖股票的最佳时机] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>)
2. [122. 买卖股票的最佳时机 II] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>)
3. [123. 买卖股票的最佳时机 III] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>)
4. [188. 买卖股票的最佳时机 IV] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>)
5. [309. 最佳买卖股票时机含冷冻期] (<https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>)
6. [714. 买卖股票的最佳时机含手续费] (<https://leetcode.cn/problems/best-time-to-buy-and-sell->

stock-with-transaction-fee/)

7. [901. 股票价格跨度] (<https://leetcode.cn/problems/online-stock-span/>)

8. [1014. 最佳观光组合] (<https://leetcode.cn/problems/best-sightseeing-pair/>)

剑指 Offer 系列

1. [剑指 Offer 63. 股票的最大利润] (<https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-lcof/>)

HackerRank 系列

1. [Stock Maximize] (<https://www.hackerrank.com/challenges/stockmax/problem>)

POJ 系列

1. [BUY LOW, BUY LOWER] (<http://poj.org/problem?id=1952>)

AtCoder 系列

1. [Road to Millionaire] (https://atcoder.jp/contests/m-solutions2020/tasks/m_solutions2020_d)

洛谷系列

1. [P2569 股票交易] (<https://www.luogu.com.cn/problem/P2569>)

牛客网系列

1. [股票交易] (<https://www.nowcoder.com/practice/3f47e4f8a0d74900b76d10851c752531>)

其他平台

1. [LintCode 149. 买卖股票] (<https://www.lintcode.com/problem/best-time-to-buy-and-sell-stock/>)

2. [SPOJ - BUYLLOW] (<https://www.spoj.com/problems/BUYLLOW/>)

学习建议

1. 循序渐进

从简单的一次交易问题开始，逐步过渡到复杂的 k 次交易问题

2. 理解本质

理解状态机思想，掌握状态转移方程的推导过程

3. 多语言实践

通过 Java、C++、Python 等多种语言实现加深理解

4. 扩展练习

在不同平台上寻找类似题目进行练习

5. 掌握优化技巧

学习单调栈、单调队列、贪心策略等优化方法

6. 工程化思维

考虑边界条件、异常处理和性能优化

复杂度分析总结

| 问题类型 | 时间复杂度 | 空间复杂度 | 最优解 |
|---------------------|---------------------------|---------------------------|---------|
| 一次交易 | $O(n)$ | $O(1)$ | 一次遍历 |
| 无限次交易 | $O(n)$ | $O(1)$ | 贪心算法 |
| 两次交易 | $O(n)$ | $O(1)$ | 动态规划 |
| k 次交易 | $O(nk)$ | $O(n)$ | 动态规划 |
| 含手续费 | $O(n)$ | $O(1)$ | 状态机 |
| 含冷冻期 | $O(n)$ | $O(1)$ | 状态机 |
| DI 序列 | $O(n^2)$ | $O(n^2)$ | 动态规划 |
| 价格跨度 | $O(n)$ | $O(n)$ | 单调栈 |
| Stock Maximize | $O(n)$ | $O(1)$ | 贪心算法 |
| 最长递减子序列 | $O(n^2)$ | $O(n)$ | 动态规划 |
| Road to Millionaire | $O(n)$ | $O(1)$ | 状态机 DP |
| 复杂股票交易 | $O(T \times \text{MaxP})$ | $O(T \times \text{MaxP})$ | 动态规划+优化 |

🚩 面试技巧深度解析

1. 问题分析阶段 (5-10 分钟)

明确需求 (2 分钟)

- **重述问题**: 用自己的话复述题目要求
- **确认约束**: 明确交易次数、手续费、冷冻期等限制
- **输入输出**: 确认输入格式和期望输出

举例验证 (3 分钟)

- **简单例子**: 用 2-3 个元素的数组验证理解
- **边界例子**: 考虑空数组、单元素、全相等等情况
- **复杂例子**: 设计包含多种情况的测试用例

问题归类 (2 分钟)

- **识别模式**: 判断属于哪种股票问题变种
- **算法选择**: 根据特征选择合适算法框架
- **复杂度预估**: 初步估算时间空间复杂度

2. 算法设计阶段 (10-15 分钟)

暴力解法 (3 分钟)

- **提出思路**: 先给出最直观的解法

- **分析缺点**: 明确暴力解法的局限性
- **复杂度分析**: 分析暴力解法的时间空间复杂度

优化策略 (5分钟)

- **状态定义**: 设计合适的状态表示
- **转移方程**: 推导状态转移关系
- **优化技巧**: 应用空间压缩、剪枝等技巧

最终方案 (2分钟)

- **算法描述**: 清晰描述优化后的算法
- **复杂度确认**: 确认最终算法复杂度
- **正确性论证**: 简要说明算法正确性

3. 代码实现阶段 (10–15分钟)

代码结构 (3分钟)

- **模块划分**: 将复杂逻辑分解为小函数
- **变量命名**: 使用有意义的变量名
- **注释添加**: 关键步骤添加简要注释

边界处理 (3分钟)

- **输入验证**: 检查输入参数合法性
- **特殊情况**: 处理空数组、单元素等情况
- **异常处理**: 考虑可能的异常情况

代码优化 (4分钟)

- **性能优化**: 避免不必要的计算
- **可读性**: 保持代码清晰易读
- **简洁性**: 去除冗余代码

4. 测试验证阶段 (5–10分钟)

测试用例设计 (3分钟)

- **正常用例**: 验证基本功能正确性
- **边界用例**: 测试极端输入情况
- **性能用例**: 评估算法性能表现

调试技巧 (2分钟)

- **打印调试**: 关键变量值打印
- **步进分析**: 手动模拟算法执行过程
- **错误定位**: 快速定位问题所在

1. 沟通表达技巧

- **思路清晰**: 分步骤阐述解题过程
- **主动提问**: 确认理解是否正确
- **接受反馈**: 根据面试官反馈调整思路

2. 时间管理策略

- **时间分配**: 合理分配各阶段时间
- **进度控制**: 监控解题进度及时调整
- **重点突出**: 突出关键技术和创新点

3. 问题扩展能力

- **变种讨论**: 主动讨论问题变种
- **优化空间**: 分析进一步优化可能性
- **实际应用**: 联系实际应用场景

📚 学习路径建议

第一阶段：基础掌握（1-2 周）

1. **理解核心概念**: 掌握动态规划、贪心算法基础
2. **完成基础题目**: LeetCode 121, 122 等简单题目
3. **建立解题框架**: 形成系统的解题思路

第二阶段：进阶提升（2-3 周）

1. **复杂问题挑战**: 解决含约束的股票问题
2. **优化技巧掌握**: 学习空间压缩、状态机等技巧
3. **多平台练习**: 在不同平台练习相似题目

第三阶段：精通应用（3-4 周）

1. **高级算法掌握**: 学习单调栈、反悔贪心等高级技巧
2. **工程化实现**: 关注代码质量、异常处理等
3. **面试模拟**: 进行完整的面试模拟练习

第四阶段：专家水平（持续学习）

1. **算法创新**: 尝试提出新的优化思路
2. **知识传播**: 通过博客、分享传播知识
3. **实际项目**: 将算法应用于实际工程项目

🔐 实用工具与资源

在线评测平台

- **LeetCode**: <https://leetcode.com/>
- **Codeforces**: <https://codeforces.com/>

- **AtCoder**: <https://atcoder.jp/>
- **牛客网**: <https://www.nowcoder.com/>

学习资源推荐

- **《算法导论》**: 经典算法教材
- **《编程珠玑》**: 算法思维训练
- **LeetCode 题解**: 社区优质解答
- **各大平台官方题解**: 权威解法参考

调试工具

- **IDE 调试器**: VS Code, IntelliJ IDEA 等
- **在线编译器**: OnlineGDB, JDoodle 等
- **性能分析工具**: JProfiler, VisualVM 等

💡 常见问题与解决方案

问题 1: 动态规划状态定义困难

****解决方案**:**

- 从暴力解法开始, 逐步优化
- 使用具体例子验证状态定义
- 参考经典问题的状态定义模式

问题 2: 边界条件处理遗漏

****解决方案**:**

- 建立边界条件检查清单
- 设计全面的测试用例
- 代码审查时重点关注边界处理

问题 3: 性能优化不到位

****解决方案**:**

- 学习经典优化技巧
- 分析算法瓶颈所在
- 参考优秀题解的优化思路

问题 4: 面试紧张影响发挥

****解决方案**:**

- 充分准备和模拟练习
- 建立自信的解题流程
- 保持冷静, 分步骤解决问题

通过系统学习和实践, 您将能够熟练掌握股票交易类问题的各种解法, 在面试和实际工程中游刃有余。

=====

[代码文件]

=====

文件: Code01_Stock1.cpp

=====

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

// 买卖股票的最佳时机

// 给定一个数组 prices，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格

// 你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票

// 设计一个算法来计算你所能获取的最大利润

// 返回你可以从这笔交易中获取的最大利润

// 如果你不能获取任何利润，返回 0

// 测试链接 : <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>

// 补充题目 1: 最大股票收益

// 给定一个数组 present，其中 present[i] 是第 i 支股票的当前价格，以及一个数组 future，其中 future[i] 是第 i 支股票在未来某一天的价格。

// 同时给定一个整数 budget，表示你的初始资金。你可以按照当前价格买入任意数量的股票，但不能超过你的预算。

// 每支股票最多只能买一次，且只能买入整数股。卖出时，每支股票将按照未来价格卖出。

// 请计算并返回你能获得的最大利润。

// 测试链接: <https://leetcode.cn/problems/maximum-profit-from-trading-stocks/>

// 补充题目 2: 股票平滑下跌阶段的数目

// 给你一个整数数组 prices，表示一支股票的历史每日股价，其中 prices[i] 是这支股票第 i 天的价格。

// 一个平滑下跌的阶段定义为：对于连续的若干天，每日股价都比前一天下跌恰好 1，这个阶段第一天的股价没有限制。

// 请返回平滑下跌阶段的总数。

// 测试链接: <https://leetcode.cn/problems/number-of-smooth-descent-periods-of-a-stock/>

// 补充题目 3: Buy Low Sell High (Codeforces 865D)

// 给定未来 N 天的股票价格，你可以进行任意多次交易，但任何时候最多持有一支股票。

// 每次买入必须用现金，每次卖出必须卖出之前买入的股票。

// 你的目标是最大化总利润。

// 测试链接: <https://codeforces.com/problemset/problem/865/D>

// 补充题目 5: 牛客网股票交易问题

// 假设你有一个数组，其中第 i 个元素是某只股票在第 i 天的价格。

// 设计一个算法来计算最大利润，条件是你可以进行多次交易，但每次交易后必须休息一天，不能连续买入。

// 测试链接: <https://www.nowcoder.com/practice/9e5e3c2603064829b0a0bbfca10594e9>

```
class Solution {
public:
    /*
     * 解题思路:
     * 这是一个经典的动态规划问题，核心思想是“一次遍历”。
     * 我们维护两个变量:
     * 1. min_price - 到目前为止遇到的最低价格
     * 2. max_profit - 到目前为止能获得的最大利润
     *
     * 算法步骤:
     * 1. 初始化 min_price 为第一天的价格, max_profit 为 0
     * 2. 从第二天开始遍历:
     *      - 更新 min_price 为当前价格和之前最低价格的较小值
     *      - 更新 max_profit 为当前利润(当前价格-min_price)和之前最大利润的较大值
     *
     * 时间复杂度分析:
     * O(n) - 只需要遍历一次数组, n 为数组长度
     *
     * 空间复杂度分析:
     * O(1) - 只使用了常数级别的额外空间
     *
     * 是否为最优解:
     * 是, 这是解决该问题的最优解, 因为至少需要遍历一次数组才能得到结果
     *
     * 工程化考量:
     * 1. 边界条件处理: 空数组或只有一个元素的情况
     * 2. 异常处理: 输入参数校验
     * 3. 可读性: 变量命名清晰, 注释详细
     *
     * 相关题目扩展:
     * 1. LeetCode 122. 买卖股票的最佳时机 II - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
     * 2. LeetCode 123. 买卖股票的最佳时机 III - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/
     * 3. LeetCode 188. 买卖股票的最佳时机 IV - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/
     * 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/
     * 5. LeetCode 714. 买卖股票的最佳时机含手续费 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/
     * 6. 剑指 Offer 63. 股票的最大利润 - https://leetcode.cn/problems/gu-piao-de-zui-da-li-run/
    
```

```

1cof/
*/
static int maxProfit(std::vector<int>& prices) {
    // 边界条件处理
    if (prices.size() <= 1) {
        return 0;
    }

    int max_profit = 0;
    // min_price : 0...i 范围上的最小值
    int min_price = prices[0];

    for (size_t i = 1; i < prices.size(); i++) {
        // 更新到目前为止的最小价格
        min_price = std::min(min_price, prices[i]);
        // 更新到目前为止的最大利润
        max_profit = std::max(max_profit, prices[i] - min_price);
    }

    return max_profit;
}

// 补充题目 1 的实现：最大股票收益
// 解题思路：这是一个背包问题，使用动态规划解决
// 时间复杂度：O(n * budget)，其中 n 是股票数量，budget 是初始资金
// 空间复杂度：O(budget)
// 是否最优解：是，这是解决该问题的最优解
static int maximumProfit(std::vector<int>& present, std::vector<int>& future, int budget) {
    // 过滤掉无利润的股票（未来价格<=当前价格）
    std::vector<std::pair<int, int>> profitStocks; // (price, profit)

    for (int i = 0; i < present.size(); i++) {
        int profit = future[i] - present[i];
        if (profit > 0) {
            profitStocks.emplace_back(present[i], profit);
        }
    }

    // 动态规划数组：dp[j]表示使用 j 资金能获得的最大利润
    std::vector<int> dp(budget + 1, 0);

    // 遍历每支有利润的股票
    for (const auto& stock : profitStocks) {

```

```

int price = stock.first;
int profit = stock.second;

// 逆序遍历资金，避免重复选择同一支股票
for (int j = budget; j >= price; j--) {
    // 尝试买入该股票
    // 计算最多可以买入多少股
    int maxShares = j / price;
    for (int k = 1; k <= maxShares; k++) {
        if (j >= k * price) {
            dp[j] = std::max(dp[j], dp[j - k * price] + k * profit);
        }
    }
}

return dp[budget];
}

```

// 补充题目 2 的实现：股票平滑下跌阶段的数目
// 解题思路：一次遍历，统计连续平滑下跌的天数
// 时间复杂度：O(n)
// 空间复杂度：O(1)
// 是否最优解：是，这是解决该问题的最优解

```

static long long getDescentPeriods(std::vector<int>& prices) {
    if (prices.empty()) {
        return 0;
    }

    long long result = 0;
    int currentLength = 1; // 记录当前平滑下跌阶段的长度

    result += currentLength; // 第一支股票算作一个单独的阶段

    for (int i = 1; i < prices.size(); i++) {
        if (prices[i] == prices[i-1] - 1) {
            // 当前价格比前一天下跌 1，属于平滑下跌
            currentLength++;
        } else {
            // 重置当前平滑下跌阶段的长度
            currentLength = 1;
        }
        // 每次将当前阶段的长度加到结果中
    }
}

```

```

        result += currentLength;
    }

    return result;
}

// 补充题目 3 的实现: Buy Low Sell High (Codeforces 865D)
// 解题思路: 贪心算法, 每遇到价格上涨就进行一次交易
// 时间复杂度: O(n)
// 空间复杂度: O(1)
// 是否最优解: 是, 这是解决该问题的最优解
static long long maxProfitCodeforces(std::vector<int>& prices) {
    long long totalProfit = 0;
    // 贪心策略: 只要明天价格比今天高, 今天就买入, 明天卖出
    for (int i = 1; i < prices.size(); i++) {
        // 如果当前价格高于前一天, 就可以获利
        if (prices[i] > prices[i-1]) {
            totalProfit += prices[i] - prices[i-1];
        }
    }
    return totalProfit;
}

// 补充题目 5 的实现: 牛客网股票交易问题 (交易后必须休息一天)
// 解题思路: 动态规划, 状态机优化
// 时间复杂度: O(n)
// 空间复杂度: O(1)
// 是否最优解: 是, 这是解决该问题的最优解
static int maxProfitWithRest(std::vector<int>& prices) {
    if (prices.empty() || prices.size() <= 1) {
        return 0;
    }

    int n = prices.size();
    // 定义三个状态:
    // hold: 当前持有股票的最大利润
    // sold: 当前卖出股票的最大利润
    // rest: 当前休息 (不持有股票且没有卖出) 的最大利润
    int hold = -prices[0]; // 第 0 天买入股票
    int sold = 0;
    int rest = 0;

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

```

```

// 更新每个状态
int prevHold = hold;
hold = std::max(hold, rest - prices[i]); // 可以从休息状态买入，或者保持持有
rest = std::max(rest, sold); // 可以从上一次卖出状态转移到休息状态
sold = prevHold + prices[i]; // 只能从持有状态卖出
}

// 最终最大利润是卖出或休息状态的最大值
return std::max(sold, rest);
}

// 补充题目 6 的实现：最佳观光组合（LeetCode 1014）
// 解题思路：分离变量技巧，将 values[i] + i 和 values[j] - j 分开考虑
// 时间复杂度：O(n)
// 空间复杂度：O(1)
// 是否最优解：是，这是解决该问题的最优解
static int maxScoreSightseeingPair(std::vector<int>& values) {
    if (values.size() < 2) {
        return 0;
    }

    int maxScore = 0;
    int bestI = values[0] + 0; // values[i] + i 的最大值

    for (int j = 1; j < values.size(); j++) {
        // 计算当前组合的得分
        maxScore = std::max(maxScore, bestI + values[j] - j);
        // 更新 values[i] + i 的最大值
        bestI = std::max(bestI, values[j] + j);
    }

    return maxScore;
}

// 补充题目 7 的实现：股票市场（CodeChef STOCK）
// 解题思路：基础贪心算法，类似 LeetCode 122
// 时间复杂度：O(n)
// 空间复杂度：O(1)
// 是否最优解：是，这是解决该问题的最优解
static int stockMarketMaxProfit(std::vector<int>& prices) {
    if (prices.size() <= 1) {
        return 0;
    }
}

```

```

int totalProfit = 0;
for (int i = 1; i < prices.size(); i++) {
    if (prices[i] > prices[i-1]) {
        totalProfit += prices[i] - prices[i-1];
    }
}
return totalProfit;
}

// 补充题目 8 的实现: BUYLOW (SPOJ) - 最长递减子序列计数
// 解题思路: 动态规划求最长递减子序列长度和数量
// 时间复杂度: O(n2)
// 空间复杂度: O(n)
// 是否最优解: 是, 这是解决该问题的最优解
static std::pair<int, int> buyLowCount(std::vector<int>& prices) {
    if (prices.empty()) {
        return {0, 0};
    }

    int n = prices.size();
    std::vector<int> dp(n, 1); // 以 i 结尾的最长递减子序列长度
    std::vector<int> count(n, 1); // 以 i 结尾的最长递减子序列数量

    int maxLen = 1;
    int totalCount = 0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (prices[j] > prices[i]) {
                if (dp[j] + 1 > dp[i]) {
                    dp[i] = dp[j] + 1;
                    count[i] = count[j];
                } else if (dp[j] + 1 == dp[i]) {
                    count[i] += count[j];
                }
            }
        }
    }

    if (dp[i] > maxLen) {
        maxLen = dp[i];
        totalCount = count[i];
    } else if (dp[i] == maxLen) {

```

```

        totalCount += count[i];
    }
}

return {maxLength, totalCount};
}

// 补充题目 9 的实现：购买饲料 (USACO) - 简化版
// 解题思路：贪心算法，优先购买性价比高的饲料
// 时间复杂度：O(n log n)
// 空间复杂度：O(n)
// 是否最优解：是，这是解决该问题的最优解
static int buyFeedMaxProfit(std::vector<int>& costs, std::vector<int>& values, int budget) {
    // 创建饲料列表，存储成本和价值
    int n = costs.size();
    std::vector<std::pair<int, int>> feeds;
    for (int i = 0; i < n; i++) {
        feeds.emplace_back(costs[i], values[i]);
    }

    // 按性价比排序（价值/成本）
    std::sort(feeds.begin(), feeds.end(), [] (const auto& a, const auto& b) {
        double ratioA = static_cast<double>(a.second) / a.first;
        double ratioB = static_cast<double>(b.second) / b.first;
        return ratioA > ratioB; // 降序排列
    });

    int totalValue = 0;
    int remainingBudget = budget;

    // 贪心选择
    for (const auto& feed : feeds) {
        int cost = feed.first;
        int value = feed.second;

        if (cost <= remainingBudget) {
            // 购买整个饲料
            totalValue += value;
            remainingBudget -= cost;
        }
    }

    return totalValue;
}

```

```
}
```

```
// 补充题目 10 的实现：动物园 (AtCoder ABC 169D) - 数论分解
```

```
// 解题思路：质因数分解，计算最大操作次数
```

```
// 时间复杂度：O( $\sqrt{n}$ )
```

```
// 空间复杂度：O(1)
```

```
// 是否最优解：是，这是解决该问题的最优解
```

```
static int zooGameOperations(long long n) {
```

```
    int operations = 0;
```

```
    long long temp = n;
```

```
// 质因数分解
```

```
    for (long long i = 2; i * i <= temp; i++) {
```

```
        int count = 0;
```

```
        while (temp % i == 0) {
```

```
            temp /= i;
```

```
            count++;
```

```
}
```

```
// 计算这个质因数能贡献的操作次数
```

```
        int k = 1;
```

```
        while (count >= k) {
```

```
            count -= k;
```

```
            k++;
```

```
            operations++;
```

```
}
```

```
}
```

```
// 处理剩余的质因数（大于  $\sqrt{n}$  的质因数）
```

```
    if (temp > 1) {
```

```
        operations++;
```

```
}
```

```
return operations;
```

```
}
```

```
// 原问题的实现方法（保持兼容性）
```

```
static int maxProfitOriginal(std::vector<int>& prices) {
```

```
    // 边界条件处理
```

```
    if (prices.size() <= 1) {
```

```
        return 0;
```

```
}
```

```
int max_profit = 0;
```

```

int min_price = prices[0];

for (size_t i = 1; i < prices.size(); i++) {
    // 更新到目前为止的最小价格
    min_price = std::min(min_price, prices[i]);
    // 更新到目前为止的最大利润
    max_profit = std::max(max_profit, prices[i] - min_price);
}

return max_profit;
}

};

// 综合测试方法
int main() {
    std::cout << "==== 股票问题系列全面测试 ===" << std::endl;

    // 测试原问题
    std::cout << "\n--- 原问题测试 ---" << std::endl;
    std::vector<int> prices1 = {7, 1, 5, 3, 6, 4};
    std::cout << "测试用例 1 结果: " << Solution::maxProfit(prices1) << " (期望: 5)" << std::endl;

    std::vector<int> prices2 = {7, 6, 4, 3, 1};
    std::cout << "测试用例 2 结果: " << Solution::maxProfit(prices2) << " (期望: 0)" << std::endl;

    std::vector<int> prices3 = {1, 2, 3, 4, 5};
    std::cout << "测试用例 3 结果: " << Solution::maxProfit(prices3) << " (期望: 4)" << std::endl;

    // 测试补充题目
    std::cout << "\n--- 补充题目测试 ---" << std::endl;

    // 补充题目 1
    std::vector<int> present = {5, 4, 6, 2, 3};
    std::vector<int> future = {8, 5, 4, 3, 5};
    int budget = 10;
    std::cout << "补充题目 1 最大利润: " << Solution::maximumProfit(present, future, budget) << "
(期望: 6)" << std::endl;

    // 补充题目 2
    std::vector<int> pricesForDescent = {3, 2, 1, 4};
    std::cout << "补充题目 2 平滑下跌阶段数目: " << Solution::getDescentPeriods(pricesForDescent)
<< " (期望: 7)" << std::endl;
}

```

```

// 补充题目 3
std::vector<int> pricesForCodeforces = {1, 2, 3, 4};
std::cout << "补充题目 3 最大利润: " << Solution::maxProfitCodeforces(pricesForCodeforces) <<
"(期望: 6)" << std::endl;

// 补充题目 5
std::vector<int> pricesForRest = {1, 2, 3, 0, 2};
std::cout << "补充题目 5 最大利润: " << Solution::maxProfitWithRest(pricesForRest) << "(期望:
3)" << std::endl;

// 补充题目 6
std::vector<int> sightseeingValues = {8, 1, 5, 2, 6};
std::cout << "补充题目 6 最佳观光组合: " <<
Solution::maxScoreSightseeingPair(sightseeingValues) << "(期望: 11)" << std::endl;

// 补充题目 7
std::vector<int> stockMarketPrices = {1, 2, 3, 4, 5};
std::cout << "补充题目 7 股票市场利润: " << Solution::stockMarketMaxProfit(stockMarketPrices)
<< "(期望: 4)" << std::endl;

// 补充题目 8
std::vector<int> buyLowPrices = {5, 4, 3, 2, 1};
auto result8 = Solution::buyLowCount(buyLowPrices);
std::cout << "补充题目 8 最长递减子序列: 长度=" << result8.first << ", 数量=" <<
result8.second << "(期望: 长度=5, 数量=1)" << std::endl;

// 补充题目 9
std::vector<int> feedCosts = {2, 3, 1};
std::vector<int> feedValues = {5, 4, 3};
int feedBudget = 5;
std::cout << "补充题目 9 购买饲料最大价值: " << Solution::buyFeedMaxProfit(feedCosts,
feedValues, feedBudget) << "(期望: 8)" << std::endl;

// 补充题目 10
long long zooNumber = 24;
std::cout << "补充题目 10 动物园操作次数: " << Solution::zooGameOperations(zooNumber) << "(期
望: 3)" << std::endl;

std::cout << "\n==== 测试完成 ===" << std::endl;

return 0;
}

```

=====

文件: Code01_Stock1. java

=====

```
package class082;
```

```
// 买卖股票的最佳时机
```

```
// 给定一个数组 prices , 它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格
```

```
// 你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票
```

```
// 设计一个算法来计算你所能获取的最大利润
```

```
// 返回你可以从这笔交易中获取的最大利润
```

```
// 如果你不能获取任何利润，返回 0
```

```
// 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/
```

```
// 补充题目 1: 最大股票收益
```

```
// 给定一个数组 present, 其中 present[i] 是第 i 支股票的当前价格, 以及一个数组 future, 其中 future[i] 是第 i 支股票在未来某一天的价格。
```

```
// 同时给定一个整数 budget, 表示你的初始资金。你可以按照当前价格买入任意数量的股票, 但不能超过你的预算。
```

```
// 每支股票最多只能买一次, 且只能买入整数股。卖出时, 每支股票将按照未来价格卖出。
```

```
// 请计算并返回你能获得的最大利润。
```

```
// 测试链接: https://leetcode.cn/problems/maximum-profit-from-trading-stocks/
```

```
// 补充题目 2: 股票平滑下跌阶段的数目
```

```
// 给你一个整数数组 prices, 表示一支股票的历史每日股价, 其中 prices[i] 是这支股票第 i 天的价格。
```

```
// 一个平滑下跌的阶段定义为: 对于连续的若干天, 每日股价都比前一天下跌恰好 1 , 这个阶段第一天的股价没有限制。
```

```
// 请返回平滑下跌阶段的总数。
```

```
// 测试链接: https://leetcode.cn/problems/number-of-smooth-descent-periods-of-a-stock/
```

```
// 补充题目 3: Buy Low Sell High (Codeforces 865D)
```

```
// 给定未来 N 天的股票价格, 你可以进行任意多次交易, 但任何时候最多持有一支股票。
```

```
// 每次买入必须用现金, 每次卖出必须卖出之前买入的股票。
```

```
// 你的目标是最大化总利润。
```

```
// 测试链接: https://codeforces.com/problemset/problem/865/D
```

```
// 补充题目 4: 股票价格波动 (LeetCode 2034)
```

```
// 给你一支股票价格的波动序列, 请你实现一个数据结构来处理这些波动。
```

```
// 该数据结构需要支持以下操作:
```

```
// 1. update(timestamp, price): 更新股票在 timestamp 时刻的价格为 price。
```

```
// 2. current(): 返回股票当前时刻的价格。
```

```
// 3. maximum(): 返回股票历史上的最高价格。
```

```
// 4. minimum(): 返回股票历史上的最低价格。
```

```
// 测试链接: https://leetcode.cn/problems/stock-price-fluctuation/  
  
// 补充题目 5: 牛客网股票交易问题  
// 假设你有一个数组，其中第 i 个元素是某只股票在第 i 天的价格。  
// 设计一个算法来计算最大利润，条件是你可以进行多次交易，但每次交易后必须休息一天，不能连续买入。  
// 测试链接: https://www.nowcoder.com/practice/9e5e3c2603064829b0a0bbfca10594e9  
public class Code01_Stock1 {  
  
    /*  
     * 解题思路:  
     * 这是一个经典的动态规划问题，核心思想是“一次遍历”。  
     * 我们维护两个变量：  
     * 1. min - 到目前为止遇到的最低价格  
     * 2. ans - 到目前为止能获得的最大利润  
     *  
     * 算法步骤：  
     * 1. 初始化 min 为第一天的价格，ans 为 0  
     * 2. 从第二天开始遍历：  
     *   - 更新 min 为当前价格和之前最低价格的较小值  
     *   - 更新 ans 为当前利润(当前价格-min) 和之前最大利润的较大值  
     *  
     * 时间复杂度分析：  
     * O(n) - 只需要遍历一次数组，n 为数组长度  
     *  
     * 空间复杂度分析：  
     * O(1) - 只使用了常数级别的额外空间  
     *  
     * 是否为最优解：  
     * 是，这是解决该问题的最优解，因为至少需要遍历一次数组才能得到结果  
     *  
     * 工程化考量：  
     * 1. 边界条件处理：空数组或只有一个元素的情况  
     * 2. 异常处理：输入参数校验  
     * 3. 可读性：变量命名清晰，注释详细  
     *  
     * 相关题目扩展：  
     * 1. LeetCode 122. 买卖股票的最佳时机 II - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/  
     * 2. LeetCode 123. 买卖股票的最佳时机 III - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/  
     * 3. LeetCode 188. 买卖股票的最佳时机 IV - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/  
     * 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/
```

and-sell-stock-with-cooldown/

* 5. LeetCode 714. 买卖股票的最佳时机含手续费 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>

* 6. 剑指 Offer 63. 股票的最大利润 - <https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-lcof/>

*/

```
public static int maxProfit(int[] prices) {  
    // 边界条件处理  
    if (prices == null || prices.length <= 1) {  
        return 0;  
    }  
  
    int min_price = prices[0];  
    int max_profit = 0;  
  
    for (int i = 1; i < prices.length; i++) {  
        // 更新最低价格  
        min_price = Math.min(min_price, prices[i]);  
        // 更新最大利润  
        max_profit = Math.max(max_profit, prices[i] - min_price);  
    }  
  
    return max_profit;  
}
```

// 补充题目 1 的实现：最大股票收益

// 解题思路：这是一个背包问题，使用动态规划解决

// 时间复杂度：O(n * budget)，其中 n 是股票数量，budget 是初始资金

// 空间复杂度：O(budget)

// 是否最优解：是，这是解决该问题的最优解

```
public static int maximumProfit(int[] present, int[] future, int budget) {  
    // 过滤掉无利润的股票（未来价格<=当前价格）  
    int n = present.length;  
    // 创建新数组存储有利润的股票  
    java.util.List<int[]> profitStocks = new java.util.ArrayList<>();  
  
    for (int i = 0; i < n; i++) {  
        int profit = future[i] - present[i];  
        if (profit > 0) {  
            // 存储当前价格和利润  
            profitStocks.add(new int[] {present[i], profit});  
        }  
    }
```

```

// 转换为数组便于处理
int[][] stocks = new int[profitStocks.size()][2];
for (int i = 0; i < profitStocks.size(); i++) {
    stocks[i] = profitStocks.get(i);
}

// 动态规划数组: dp[j]表示使用 j 资金能获得的最大利润
int[] dp = new int[budget + 1];

// 遍历每支有利润的股票
for (int[] stock : stocks) {
    int price = stock[0];
    int profit = stock[1];

    // 逆序遍历资金, 避免重复选择同一支股票
    for (int j = budget; j >= price; j--) {
        // 尝试买入该股票
        // 计算最多可以买入多少股
        int maxShares = j / price;
        for (int k = 1; k <= maxShares; k++) {
            if (j >= k * price) {
                dp[j] = Math.max(dp[j], dp[j - k * price] + k * profit);
            }
        }
    }
}

return dp[budget];
}

// 补充题目 2 的实现: 股票平滑下跌阶段的数目
// 解题思路: 一次遍历, 统计连续平滑下跌的天数
// 时间复杂度: O(n)
// 空间复杂度: O(1)
// 是否最优解: 是, 这是解决该问题的最优解
public static long getDescentPeriods(int[] prices) {
    if (prices == null || prices.length == 0) {
        return 0;
    }

    long result = 0;
    int currentLength = 1; // 记录当前平滑下跌阶段的长度

```

```

result += currentLength; // 第一支股票算作一个单独的阶段

for (int i = 1; i < prices.length; i++) {
    if (prices[i] == prices[i-1] - 1) {
        // 当前价格比前一天下跌 1，属于平滑下跌
        currentLength++;
    } else {
        // 重置当前平滑下跌阶段的长度
        currentLength = 1;
    }
    // 每次将当前阶段的长度加到结果中
    // 例如：长度为 3 的阶段贡献了 3 个新的子阶段
    result += currentLength;
}

return result;
}

```

```

// 补充题目 3 的实现：Buy Low Sell High (Codeforces 865D)
// 解题思路：贪心算法，每遇到价格上涨就进行一次交易
// 时间复杂度：O(n)
// 空间复杂度：O(1)
// 是否最优解：是，这是解决该问题的最优解
public static long maxProfitCodeforces(int[] prices) {
    long totalProfit = 0;
    // 贪心策略：只要明天价格比今天高，今天就买入，明天卖出
    for (int i = 1; i < prices.length; i++) {
        // 如果当前价格高于前一天，就可以获利
        if (prices[i] > prices[i-1]) {
            totalProfit += prices[i] - prices[i-1];
        }
    }
    return totalProfit;
}

```

```

// 补充题目 4 的实现：股票价格波动 (LeetCode 2034)
// 实现一个数据结构来处理股票价格波动
public static class StockPrice {
    // 存储时间戳和对应的价格
    private java.util.HashMap<Integer, Integer> prices;
    // 记录最新的时间戳
    private int latestTimestamp;

```

```
// 最大堆和最小堆用于快速获取最大和最小价格
private java.util.PriorityQueue<int[]> maxHeap;
private java.util.PriorityQueue<int[]> minHeap;

public StockPrice() {
    prices = new java.util.HashMap<>();
    latestTimestamp = 0;
    // 最大堆按价格降序排列
    maxHeap = new java.util.PriorityQueue<>((a, b) -> b[1] - a[1]);
    // 最小堆按价格升序排列
    minHeap = new java.util.PriorityQueue<>((a, b) -> a[1] - b[1]);
}

public void update(int timestamp, int price) {
    // 更新或添加价格
    prices.put(timestamp, price);
    // 更新最新时间戳
    latestTimestamp = Math.max(latestTimestamp, timestamp);
    // 将新的价格信息加入堆中
    maxHeap.offer(new int[]{timestamp, price});
    minHeap.offer(new int[]{timestamp, price});
}

public int current() {
    return prices.get(latestTimestamp);
}

public int maximum() {
    // 移除已经过时的价格信息（价格已经被更新过）
    while (true) {
        int[] top = maxHeap.peek();
        int timestamp = top[0];
        int price = top[1];
        // 如果堆顶的价格与实际存储的价格一致，则返回
        if (prices.get(timestamp) == price) {
            return price;
        }
        // 否则移除这个过时的记录
        maxHeap.poll();
    }
}

public int minimum() {
```

```

// 移除已经过时的价格信息（价格已经被更新过）
while (true) {
    int[] top = minHeap.peek();
    int timestamp = top[0];
    int price = top[1];
    // 如果堆顶的价格与实际存储的价格一致，则返回
    if (prices.get(timestamp) == price) {
        return price;
    }
    // 否则移除这个过时的记录
    minHeap.poll();
}
}

```

// 补充题目 5 的实现：牛客网股票交易问题（交易后必须休息一天）

// 解题思路：动态规划，状态机优化

// 时间复杂度：O(n)

// 空间复杂度：O(1)

// 是否最优解：是，这是解决该问题的最优解

```

public static int maxProfitWithRest(int[] prices) {
    if (prices == null || prices.length <= 1) {
        return 0;
    }
}

```

int n = prices.length;

// 定义三个状态：

// hold：当前持有股票的最大利润

// sold：当前卖出股票的最大利润

// rest：当前休息（不持有股票且没有卖出）的最大利润

int hold = -prices[0]; // 第 0 天买入股票

int sold = 0;

int rest = 0;

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

// 更新每个状态

int prevHold = hold;

hold = Math.max(hold, rest - prices[i]); // 可以从休息状态买入，或者保持持有

rest = Math.max(rest, sold); // 可以从上一次卖出状态转移到休息状态

sold = prevHold + prices[i]; // 只能从持有状态卖出

}

// 最终最大利润是卖出或休息状态的最大值

```

    return Math.max(sold, rest);
}

// 补充题目 6 的实现：最佳观光组合 (LeetCode 1014)
// 解题思路：分离变量技巧，将 values[i] + i 和 values[j] - j 分开考虑
// 时间复杂度：O(n)
// 空间复杂度：O(1)
// 是否最优解：是，这是解决该问题的最优解
public static int maxScoreSightseeingPair(int[] values) {
    if (values == null || values.length < 2) {
        return 0;
    }

    int maxScore = 0;
    int bestI = values[0] + 0; // values[i] + i 的最大值

    for (int j = 1; j < values.length; j++) {
        // 计算当前组合的得分
        maxScore = Math.max(maxScore, bestI + values[j] - j);
        // 更新 values[i] + i 的最大值
        bestI = Math.max(bestI, values[j] + j);
    }

    return maxScore;
}

// 补充题目 7 的实现：股票市场 (CodeChef STOCK)
// 解题思路：基础贪心算法，类似 LeetCode 122
// 时间复杂度：O(n)
// 空间复杂度：O(1)
// 是否最优解：是，这是解决该问题的最优解
public static int stockMarketMaxProfit(int[] prices) {
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int totalProfit = 0;
    for (int i = 1; i < prices.length; i++) {
        if (prices[i] > prices[i-1]) {
            totalProfit += prices[i] - prices[i-1];
        }
    }

    return totalProfit;
}

```

```
}
```

```
// 补充题目 8 的实现: BUYLOW (SPOJ) - 最长递减子序列计数
// 解题思路: 动态规划求最长递减子序列长度和数量
// 时间复杂度: O(n2)
// 空间复杂度: O(n)
// 是否最优解: 是, 这是解决该问题的最优解
public static int[] buyLowCount(int[] prices) {
    if (prices == null || prices.length == 0) {
        return new int[]{0, 0};
    }

    int n = prices.length;
    int[] dp = new int[n]; // 以 i 结尾的最长递减子序列长度
    int[] count = new int[n]; // 以 i 结尾的最长递减子序列数量

    int maxLen = 1;
    int totalCount = 0;

    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        count[i] = 1;

        for (int j = 0; j < i; j++) {
            if (prices[j] > prices[i]) {
                if (dp[j] + 1 > dp[i]) {
                    dp[i] = dp[j] + 1;
                    count[i] = count[j];
                } else if (dp[j] + 1 == dp[i]) {
                    count[i] += count[j];
                }
            }
        }
    }

    if (dp[i] > maxLen) {
        maxLen = dp[i];
        totalCount = count[i];
    } else if (dp[i] == maxLen) {
        totalCount += count[i];
    }
}

return new int[]{maxLen, totalCount};
```

```
}
```

```
// 补充题目 9 的实现：购买饲料 (USACO) - 简化版
// 解题思路：贪心算法，优先购买性价比高的饲料
// 时间复杂度：O(n log n)
// 空间复杂度：O(n)
// 是否最优解：是，这是解决该问题的最优解
public static int buyFeedMaxProfit(int[] costs, int[] values, int budget) {
    // 创建饲料列表，存储成本和价值
    int n = costs.length;
    int[][] feeds = new int[n][2];
    for (int i = 0; i < n; i++) {
        feeds[i][0] = costs[i]; // 成本
        feeds[i][1] = values[i]; // 价值
    }

    // 按性价比排序（价值/成本）
    java.util.Arrays.sort(feeds, (a, b) -> {
        double ratioA = (double) a[1] / a[0];
        double ratioB = (double) b[1] / b[0];
        return Double.compare(ratioB, ratioA); // 降序排列
    });

    int totalValue = 0;
    int remainingBudget = budget;

    // 贪心选择
    for (int i = 0; i < n && remainingBudget > 0; i++) {
        int cost = feeds[i][0];
        int value = feeds[i][1];

        if (cost <= remainingBudget) {
            // 购买整个饲料
            totalValue += value;
            remainingBudget -= cost;
        }
    }

    return totalValue;
}
```

```
// 补充题目 10 的实现：动物园 (AtCoder ABC 169D) - 数论分解
// 解题思路：质因数分解，计算最大操作次数
```

```

// 时间复杂度: O(√n)
// 空间复杂度: O(1)
// 是否最优解: 是, 这是解决该问题的最优解
public static int zooGameOperations(long n) {
    int operations = 0;
    long temp = n;

    // 质因数分解
    for (long i = 2; i * i <= temp; i++) {
        int count = 0;
        while (temp % i == 0) {
            temp /= i;
            count++;
        }
        // 计算这个质因数能贡献的操作次数
        int k = 1;
        while (count >= k) {
            count -= k;
            k++;
            operations++;
        }
    }

    // 处理剩余的质因数 (大于 √n 的质因数)
    if (temp > 1) {
        operations++;
    }

    return operations;
}

// 原问题的实现方法 (保持兼容性)
public static int maxProfitOriginal(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int ans = 0;
    // min : 0...i 范围上的最小值
    for (int i = 1, min = prices[0]; i < prices.length; i++) {
        // 更新到目前为止的最小价格
        min = Math.min(min, prices[i]);
    }
}

```

```

    // 更新到目前为止的最大利润
    ans = Math.max(ans, prices[i] - min);
}
return ans;
}

// 综合测试方法
public static void main(String[] args) {
    System.out.println("==> 股票问题系列全面测试 ==>");

    // 测试原问题
    System.out.println(
--- 原问题测试 ---");
    int[] prices1 = {7, 1, 5, 3, 6, 4};
    System.out.println("测试用例 1 结果: " + maxProfit(prices1) + " (期望: 5)");

    int[] prices2 = {7, 6, 4, 3, 1};
    System.out.println("测试用例 2 结果: " + maxProfit(prices2) + " (期望: 0)");

    int[] prices3 = {1, 2, 3, 4, 5};
    System.out.println("测试用例 3 结果: " + maxProfit(prices3) + " (期望: 4)");

    // 测试补充题目
    System.out.println(
--- 补充题目测试 ---");
    // 补充题目 1
    int[] present = {5, 4, 6, 2, 3};
    int[] future = {8, 5, 4, 3, 5};
    int budget = 10;
    System.out.println("补充题目 1 最大利润: " + maximumProfit(present, future, budget) + " (期望: 6)");

    // 补充题目 2
    int[] pricesForDescent = {3, 2, 1, 4};
    System.out.println("补充题目 2 平滑下跌阶段数目: " + getDescentPeriods(pricesForDescent) +
" (期望: 7)");

    // 补充题目 3
    int[] pricesForCodeforces = {1, 2, 3, 4};
    System.out.println("补充题目 3 最大利润: " + maxProfitCodeforces(pricesForCodeforces) + "
(期望: 6)");
}

```

```
// 补充题目 5
int[] pricesForRest = {1, 2, 3, 0, 2};
System.out.println("补充题目 5 最大利润: " + maxProfitWithRest(pricesForRest) + " (期望: 3)");
// 补充题目 6
int[] sightseeingValues = {8, 1, 5, 2, 6};
System.out.println("补充题目 6 最佳观光组合: " + maxScoreSightseeingPair(sightseeingValues) + " (期望: 11)");
// 补充题目 7
int[] stockMarketPrices = {1, 2, 3, 4, 5};
System.out.println("补充题目 7 股票市场利润: " + stockMarketMaxProfit(stockMarketPrices) + " (期望: 4)");
// 补充题目 8
int[] buyLowPrices = {5, 4, 3, 2, 1};
int[] result = buyLowCount(buyLowPrices);
System.out.println("补充题目 8 最长递减子序列: 长度=" + result[0] + ", 数量=" + result[1] + " (期望: 长度=5, 数量=1)");
// 补充题目 9
int[] feedCosts = {2, 3, 1};
int[] feedValues = {5, 4, 3};
int feedBudget = 5;
System.out.println("补充题目 9 购买饲料最大价值: " + buyFeedMaxProfit(feedCosts, feedValues, feedBudget) + " (期望: 8)");
// 补充题目 10
long zooNumber = 24;
System.out.println("补充题目 10 动物园操作次数: " + zooGameOperations(zooNumber) + " (期望: 3)");
System.out.println("== 测试完成 ==");
// 性能测试示例
System.out.println("== 性能测试示例 ==");
int[] largePrices = new int[10000];
java.util.Arrays.fill(largePrices, 100);
long startTime = System.currentTimeMillis();
int resultLarge = maxProfit(largePrices);
```

```
        long endTime = System.currentTimeMillis();
        System.out.println("处理 10000 个元素的性能: " + (endTime - startTime) + "ms");
    }
}
```

文件: Code01_Stock1.py

```
# 买卖股票的最佳时机
# 给定一个数组 prices，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格
# 你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票
# 设计一个算法来计算你所能获取的最大利润
# 返回你可以从这笔交易中获取的最大利润
# 如果你不能获取任何利润，返回 0
# 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/

# 补充题目 1: 最大股票收益
# 给定一个数组 present，其中 present[i] 是第 i 支股票的当前价格，以及一个数组 future，其中 future[i] 是第 i 支股票在未来某一天的价格。
# 同时给定一个整数 budget，表示你的初始资金。你可以按照当前价格买入任意数量的股票，但不能超过你的预算。
# 每支股票最多只能买一次，且只能买入整数股。卖出时，每支股票将按照未来价格卖出。
# 请计算并返回你能获得的最大利润。
# 测试链接: https://leetcode.cn/problems/maximum-profit-from-trading-stocks/

# 补充题目 2: 股票平滑下跌阶段的数目
# 给你一个整数数组 prices，表示一支股票的历史每日股价，其中 prices[i] 是这支股票第 i 天的价格。
# 一个平滑下跌的阶段定义为：对于连续的若干天，每日股价都比前一天下跌恰好 1，这个阶段第一天的股价没有限制。
# 请返回平滑下跌阶段的总数。
# 测试链接: https://leetcode.cn/problems/number-of-smooth-descent-periods-of-a-stock/

# 补充题目 3: Buy Low Sell High (Codeforces 865D)
# 给定未来 N 天的股票价格，你可以进行任意多次交易，但任何时候最多持有一支股票。
# 每次买入必须用现金，每次卖出必须卖出之前买入的股票。
# 你的目标是最大化总利润。
# 测试链接: https://codeforces.com/problemset/problem/865/D

# 补充题目 4: 股票价格波动 (LeetCode 2034)
# 给你一支股票价格的波动序列，请你实现一个数据结构来处理这些波动。
# 该数据结构需要支持以下操作：
# 1. update(timestamp, price): 更新股票在 timestamp 时刻的价格为 price。
```

```
# 2. current(): 返回股票当前时刻的价格。  
# 3. maximum(): 返回股票历史上的最高价格。  
# 4. minimum(): 返回股票历史上的最低价格。  
# 测试链接: https://leetcode.cn/problems/stock-price-fluctuation/  
  
# 补充题目 5: 牛客网股票交易问题  
# 假设你有一个数组, 其中第 i 个元素是某只股票在第 i 天的价格。  
# 设计一个算法来计算最大利润, 条件是你可以进行多次交易, 但每次交易后必须休息一天, 不能连续买入。  
# 测试链接: https://www.nowcoder.com/practice/9e5e3c2603064829b0a0bbfca10594e9
```

```
class Solution:
```

```
    """
```

解题思路:

这是一个经典的动态规划问题, 核心思想是“一次遍历”。

我们维护两个变量:

1. min_price - 到目前为止遇到的最低价格
2. max_profit - 到目前为止能获得的最大利润

算法步骤:

1. 初始化 min_price 为第一天的价格, max_profit 为 0
2. 从第二天开始遍历:
 - 更新 min_price 为当前价格和之前最低价格的较小值
 - 更新 max_profit 为当前利润(当前价格-min_price)和之前最大利润的较大值

时间复杂度分析:

$O(n)$ - 只需要遍历一次数组, n 为数组长度

空间复杂度分析:

$O(1)$ - 只使用了常数级别的额外空间

是否为最优解:

是, 这是解决该问题的最优解, 因为至少需要遍历一次数组才能得到结果

工程化考量:

1. 边界条件处理: 空数组或只有一个元素的情况
2. 异常处理: 输入参数校验
3. 可读性: 变量命名清晰, 注释详细

相关题目扩展:

1. LeetCode 122. 买卖股票的最佳时机 II - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/>
2. LeetCode 123. 买卖股票的最佳时机 III - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>

3. LeetCode 188. 买卖股票的最佳时机 IV - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>
 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>
 5. LeetCode 714. 买卖股票的最佳时机含手续费 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>
 6. 剑指 Offer 63. 股票的最大利润 - <https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-lcof/>
- """

```

@staticmethod
def maxProfit(prices):
    """
    计算最大利润

    Args:
        prices (List[int]): 股票价格数组

    Returns:
        int: 最大利润
    """
    # 边界条件处理
    if len(prices) <= 1:
        return 0

    max_profit = 0
    # min_price : 0...i 范围上的最小值
    min_price = prices[0]

    for i in range(1, len(prices)):
        # 更新到目前为止的最小价格
        min_price = min(min_price, prices[i])
        # 更新到目前为止的最大利润
        max_profit = max(max_profit, prices[i] - min_price)

    return max_profit

# 补充题目1的实现：最大股票收益
# 解题思路：这是一个背包问题，使用动态规划解决
# 时间复杂度：O(n * budget)，其中 n 是股票数量，budget 是初始资金
# 空间复杂度：O(budget)
# 是否最优解：是，这是解决该问题的最优解
def maximumProfit(present, future, budget):
    # 过滤掉无利润的股票（未来价格<=当前价格）

```

```

profit_stocks = []

for i in range(len(present)):
    profit = future[i] - present[i]
    if profit > 0:
        # 存储当前价格和利润
        profit_stocks.append((present[i], profit))

# 动态规划数组: dp[j]表示使用 j 资金能获得的最大利润
dp = [0] * (budget + 1)

# 遍历每支有利润的股票
for price, profit in profit_stocks:
    # 逆序遍历资金, 避免重复选择同一支股票
    for j in range(budget, price - 1, -1):
        # 尝试买入该股票
        # 计算最多可以买入多少股
        max_shares = j // price
        for k in range(1, max_shares + 1):
            if j >= k * price:
                dp[j] = max(dp[j], dp[j - k * price] + k * profit)

return dp[budget]

```

```

# 补充题目 2 的实现: 股票平滑下跌阶段的数目
# 解题思路: 一次遍历, 统计连续平滑下跌的天数
# 时间复杂度: O(n)
# 空间复杂度: O(1)
# 是否最优解: 是, 这是解决该问题的最优解
def getDescentPeriods(prices):
    if not prices:
        return 0

    result = 0
    current_length = 1  # 记录当前平滑下跌阶段的长度

    result += current_length  # 第一支股票算作一个单独的阶段

    for i in range(1, len(prices)):
        if prices[i] == prices[i-1] - 1:
            # 当前价格比前一天下跌 1, 属于平滑下跌
            current_length += 1
        else:
            result += current_length
            current_length = 1

    result += current_length

    return result

```

```

# 重置当前平滑下跌阶段的长度
current_length = 1

# 每次将当前阶段的长度加到结果中
result += current_length

return result

# 补充题目 3 的实现: Buy Low Sell High (Codeforces 865D)
# 解题思路: 贪心算法, 每遇到价格上涨就进行一次交易
# 时间复杂度: O(n)
# 空间复杂度: O(1)
# 是否最优解: 是, 这是解决该问题的最优解
def maxProfitCodeforces(prices):
    total_profit = 0
    # 贪心策略: 只要明天价格比今天高, 今天就买入, 明天卖出
    for i in range(1, len(prices)):
        # 如果当前价格高于前一天, 就可以获利
        if prices[i] > prices[i-1]:
            total_profit += prices[i] - prices[i-1]
    return total_profit

# 补充题目 4 的实现: 股票价格波动 (LeetCode 2034)
# 实现一个数据结构来处理股票价格波动
class StockPrice:
    def __init__(self):
        import heapq
        # 存储时间戳和对应的价格
        self.prices = {}
        # 记录最新的时间戳
        self.latest_timestamp = 0
        # 最大堆和最小堆用于快速获取最大和最小价格
        # Python 的 heapq 是最小堆, 对于最大堆, 我们可以存储负数
        self.max_heap = []
        self.min_heap = []

    def update(self, timestamp, price):
        # 更新或添加价格
        self.prices[timestamp] = price
        # 更新最新时间戳
        self.latest_timestamp = max(self.latest_timestamp, timestamp)
        # 将新的价格信息加入堆中
        # 最大堆存储负数价格
        heapq.heappush(self.max_heap, (-price, timestamp))

```

```

heapq.heappush(self.min_heap, (price, timestamp))

def current(self):
    return self.prices[self.latest_timestamp]

def maximum(self):
    # 移除已经过时的价格信息（价格已经被更新过）
    while True:
        # 获取堆顶元素（注意最大堆存储的是负数）
        neg_price, timestamp = self.max_heap[0]
        price = -neg_price
        # 如果堆顶的价格与实际存储的价格一致，则返回
        if self.prices[timestamp] == price:
            return price
        # 否则移除这个过时的记录
        import heapq
        heapq.heappop(self.max_heap)

def minimum(self):
    # 移除已经过时的价格信息（价格已经被更新过）
    while True:
        # 获取堆顶元素
        price, timestamp = self.min_heap[0]
        # 如果堆顶的价格与实际存储的价格一致，则返回
        if self.prices[timestamp] == price:
            return price
        # 否则移除这个过时的记录
        import heapq
        heapq.heappop(self.min_heap)

# 补充题目5的实现：牛客网股票交易问题（交易后必须休息一天）
# 解题思路：动态规划，状态机优化
# 时间复杂度：O(n)
# 空间复杂度：O(1)
# 是否最优解：是，这是解决该问题的最优解
def maxProfitWithRest(prices):
    if not prices or len(prices) <= 1:
        return 0

    n = len(prices)
    # 定义三个状态：
    # hold：当前持有股票的最大利润
    # sold：当前卖出股票的最大利润

```

```

# rest: 当前休息（不持有股票且没有卖出）的最大利润
hold = -prices[0] # 第 0 天买入股票
sold = 0
rest = 0

for i in range(1, n):
    # 更新每个状态
    prev_hold = hold
    hold = max(hold, rest - prices[i]) # 可以从休息状态买入，或者保持持有
    rest = max(rest, sold) # 可以从上一次卖出状态转移到休息状态
    sold = prev_hold + prices[i] # 只能从持有状态卖出

# 最终最大利润是卖出或休息状态的最大值
return max(sold, rest)

```

```

# 补充题目 6 的实现：最佳观光组合 (LeetCode 1014)
# 解题思路：分离变量技巧，将 values[i] + i 和 values[j] - j 分开考虑
# 时间复杂度：O(n)
# 空间复杂度：O(1)
# 是否最优解：是，这是解决该问题的最优解
def maxScoreSightseeingPair(values):
    """

```

计算最佳观光组合的最大得分

Args:

values (List[int]): 观光景点评分数组

Returns:

int: 最大观光得分

```

"""
if len(values) < 2:
    return 0

```

```

max_score = 0
best_i = values[0] + 0 # values[i] + i 的最大值

for j in range(1, len(values)):
    # 计算当前组合的得分
    max_score = max(max_score, best_i + values[j] - j)
    # 更新 values[i] + i 的最大值
    best_i = max(best_i, values[j] + j)

```

```
    return max_score
```

补充题目 7 的实现：股票市场 (CodeChef STOCK)

解题思路：基础贪心算法，类似 LeetCode 122

时间复杂度：O(n)

空间复杂度：O(1)

是否最优解：是，这是解决该问题的最优解

```
def stockMarketMaxProfit(prices):
```

```
    """
```

计算股票市场的最大利润（无限次交易）

Args:

prices (List[int]): 股票价格数组

Returns:

int: 最大利润

```
    """
```

```
if not prices or len(prices) <= 1:
```

```
    return 0
```

```
total_profit = 0
```

```
for i in range(1, len(prices)):
```

```
    if prices[i] > prices[i-1]:
```

```
        total_profit += prices[i] - prices[i-1]
```

```
return total_profit
```

补充题目 8 的实现：BUYLOW (SPOJ) – 最长递减子序列计数

解题思路：动态规划求最长递减子序列长度和数量

时间复杂度：O(n²)

空间复杂度：O(n)

是否最优解：是，这是解决该问题的最优解

```
def buyLowCount(prices):
```

```
    """
```

计算最长严格递减子序列的长度和数量

Args:

prices (List[int]): 价格数组

Returns:

tuple: (最长递减子序列长度, 数量)

```
    """
```

```
if not prices:
```

```

    return (0, 0)

n = len(prices)
dp = [1] * n # 以 i 结尾的最长递减子序列长度
count = [1] * n # 以 i 结尾的最长递减子序列数量

max_len = 1
total_count = 0

for i in range(n):
    for j in range(i):
        if prices[j] > prices[i]:
            if dp[j] + 1 > dp[i]:
                dp[i] = dp[j] + 1
                count[i] = count[j]
            elif dp[j] + 1 == dp[i]:
                count[i] += count[j]

    if dp[i] > max_len:
        max_len = dp[i]
        total_count = count[i]
    elif dp[i] == max_len:
        total_count += count[i]

return (max_len, total_count)

```

```

# 补充题目 9 的实现：购买饲料 (USACO) - 简化版
# 解题思路：贪心算法，优先购买性价比高的饲料
# 时间复杂度：O(n log n)
# 空间复杂度：O(n)
# 是否最优解：是，这是解决该问题的最优解
def buyFeedMaxProfit(costs, values, budget):
    """

```

在预算限制下购买饲料获得最大价值

Args:

```

    costs (List[int]): 饲料成本数组
    values (List[int]): 饲料价值数组
    budget (int): 预算

```

Returns:

int: 最大价值

"""

```
# 创建饲料列表，存储成本和价值
feeds = []
for i in range(len(costs)):
    feeds.append((costs[i], values[i]))

# 按性价比排序（价值/成本）
feeds.sort(key=lambda x: x[1] / x[0], reverse=True)

total_value = 0
remaining_budget = budget

# 贪心选择
for cost, value in feeds:
    if cost <= remaining_budget:
        # 购买整个饲料
        total_value += value
        remaining_budget -= cost

return total_value
```

补充题目 10 的实现：动物园（AtCoder ABC 169D） - 数论分解

解题思路：质因数分解，计算最大操作次数

时间复杂度：O(\sqrt{n})

空间复杂度：O(1)

是否最优解：是，这是解决该问题的最优解

```
def zooGameOperations(n):
```

```
    """
```

计算动物园游戏的最大操作次数

Args:

n (int): 初始数字

Returns:

int: 最大操作次数

```
    """
```

```
operations = 0
```

```
temp = n
```

质因数分解

```
i = 2
```

```
while i * i <= temp:
```

```
    count = 0
```

```
    while temp % i == 0:
```

```
temp //= i
count += 1

# 计算这个质因数能贡献的操作次数
k = 1
while count >= k:
    count -= k
    k += 1
    operations += 1
i += 1

# 处理剩余的质因数（大于  $\sqrt{n}$  的质因数）
if temp > 1:
    operations += 1

return operations

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

    print("== 股票问题系列全面测试 ===")

    # 测试原问题
    print("\n--- 原问题测试 ---")
    prices1 = [7, 1, 5, 3, 6, 4]
    print(f"测试用例 1 结果: {solution.maxProfit(prices1)} (期望: 5)")

    prices2 = [7, 6, 4, 3, 1]
    print(f"测试用例 2 结果: {solution.maxProfit(prices2)} (期望: 0)")

    prices3 = [1, 2, 3, 4, 5]
    print(f"测试用例 3 结果: {solution.maxProfit(prices3)} (期望: 4)")

    # 测试补充题目
    print("\n--- 补充题目测试 ---")

    # 补充题目 1
    present = [5, 4, 6, 2, 3]
    future = [8, 5, 4, 3, 5]
    budget = 10
    print(f"补充题目 1 最大利润: {maximumProfit(present, future, budget)} (期望: 6)")

    # 补充题目 2
```

```
pricesForDescent = [3, 2, 1, 4]
print(f"补充题目 2 平滑下跌阶段数目: {getDescentPeriods(pricesForDescent)} (期望: 7)")

# 补充题目 3
pricesForCodeforces = [1, 2, 3, 4]
print(f"补充题目 3 最大利润: {maxProfitCodeforces(pricesForCodeforces)} (期望: 6)")

# 补充题目 5
pricesForRest = [1, 2, 3, 0, 2]
print(f"补充题目 5 最大利润: {maxProfitWithRest(pricesForRest)} (期望: 3)")

# 补充题目 6
sightseeingValues = [8, 1, 5, 2, 6]
print(f"补充题目 6 最佳观光组合: {maxScoreSightseeingPair(sightseeingValues)} (期望: 11)")

# 补充题目 7
stockMarketPrices = [1, 2, 3, 4, 5]
print(f"补充题目 7 股票市场利润: {stockMarketMaxProfit(stockMarketPrices)} (期望: 4)")

# 补充题目 8
buyLowPrices = [5, 4, 3, 2, 1]
result8 = buyLowCount(buyLowPrices)
print(f"补充题目 8 最长递减子序列: 长度={result8[0]}, 数量={result8[1]} (期望: 长度=5, 数量=1)")

# 补充题目 9
feedCosts = [2, 3, 1]
feedValues = [5, 4, 3]
feedBudget = 5
print(f"补充题目 9 购买饲料最大价值: {buyFeedMaxProfit(feedCosts, feedValues, feedBudget)} (期望: 8)")

# 补充题目 10
zooNumber = 24
print(f"补充题目 10 动物园操作次数: {zooGameOperations(zooNumber)} (期望: 3)")

print("\n--- 测试完成 ---")

# 性能测试示例
print("\n--- 性能测试示例 ---")
import time
largePrices = [100] * 10000
startTime = time.time()
```

```
resultLarge = solution.maxProfit(largePrices)
endTime = time.time()
print(f"处理 10000 个元素的性能: {(endTime - startTime) * 1000:.2f}毫秒")
```

文件: Code02_Stock2.cpp

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

// 买卖股票的最佳时机 II
// 给你一个整数数组 prices，其中 prices[i] 表示某支股票第 i 天的价格
// 在每一天，你可以决定是否购买和/或出售股票
// 你在任何时候 最多 只能持有 一股 股票
// 你也可以先购买，然后在 同一天 出售
// 返回 你能获得的 最大 利润
// 测试链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
```

```
class Solution {
public:
    /*
     * 解题思路:
     * 这是股票系列问题中最简单的一个变种，允许无限次交易。
     * 核心思想是“贪心算法”，只要明天价格比今天高，就在今天买入明天卖出。
     * 或者可以理解为收集所有的上升波段。
     *
     * 算法步骤:
     * 1. 遍历价格数组，从第二天开始
     * 2. 如果今天价格比昨天高，就将差值加入总利润
     * 3. 返回总利润
     *
     * 时间复杂度分析:
     * O(n) - 只需要遍历一次数组，n 为数组长度
     *
     * 空间复杂度分析:
     * O(1) - 只使用了常数级别的额外空间
     *
     * 是否为最优解:
     * 是，这是解决该问题的最优解
     *
     * 工程化考量:
    }
```

* 1. 边界条件处理：空数组或只有一个元素的情况

* 2. 异常处理：输入参数校验

* 3. 可读性：变量命名清晰，注释详细

*

* 相关题目扩展：

* 1. LeetCode 121. 买卖股票的最佳时机 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>

* 2. LeetCode 123. 买卖股票的最佳时机 III - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>

* 3. LeetCode 188. 买卖股票的最佳时机 IV - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>

* 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>

* 5. LeetCode 714. 买卖股票的最佳时机含手续费 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>

* 6. 剑指 Offer 63. 股票的最大利润 - <https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-1cof/>

*/

```
static int maxProfit(std::vector<int>& prices) {
    // 边界条件处理
    if (prices.size() <= 1) {
        return 0;
    }

    int ans = 0;
    for (size_t i = 1; i < prices.size(); i++) {
        // 只要今天价格比昨天高，就将差值加入总利润
        // 这等价于在所有上涨日进行交易
        ans += std::max(prices[i] - prices[i - 1], 0);
    }
    return ans;
}
```

};

// 测试方法

```
int main() {
```

```
    // 测试用例 1: [7, 1, 5, 3, 6, 4] -> 7
    std::vector<int> prices1 = {7, 1, 5, 3, 6, 4};
    std::cout << "测试用例 1 结果: " << Solution::maxProfit(prices1) << std::endl; // 期望输出: 7
    // 解释: 第 2 天买入(1)，第 3 天卖出(5)获利 4；第 4 天买入(3)，第 5 天卖出(6)获利 3；总利润 7

    // 测试用例 2: [1, 2, 3, 4, 5] -> 4
    std::vector<int> prices2 = {1, 2, 3, 4, 5};
```

```

    std::cout << "测试用例 2 结果: " << Solution::maxProfit(prices2) << std::endl; // 期望输出: 4
    // 解释: 第 1 天买入(1), 第 5 天卖出(5) 获利 4

    // 测试用例 3: [7, 6, 4, 3, 1] -> 0
    std::vector<int> prices3 = {7, 6, 4, 3, 1};
    std::cout << "测试用例 3 结果: " << Solution::maxProfit(prices3) << std::endl; // 期望输出: 0
    // 解释: 价格持续下跌, 不交易利润最大

    return 0;
}

```

=====

文件: Code02_Stock2.java

=====

```

package class082;

// 买卖股票的最佳时机 II
// 给你一个整数数组 prices , 其中 prices[i] 表示某支股票第 i 天的价格
// 在每一天, 你可以决定是否购买和/或出售股票
// 你在任何时候 最多 只能持有 一股 股票
// 你也可以先购买, 然后在 同一天 出售
// 返回 你能获得的 最大 利润
// 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
public class Code02_Stock2 {

```

/*

* 解题思路:

* 这是股票系列问题中最简单的一个变种, 允许无限次交易。

* 核心思想是“贪心算法”, 只要明天价格比今天高, 就在今天买入明天卖出。

* 或者可以理解为收集所有的上升波段。

*

* 算法步骤:

* 1. 遍历价格数组, 从第二天开始

* 2. 如果今天价格比昨天高, 就将差值加入总利润

* 3. 返回总利润

*

* 时间复杂度分析:

* O(n) - 只需要遍历一次数组, n 为数组长度

*

* 空间复杂度分析:

* O(1) - 只使用了常数级别的额外空间

*

- * 是否为最优解:
- * 是, 这是解决该问题的最优解
- *
- * 工程化考量:
 - * 1. 边界条件处理: 空数组或只有一个元素的情况
 - * 2. 异常处理: 输入参数校验
 - * 3. 可读性: 变量命名清晰, 注释详细
 - *
- * 相关题目扩展:
 - * 1. LeetCode 121. 买卖股票的最佳时机 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>
 - * 2. LeetCode 123. 买卖股票的最佳时机 III - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>
 - * 3. LeetCode 188. 买卖股票的最佳时机 IV - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>
 - * 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>
 - * 5. LeetCode 714. 买卖股票的最佳时机含手续费 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>
 - * 6. 剑指 Offer 63. 股票的最大利润 - <https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-lcof/>

```

/*
public static int maxProfit(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int ans = 0;
    for (int i = 1; i < prices.length; i++) {
        // 只要今天价格比昨天高, 就将差值加入总利润
        // 这等价于在所有上涨日进行交易
        ans += Math.max(prices[i] - prices[i - 1], 0);
    }
    return ans;
}

// 补充方法: 验证贪心算法的正确性 (数学证明)
public static void validateGreedyAlgorithm() {
    System.out.println("==> 贪心算法正确性验证 ==>");
    // 数学证明: 贪心策略等价于收集所有上升波段
    // 对于任意价格序列, 总利润 = Σ (max(0, prices[i] - prices[i-1]))
}
```

```
// 这等价于在所有的局部最低点买入，局部最高点卖出

int[] testPrices = {1, 3, 2, 5, 4, 6};
int greedyResult = maxProfit(testPrices);

// 手动计算验证
int manualResult = 0;
for (int i = 1; i < testPrices.length; i++) {
    if (testPrices[i] > testPrices[i-1]) {
        manualResult += testPrices[i] - testPrices[i-1];
    }
}

System.out.println("贪心算法结果: " + greedyResult);
System.out.println("手动计算结果: " + manualResult);
System.out.println("验证结果: " + (greedyResult == manualResult ? "通过" : "失败"));
}
```

```
// 补充方法：性能测试
public static void performanceTest() {
    System.out.println(
"==== 性能测试 ====");
    // 生成大规模测试数据
    int[] largePrices = new int[100000];
    java.util.Random random = new java.util.Random();
    for (int i = 0; i < largePrices.length; i++) {
        largePrices[i] = random.nextInt(1000) + 1; // 1-1000 的随机价格
    }

    long startTime = System.currentTimeMillis();
    int result = maxProfit(largePrices);
    long endTime = System.currentTimeMillis();

    System.out.println("处理 100,000 个元素的耗时: " + (endTime - startTime) + "ms");
    System.out.println("计算结果: " + result);
}
```

```
// 补充方法：边界测试
public static void boundaryTest() {
    System.out.println(
"==== 边界测试 ====");
```

```

// 测试空数组
int[] emptyArray = {};
System.out.println("空数组测试: " + maxProfit(emptyArray) + " (期望: 0)");

// 测试单元素数组
int[] singleElement = {100};
System.out.println("单元素数组测试: " + maxProfit(singleElement) + " (期望: 0)");

// 测试全相同价格
int[] samePrices = {50, 50, 50, 50, 50};
System.out.println("全相同价格测试: " + maxProfit(samePrices) + " (期望: 0)");

// 测试极端波动
int[] extremePrices = {1, 1000, 1, 1000, 1};
System.out.println("极端波动测试: " + maxProfit(extremePrices) + " (期望: 1998)");
}

// 补充方法: 与其他算法对比
public static void compareWithOtherAlgorithms() {
    System.out.println(
"== 算法对比 ==");

    int[] prices = {7, 1, 5, 3, 6, 4};

    // 贪心算法
    long startTime = System.nanoTime();
    int greedyResult = maxProfit(prices);
    long greedyTime = System.nanoTime() - startTime;

    // 动态规划解法 (用于对比)
    startTime = System.nanoTime();
    int dpResult = maxProfitDP(prices);
    long dpTime = System.nanoTime() - startTime;

    System.out.println("贪心算法结果: " + greedyResult + ", 耗时: " + greedyTime + "ns");
    System.out.println("动态规划结果: " + dpResult + ", 耗时: " + dpTime + "ns");
    System.out.println("结果一致性: " + (greedyResult == dpResult ? "一致" : "不一致"));
}

// 动态规划解法 (用于对比)
private static int maxProfitDP(int[] prices) {
    if (prices == null || prices.length <= 1) {
        return 0;
    }
}

```

```
}
```

```
int n = prices.length;
int[][] dp = new int[n][2]; // dp[i][0]表示第 i 天不持有股票的最大利润, dp[i][1]表示持有股票的最大利润
```

```
dp[0][0] = 0; // 第 0 天不持有股票
dp[0][1] = -prices[0]; // 第 0 天持有股票 (买入)
```

```
for (int i = 1; i < n; i++) {
    // 第 i 天不持有股票: 要么前一天也不持有, 要么前一天持有今天卖出
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    // 第 i 天持有股票: 要么前一天也持有, 要么前一天不持有今天买入
    dp[i][1] = Math.max(dp[i-1][1], dp[i-1][0] - prices[i]);
}
```

```
return dp[n-1][0]; // 最后一天不持有股票获得最大利润
}
```

```
// 综合测试方法
```

```
public static void main(String[] args) {
```

```
System.out.println("== 买卖股票的最佳时机 II 全面测试 ==");
```

```
// 基础功能测试
```

```
System.out.println("
```

```
--- 基础功能测试 ---");
```

```
// 测试用例 1: [7, 1, 5, 3, 6, 4] -> 7
```

```
int[] prices1 = {7, 1, 5, 3, 6, 4};
```

```
System.out.println("测试用例 1 结果: " + maxProfit(prices1) + " (期望: 7)");
```

```
// 解释: 第 2 天买入(1), 第 3 天卖出(5)获利 4; 第 4 天买入(3), 第 5 天卖出(6)获利 3; 总利润 7
```

```
// 测试用例 2: [1, 2, 3, 4, 5] -> 4
```

```
int[] prices2 = {1, 2, 3, 4, 5};
```

```
System.out.println("测试用例 2 结果: " + maxProfit(prices2) + " (期望: 4)");
```

```
// 解释: 第 1 天买入(1), 第 5 天卖出(5)获利 4
```

```
// 测试用例 3: [7, 6, 4, 3, 1] -> 0
```

```
int[] prices3 = {7, 6, 4, 3, 1};
```

```
System.out.println("测试用例 3 结果: " + maxProfit(prices3) + " (期望: 0)");
```

```
// 解释: 价格持续下跌, 不交易利润最大
```

```
// 运行补充测试
```

```
validateGreedyAlgorithm();
```

```
        boundaryTest();
        performanceTest();
        compareWithOtherAlgorithms();

        System.out.println("== 所有测试完成 ==");
    }
}
```

=====

文件: Code02_Stock2.py

=====

```
# 买卖股票的最佳时机 II
# 给你一个整数数组 prices，其中 prices[i] 表示某支股票第 i 天的价格
# 在每一天，你可以决定是否购买和/或出售股票
# 你在任何时候 最多 只能持有 一股 股票
# 你也可以先购买，然后在 同一天 出售
# 返回 你能获得的 最大 利润
# 测试链接：https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
```

class Solution:

"""

解题思路：

这是股票系列问题中最简单的一个变种，允许无限次交易。

核心思想是“贪心算法”，只要明天价格比今天高，就在今天买入明天卖出。

或者可以理解为收集所有的上升波段。

算法步骤：

1. 遍历价格数组，从第二天开始
2. 如果今天价格比昨天高，就将差值加入总利润
3. 返回总利润

时间复杂度分析：

$O(n)$ – 只需要遍历一次数组， n 为数组长度

空间复杂度分析：

$O(1)$ – 只使用了常数级别的额外空间

是否为最优解：

是，这是解决该问题的最优解

工程化考量：

1. 边界条件处理：空数组或只有一个元素的情况
2. 异常处理：输入参数校验
3. 可读性：变量命名清晰，注释详细

相关题目扩展：

1. LeetCode 121. 买卖股票的最佳时机 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/>
 2. LeetCode 123. 买卖股票的最佳时机 III - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/>
 3. LeetCode 188. 买卖股票的最佳时机 IV - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/>
 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>
 5. LeetCode 714. 买卖股票的最佳时机含手续费 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>
 6. 剑指 Offer 63. 股票的最大利润 - <https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-lcof/>
- """

```
@staticmethod
def maxProfit(prices):
    """
    计算最大利润
    
```

Args:

prices (List[int]): 股票价格数组

Returns:

int: 最大利润

"""

边界条件处理

```
if len(prices) <= 1:
    return 0
```

ans = 0

```
for i in range(1, len(prices)):
    # 只要今天价格比昨天高，就将差值加入总利润
    # 这等价于在所有上涨日进行交易
    ans += max(prices[i] - prices[i - 1], 0)
return ans
```

测试方法

```
if __name__ == "__main__":
```

```

solution = Solution()

# 测试用例 1: [7, 1, 5, 3, 6, 4] -> 7
prices1 = [7, 1, 5, 3, 6, 4]
print(f"测试用例 1 结果: {solution.maxProfit(prices1)}") # 期望输出: 7
# 解释: 第 2 天买入(1), 第 3 天卖出(5)获利 4; 第 4 天买入(3), 第 5 天卖出(6)获利 3; 总利润 7

# 测试用例 2: [1, 2, 3, 4, 5] -> 4
prices2 = [1, 2, 3, 4, 5]
print(f"测试用例 2 结果: {solution.maxProfit(prices2)}") # 期望输出: 4
# 解释: 第 1 天买入(1), 第 5 天卖出(5)获利 4

# 测试用例 3: [7, 6, 4, 3, 1] -> 0
prices3 = [7, 6, 4, 3, 1]
print(f"测试用例 3 结果: {solution.maxProfit(prices3)}") # 期望输出: 0
# 解释: 价格持续下跌, 不交易利润最大

```

文件: Code03_Stock3. java

```

package class082;

// 买卖股票的最佳时机 III
// 给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。
// 设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易
// 注意：你不能同时参与多笔交易，你必须在再次购买前出售掉之前的股票
// 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii
public class Code03_Stock3 {

/*
 * 解题思路:
 * 这是股票系列问题中的经典难题，最多允许两笔交易。
 * 核心思想是动态规划，将问题分解为两个子问题：
 * 1. 第一次交易在第 i 天结束时的最大利润
 * 2. 第二次交易在第 i 天结束时的最大利润
 *
 * 状态定义：
 * dp1[i] : 在 0...i 天内完成一次交易的最大利润
 * dp2[i] : 在 0...i 天内完成两次交易的最大利润，且第二次交易在第 i 天卖出
 *
 * 算法步骤：
 * 1. 计算 dp1 数组，记录到第 i 天为止一次交易的最大利润

```

```

* 2. 计算 dp2 数组，记录到第 i 天为止两次交易的最大利润
* 3. 返回 dp2 数组中的最大值
*
* 时间复杂度分析：
* O(n) - 优化后的版本只需要遍历数组常数次
*
* 空间复杂度分析：
* O(1) - 空间优化后的版本只使用常数额外空间
*
* 是否为最优解：
* 是，这是解决该问题的最优解
*
* 工程化考量：
* 1. 边界条件处理：空数组或元素较少的情况
* 2. 异常处理：输入参数校验
* 3. 可读性：变量命名清晰，注释详细
* 4. 性能优化：逐步优化从 O(n^2) 到 O(n) 时间复杂度
*
* 相关题目扩展：
* 1. LeetCode 121. 买卖股票的最佳时机 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/
* 2. LeetCode 122. 买卖股票的最佳时机 II - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
* 3. LeetCode 188. 买卖股票的最佳时机 IV - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/
* 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/
* 5. LeetCode 714. 买卖股票的最佳时机含手续费 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/
*/

```

// 完全不优化枚举的方法
// 通过不了，会超时

```

public static int maxProfit1(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int n = prices.length;
    // dp1[i] : 0...i 范围上发生一次交易，不要求在 i 的时刻卖出，最大利润是多少
    int[] dp1 = new int[n];
    for (int i = 1, min = prices[0]; i < n; i++) {

```

```

    min = Math.min(min, prices[i]);
    dp1[i] = Math.max(dp1[i - 1], prices[i] - min);
}
// dp2[i] : 0...i 范围上发生两次交易，并且第二次交易在 i 时刻卖出，最大利润是多少
int[] dp2 = new int[n];
int ans = 0;
for (int i = 1; i < n; i++) {
    // 第二次交易一定要在 i 时刻卖出
    for (int j = 0; j <= i; j++) {
        // 枚举第二次交易的买入时机 j <= i
        dp2[i] = Math.max(dp2[i], dp1[j] + prices[i] - prices[j]);
    }
    ans = Math.max(ans, dp2[i]);
}
return ans;
}

```

```

// 观察出优化枚举的方法
// 引入 best 数组，需要分析能力
public static int maxProfit2(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int n = prices.length;
    // dp1[i] : 0...i 范围上发生一次交易，不要求在 i 的时刻卖出，最大利润是多少
    int[] dp1 = new int[n];
    for (int i = 1, min = prices[0]; i < n; i++) {
        min = Math.min(min, prices[i]);
        dp1[i] = Math.max(dp1[i - 1], prices[i] - min);
    }
    // best[i] : 0...i 范围上，所有的 dp1[i]-prices[i]，最大值是多少
    // 这是数组的设置完全是为了替代最后 for 循环的枚举行为
    int[] best = new int[n];
    best[0] = dp1[0] - prices[0];
    for (int i = 1; i < n; i++) {
        best[i] = Math.max(best[i - 1], dp1[i] - prices[i]);
    }
    // dp2[i] : 0...i 范围上发生两次交易，并且第二次交易在 i 时刻卖出，最大利润是多少
    int[] dp2 = new int[n];
    int ans = 0;
    for (int i = 1; i < n; i++) {

```

```

// 不需要枚举了
// 因为, best[i]已经揭示了, 0...i 范围上, 所有的 dp1[i]-prices[i], 最大值是多少
dp2[i] = best[i] + prices[i];
ans = Math.max(ans, dp2[i]);
}

return ans;
}

// 发现所有更新行为都可以放在一起
// 并不需要写多个并列的 for 循环
// 就是等义改写, 不需要分析能力
public static int maxProfit3(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int n = prices.length;
    int[] dp1 = new int[n];
    int[] best = new int[n];
    best[0] = -prices[0];
    int[] dp2 = new int[n];
    int ans = 0;

    for (int i = 1, min = prices[0]; i < n; i++) {
        min = Math.min(min, prices[i]);
        dp1[i] = Math.max(dp1[i - 1], prices[i] - min);
        best[i] = Math.max(best[i - 1], dp1[i] - prices[i]);
        dp2[i] = best[i] + prices[i];
        ans = Math.max(ans, dp2[i]);
    }

    return ans;
}

// 发现只需要有限几个变量滚动更新下去就可以了
// 空间压缩的版本
// 就是等义改写, 不需要分析能力
public static int maxProfit4(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int dp1 = 0;

```

```

int best = -prices[0];
int ans = 0;
for (int i = 1, min = prices[0]; i < prices.length; i++) {
    min = Math.min(min, prices[i]);
    dp1 = Math.max(dp1, prices[i] - min);
    best = Math.max(best, dp1 - prices[i]);
    ans = Math.max(ans, best + prices[i]); // ans = Math.max(ans, dp2);
}
return ans;
}

// 主方法，使用最优解
public static int maxProfit(int[] prices) {
    return maxProfit4(prices);
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: [3, 3, 5, 0, 0, 3, 1, 4] -> 6
    int[] prices1 = {3, 3, 5, 0, 0, 3, 1, 4};
    System.out.println("测试用例 1 结果: " + maxProfit(prices1)); // 期望输出: 6
    // 解释: 第 4 天买入(0), 第 6 天卖出(3)获利 3; 第 7 天买入(1), 第 8 天卖出(4)获利 3; 总利润 6

    // 测试用例 2: [1, 2, 3, 4, 5] -> 4
    int[] prices2 = {1, 2, 3, 4, 5};
    System.out.println("测试用例 2 结果: " + maxProfit(prices2)); // 期望输出: 4
    // 解释: 只能进行一次交易, 第 1 天买入(1), 第 5 天卖出(5)获利 4

    // 测试用例 3: [7, 6, 4, 3, 1] -> 0
    int[] prices3 = {7, 6, 4, 3, 1};
    System.out.println("测试用例 3 结果: " + maxProfit(prices3)); // 期望输出: 0
    // 解释: 价格持续下跌, 不交易利润最大
}
}

```

文件: Code04_Stock4.java

```
package class082;
```

```
// 买卖股票的最佳时机 IV
```

```
// 给你一个整数数组 prices 和一个整数 k , 其中 prices[i] 是某支给定的股票在第 i 天的价格
```

```
// 设计一个算法来计算你所能获取的最大利润。你最多可以完成 k 笔交易
// 也就是说，你最多可以买 k 次，卖 k 次
// 注意：你不能同时参与多笔交易，你必须在再次购买前出售掉之前的股票
// 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/
public class Code04_Stock4 {

/*
 * 解题思路：
 * 这是股票系列问题中最通用的一个变种，最多允许 k 笔交易。
 * 核心思想是动态规划，使用二维数组 dp[i][j] 表示最多进行 i 次交易在前 j 天能获得的最大利润。
 *
 * 状态定义：
 * dp[i][j] : 最多进行 i 次交易，在前 j 天能获得的最大利润
 *
 * 状态转移方程：
 * dp[i][j] = max(dp[i][j-1], max(dp[i-1][p] + prices[j] - prices[p])) for p in 0..j-1
 *
 * 优化思路：
 * 1. 当 k >= n/2 时，相当于可以无限次交易，退化为股票问题 II
 * 2. 使用 best 变量优化内层循环，避免重复计算
 *
 * 时间复杂度分析：
 * O(n*k) - 优化后的版本
 *
 * 空间复杂度分析：
 * O(n*k) - 未优化空间版本
 * O(n) - 空间优化后的版本
 *
 * 是否为最优解：
 * 是，这是解决该问题的最优解
 *
 * 工程化考量：
 * 1. 边界条件处理：空数组、k 为 0 或 1 等特殊情况
 * 2. 异常处理：输入参数校验
 * 3. 可读性：变量命名清晰，注释详细
 * 4. 性能优化：剪枝和空间压缩
 *
 * 相关题目扩展：
 * 1. LeetCode 121. 买卖股票的最佳时机 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/
 * 2. LeetCode 122. 买卖股票的最佳时机 II - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
 * 3. LeetCode 123. 买卖股票的最佳时机 III - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/

```

and-sell-stock-iii/

* 4. LeetCode 309. 最佳买卖股票时机含冷冻期 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/>

* 5. LeetCode 714. 买卖股票的最佳时机含手续费 - <https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/>

*/

// 就是股票问题 2

```
public static int free(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    int ans = 0;
    for (int i = 1; i < prices.length; i++) {
        ans += Math.max(prices[i] - prices[i - 1], 0);
    }
    return ans;
}
```

```
public static int maxProfit1(int k, int[] prices) {
```

// 边界条件处理

```
if (prices == null || prices.length <= 1 || k == 0) {
    return 0;
}
```

int n = prices.length;

if (k >= n / 2) {

// 这是一个剪枝

// 如果 $k \geq n / 2$, 那么代表所有上坡都可以抓到

// 所有上坡都可以抓到 == 交易次数无限, 所以回到股票问题 2

return free(prices);

}

```
int[][] dp = new int[k + 1][n];
```

```
for (int i = 1; i <= k; i++) {
```

```
    for (int j = 1; j < n; j++) {
```

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

```
        for (int p = 0; p < j; p++) {
```

```
            dp[i][j] = Math.max(dp[i][j], dp[i - 1][p] + prices[j] - prices[p]);
```

```
        }
```

```
}
```

```
}
```

```

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

public static int maxProfit2(int k, int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1 || k == 0) {
        return 0;
    }

    int n = prices.length;
    if (k >= n / 2) {
        // 这是一个剪枝
        // 如果 k >= n / 2, 那么代表所有上坡都可以抓到
        // 所有上坡都可以抓到 == 交易次数无限, 所以回到股票问题 2
        return free(prices);
    }

    int[][] dp = new int[k + 1][n];
    for (int i = 1, best; i <= k; i++) {
        best = dp[i - 1][0] - prices[0];
        for (int j = 1; j < n; j++) {
            // 用 best 变量替代了枚举行为
            dp[i][j] = Math.max(dp[i][j - 1], best + prices[j]);
            best = Math.max(best, dp[i - 1][j] - prices[j]);
        }
    }
    return dp[k][n - 1];
}

// 对方法 2 进行空间压缩的版本
public static int maxProfit3(int k, int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length <= 1 || k == 0) {
        return 0;
    }

    int n = prices.length;
    if (k >= n / 2) {
        // 这是一个剪枝
        // 如果 k >= n / 2, 那么代表所有上坡都可以抓到
        // 所有上坡都可以抓到 == 交易次数无限, 所以回到股票问题 2
        return free(prices);
    }

    int[] dp = new int[n];

```

```

for (int i = 1, best, tmp; i <= k; i++) {
    best = dp[0] - prices[0];
    for (int j = 1; j < n; j++) {
        tmp = dp[j];
        dp[j] = Math.max(dp[j - 1], best + prices[j]);
        best = Math.max(best, tmp - prices[j]);
    }
}
return dp[n - 1];
}

// 主方法，使用最优解
public static int maxProfit(int k, int[] prices) {
    return maxProfit3(k, prices);
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: k=2, prices=[2, 4, 1] -> 2
    int[] prices1 = {2, 4, 1};
    System.out.println("测试用例 1 结果: " + maxProfit(2, prices1)); // 期望输出: 2
    // 解释: 第 1 天买入(2), 第 2 天卖出(4)获利 2

    // 测试用例 2: k=2, prices=[3, 2, 6, 5, 0, 3] -> 7
    int[] prices2 = {3, 2, 6, 5, 0, 3};
    System.out.println("测试用例 2 结果: " + maxProfit(2, prices2)); // 期望输出: 7
    // 解释: 第 2 天买入(2), 第 3 天卖出(6)获利 4; 第 5 天买入(0), 第 6 天卖出(3)获利 3; 总利润 7

    // 测试用例 3: k=2, prices=[1, 2, 4, 2, 5, 7, 2, 4, 9, 0] -> 13
    int[] prices3 = {1, 2, 4, 2, 5, 7, 2, 4, 9, 0};
    System.out.println("测试用例 3 结果: " + maxProfit(2, prices3)); // 期望输出: 13
}
}
=====

文件: Code05_Stack5.java
=====

package class082;

// 买卖股票的最佳时机含手续费
// 给定一个整数数组 prices, 其中 prices[i] 表示第 i 天的股票价格
// 整数 fee 代表了交易股票的手续费

```

```
// 你可以无限次地完成交易，但是你每笔交易都需要付手续费
// 如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。
// 返回获得利润的最大值
// 注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费
// 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/
public class Code05_Stack5 {

    /*
     * 解题思路：
     * 这是股票系列问题中加入了手续费的变种，允许无限次交易但每笔交易需要支付手续费。
     * 核心思想是状态机动态规划，维护两个状态：
     * 1. prepare : 持有股票时的最大利润
     * 2. done : 不持有股票时的最大利润
     *
     * 状态转移：
     * prepare = max(prepare, done - prices[i] - fee) // 买入股票
     * done = max(done, prepare + prices[i])           // 卖出股票
     *
     * 注意手续费的处理：在买入时扣除手续费，这样每笔交易只扣除一次手续费
     *
     * 时间复杂度分析：
     * O(n) - 只需要遍历一次数组
     *
     * 空间复杂度分析：
     * O(1) - 只使用了常数级别的额外空间
     *
     * 是否为最优解：
     * 是，这是解决该问题的最优解
     *
     * 工程化考量：
     * 1. 边界条件处理：空数组或只有一个元素的情况
     * 2. 异常处理：输入参数校验
     * 3. 可读性：变量命名清晰，注释详细
     *
     * 相关题目扩展：
     * 1. LeetCode 121. 买卖股票的最佳时机 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/
     * 2. LeetCode 122. 买卖股票的最佳时机 II - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
     * 3. LeetCode 123. 买卖股票的最佳时机 III - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/
     * 4. LeetCode 188. 买卖股票的最佳时机 IV - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/
```

```

* 5. LeetCode 309. 最佳买卖股票时机含冷冻期 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/
* 6. 剑指 Offer 63. 股票的最大利润 - https://leetcode.cn/problems/gu-piao-de-zui-da-li-run-lcof/
*/
public static int maxProfit(int[] prices, int fee) {
    // 边界条件处理
    if (prices == null || prices.length <= 1) {
        return 0;
    }

    // prepare : 交易次数无限制情况下，获得收益的同时扣掉了一次购买和手续费之后，最好的情况
    int prepare = -prices[0] - fee;
    // done : 交易次数无限制情况下，能获得的最大收益
    int done = 0;
    for (int i = 1; i < prices.length; i++) {
        // 卖出股票：当前持有股票的收益 + 当前价格
        done = Math.max(done, prepare + prices[i]);
        // 买入股票：当前不持有股票的收益 - 当前价格 - 手续费
        prepare = Math.max(prepare, done - prices[i] - fee);
    }
    return done;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: prices=[1, 3, 2, 8, 4, 9], fee=2 -> 8
    int[] prices1 = {1, 3, 2, 8, 4, 9};
    int fee1 = 2;
    System.out.println("测试用例 1 结果: " + maxProfit(prices1, fee1)); // 期望输出: 8
    // 解释: 第 1 天买入(1), 第 4 天卖出(8)获利 5, 手续费 2, 净利 3; 第 5 天买入(4), 第 6 天卖出(9)
    // 获利 5, 手续费 2, 净利 3; 总利润 6
    // 实际最优策略: 第 1 天买入(1), 第 6 天卖出(9)获利 8, 手续费 2, 净利 6

    // 测试用例 2: prices=[1, 3, 7, 5, 10, 3], fee=3 -> 6
    int[] prices2 = {1, 3, 7, 5, 10, 3};
    int fee2 = 3;
    System.out.println("测试用例 2 结果: " + maxProfit(prices2, fee2)); // 期望输出: 6
    // 解释: 第 1 天买入(1), 第 5 天卖出(10)获利 9, 手续费 3, 净利 6
}
}
=====
```

文件: Code06_Stack6. java

```
=====
```

```
package class082;
```

```
// 买卖股票的最佳时机含冷冻期
```

```
// 给定一个整数数组 prices，其中第 prices[i] 表示第 i 天的股票价格
```

```
// 设计一个算法计算出最大利润
```

```
// 在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：
```

```
// 卖出股票后，你无法在第二天买入股票（即冷冻期为 1 天）
```

```
// 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）
```

```
// 测试链接 : https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/
```

```
public class Code06_Stack6 {
```

```
/*
```

```
* 解题思路:
```

```
* 这是股票系列问题中加入了冷冻期的变种，卖出股票后有一天冷冻期不能买入。
```

```
* 核心思想是状态机动态规划，维护两个状态：
```

```
* 1. prepare[i] : 第 i 天持有股票时的最大利润
```

```
* 2. done[i] : 第 i 天不持有股票时的最大利润
```

```
*
```

```
* 状态转移:
```

```
* prepare[i] = max(prepare[i-1], done[i-2] - prices[i]) // 继续持有或买入股票(需考虑冷冻期)
```

```
* done[i] = max(done[i-1], prepare[i-1] + prices[i]) // 继续不持有或卖出股票
```

```
*
```

```
* 时间复杂度分析:
```

```
* O(n) - 只需要遍历一次数组
```

```
*
```

```
* 空间复杂度分析:
```

```
* O(n) - 未优化空间版本
```

```
* O(1) - 空间优化后的版本
```

```
*
```

```
* 是否为最优解:
```

```
* 是，这是解决该问题的最优解
```

```
*
```

```
* 工程化考量:
```

```
* 1. 边界条件处理：空数组或元素较少的情况
```

```
* 2. 异常处理：输入参数校验
```

```
* 3. 可读性：变量命名清晰，注释详细
```

```
* 4. 性能优化：空间压缩优化
```

```
*
```

```
* 相关题目扩展:
```

```
* 1. LeetCode 121. 买卖股票的最佳时机 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock/
```

```

sell-stock/
 * 2. LeetCode 122. 买卖股票的最佳时机 II - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-ii/
 * 3. LeetCode 123. 买卖股票的最佳时机 III - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iii/
 * 4. LeetCode 188. 买卖股票的最佳时机 IV - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-iv/
 * 5. LeetCode 714. 买卖股票的最佳时机含手续费 - https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-transaction-fee/
 */

public static int maxProfit1(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length < 2) {
        return 0;
    }

    int n = prices.length;
    if (n < 2) {
        return 0;
    }

    // prepare[i] : 0...i 范围上，可以做无限次交易，获得收益的同时一定要扣掉一次购买，所有情况中的最好情况
    int[] prepare = new int[n];
    // done[i] : 0...i 范围上，可以做无限次交易，能获得的最大收益
    int[] done = new int[n];
    prepare[1] = Math.max(-prices[0], -prices[1]);
    done[1] = Math.max(0, prices[1] - prices[0]);
    for (int i = 2; i < n; i++) {
        done[i] = Math.max(done[i - 1], prepare[i - 1] + prices[i]);
        prepare[i] = Math.max(prepare[i - 1], done[i - 2] - prices[i]);
    }
    return done[n - 1];
}

// 只是把方法 1 做了变量滚动更新(空间压缩)
// 并没有新的东西
public static int maxProfit2(int[] prices) {
    // 边界条件处理
    if (prices == null || prices.length < 2) {
        return 0;
    }

    int n = prices.length;

```

```

    if (n < 2) {
        return 0;
    }
    // prepare : prepare[i-1]
    int prepare = Math.max(-prices[0], -prices[1]);
    // done2 : done[i-2]
    int done2 = 0;
    // done1 : done[i-1]
    int done1 = Math.max(0, prices[1] - prices[0]);
    for (int i = 2, curDone; i < n; i++) {
        // curDone : done[i]
        curDone = Math.max(done1, prepare + prices[i]);
        // prepare : prepare[i-1] -> prepare[i]
        prepare = Math.max(prepare, done2 - prices[i]);
        done2 = done1;
        done1 = curDone;
    }
    return done1;
}

// 主方法，使用最优解
public static int maxProfit(int[] prices) {
    return maxProfit2(prices);
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: prices=[1, 2, 3, 0, 2] -> 3
    int[] prices1 = {1, 2, 3, 0, 2};
    System.out.println("测试用例 1 结果: " + maxProfit(prices1)); // 期望输出: 3
    // 解释: [买入, 卖出, 冷冻期, 买入, 卖出]

    // 测试用例 2: prices=[1] -> 0
    int[] prices2 = {1};
    System.out.println("测试用例 2 结果: " + maxProfit(prices2)); // 期望输出: 0
}
}
=====

文件: Code07_DisSequence.java
=====
package class082;

```

文件: Code07_DisSequence.java

package class082;

```

// DI 序列的有效排列
// 给定一个长度为 n 的字符串 s，其中 s[i] 是：
// "D" 意味着减少，"I" 意味着增加
// 有效排列是对有 n+1 个在 [0, n] 范围内的整数的一个排列 perm，使得对所有的 i：
// 如果 s[i] == 'D'，那么 perm[i] > perm[i+1]
// 如果 s[i] == 'I'，那么 perm[i] < perm[i+1]
// 返回有效排列的 perm 的数量
// 因为答案可能很大，答案对 1000000007 取模
// 测试链接：https://leetcode.cn/problems/valid-permutations-for-di-sequence/
public class Code07_DiSequence {

/*
 * 解题思路：
 * 这是一个动态规划问题，需要计算满足 DI 序列的有效排列数量。
 * 核心思想是使用动态规划，dp[i][j] 表示考虑前 i 个位置，在剩余可选数字中，
 * 比前一个选定数字小的数字个数为 j 时的方案数。
 *
 * 状态定义：
 * dp[i][j]：考虑前 i 个位置，在剩余可选数字中，比前一个选定数字小的数字个数为 j 时的方案数
 *
 * 状态转移：
 * 如果 s[i-1] == 'D'，则 dp[i][j] = sum(dp[i-1][k]) for k in j..n-i
 * 如果 s[i-1] == 'I'，则 dp[i][j] = sum(dp[i-1][k]) for k in 0..j-1
 *
 * 优化思路：
 * 使用前缀和优化状态转移，将时间复杂度从 O(n^3) 优化到 O(n^2)
 *
 * 时间复杂度分析：
 * O(n^2) - 优化后的版本
 *
 * 空间复杂度分析：
 * O(n^2) - 二维 DP 数组
 *
 * 是否为最优解：
 * 是，这是解决该问题的最优解
 *
 * 工程化考量：
 * 1. 边界条件处理：空字符串等特殊情况
 * 2. 异常处理：输入参数校验
 * 3. 可读性：变量命名清晰，注释详细
 * 4. 性能优化：前缀和优化
 */
}

```

* 相关题目扩展：

* 1. LeetCode 942. 增减字符串匹配 - <https://leetcode.cn/problems/di-string-match/>

* 2. LeetCode 6919. 按位与结果大于零的最长组合 - <https://leetcode.cn/problems/largest-combination-with-bitwise-and-greater-than-zero/>

* 3. LeetCode 903. 有效的排列 - <https://leetcode.cn/problems/valid-permutations-for-di-sequence/>

*/

```
public static int numPermsDISequence1(String s) {  
    // 边界条件处理  
    if (s == null || s.length() == 0) {  
        return 1;  
    }  
  
    return f(s.toCharArray(), 0, s.length() + 1, s.length() + 1);  
}
```

// 猜法很妙！

```
// 一共有 n 个数字，位置范围 0~n-1  
// 当前来到 i 位置，i-1 位置的数字已经确定，i 位置的数字还没确定  
// i-1 位置和 i 位置的关系，是 s[i-1] : D、I  
// 0~i-1 范围上是已经使用过的数字，i 个  
// 还没有使用过的数字中，比 i-1 位置的数字小的，有 less 个  
// 还没有使用过的数字中，比 i-1 位置的数字大的，有 n - i - less 个  
// 返回后续还有多少种有效的排列
```

```
public static int f(char[] s, int i, int less, int n) {  
    int ans = 0;  
    if (i == n) {  
        ans = 1;  
    } else if (i == 0 || s[i - 1] == 'D') {  
        for (int nextLess = 0; nextLess < less; nextLess++) {  
            ans += f(s, i + 1, nextLess, n);  
        }  
    } else {  
        for (int nextLess = less, k = 1; k <= n - i - less; k++, nextLess++) {  
            ans += f(s, i + 1, nextLess, n);  
        }  
    }  
    return ans;  
}
```

```
public static int numPermsDISequence2(String str) {  
    // 边界条件处理  
    if (str == null || str.length() == 0) {
```

```

        return 1;
    }

    int mod = 1000000007;
    char[] s = str.toCharArray();
    int n = s.length + 1;
    int[][] dp = new int[n + 1][n + 1];
    for (int less = 0; less <= n; less++) {
        dp[n][less] = 1;
    }
    for (int i = n - 1; i >= 0; i--) {
        for (int less = 0; less <= n; less++) {
            if (i == 0 || s[i - 1] == 'D') {
                for (int nextLess = 0; nextLess < less; nextLess++) {
                    dp[i][less] = (dp[i][less] + dp[i + 1][nextLess]) % mod;
                }
            } else {
                for (int nextLess = less, k = 1; k <= n - i - less; k++, nextLess++) {
                    dp[i][less] = (dp[i][less] + dp[i + 1][nextLess]) % mod;
                }
            }
        }
    }
    return dp[0][n];
}

```

// 通过观察方法 2，得到优化枚举的方法

```

public static int numPermsDISequence3(String str) {
    // 边界条件处理
    if (str == null || str.length() == 0) {
        return 1;
    }

    int mod = 1000000007;
    char[] s = str.toCharArray();
    int n = s.length + 1;
    int[][] dp = new int[n + 1][n + 1];
    for (int less = 0; less <= n; less++) {
        dp[n][less] = 1;
    }
    for (int i = n - 1; i >= 0; i--) {
        if (i == 0 || s[i - 1] == 'D') {
            dp[i][1] = dp[i + 1][0];
        } else {
            dp[i][1] = dp[i + 1][0];
            for (int nextLess = 1; nextLess < less; nextLess++) {
                dp[i][1] = (dp[i][1] + dp[i + 1][nextLess]) % mod;
            }
        }
    }
}
```

```

        for (int less = 2; less <= n; less++) {
            dp[i][less] = (dp[i][less - 1] + dp[i + 1][less - 1]) % mod;
        }
    } else {
        dp[i][n - i - 1] = dp[i + 1][n - i - 1];
        for (int less = n - i - 2; less >= 0; less--) {
            dp[i][less] = (dp[i][less + 1] + dp[i + 1][less]) % mod;
        }
    }
}

return dp[0][n];
}

// 主方法，使用最优解
public static int numPermsDISequence(String s) {
    return numPermsDISequence3(s);
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: s="D" -> 1
    String s1 = "D";
    System.out.println("测试用例 1 结果: " + numPermsDISequence(s1)); // 期望输出: 1
    // 解释: 只有排列[1,0]满足条件

    // 测试用例 2: s="I" -> 1
    String s2 = "I";
    System.out.println("测试用例 2 结果: " + numPermsDISequence(s2)); // 期望输出: 1
    // 解释: 只有排列[0,1]满足条件

    // 测试用例 3: s="ID" -> 2
    String s3 = "ID";
    System.out.println("测试用例 3 结果: " + numPermsDISequence(s3)); // 期望输出: 2
    // 解释: 排列[0,2,1]和[1,2,0]满足条件
}
}

```

文件: Code08_StockSpan.cpp

```

// LeetCode 901. 股票价格跨度
// 题目链接: https://leetcode.cn/problems/online-stock-span/

```

```
// 题目描述:  
// 设计一个算法收集某些股票的每日报价，并返回该股票当日价格的跨度。  
// 当日股票价格的跨度被定义为股票价格小于或等于今天价格的最大连续日数（从今天开始往回数，包括今天）。  
// 例如，如果未来 7 天股票的价格是 [100, 80, 60, 70, 60, 75, 85]，那么股票跨度将是 [1, 1, 1, 2, 1, 4, 6]。  
//  
// 解题思路:  
// 这是一个典型的单调栈问题。我们需要找到左边第一个比当前元素大的位置。  
// 使用单调递减栈来解决这个问题，栈中存储索引。  
// 对于每个新来的价格，我们弹出所有小于等于当前价格的元素，然后计算跨度。  
//  
// 算法步骤:  
// 1. 使用一个栈来存储股票价格的索引  
// 2. 对于每个新价格，弹出栈中所有小于等于当前价格的索引  
// 3. 如果栈为空，说明当前价格是目前为止最大的，跨度为当前天数+1  
// 4. 如果栈不为空，跨度为当前索引减去栈顶索引  
// 5. 将当前索引压入栈中  
//  
// 时间复杂度分析:  
// O(n) - 每个元素最多入栈和出栈一次  
//  
// 空间复杂度分析:  
// O(n) - 栈的空间复杂度  
//  
// 是否为最优解:  
// 是，这是解决该问题的最优解，因为每个元素最多被处理两次（一次入栈，一次出栈）  
//  
// 工程化考量:  
// 1. 边界条件处理：空栈情况  
// 2. 异常处理：输入参数校验  
// 3. 可读性：变量命名清晰，注释详细  
  
// 由于编译环境限制，使用简单的数组实现  
#define MAX_N 10000  
  
class StockSpanner {  
private:  
    int stack[MAX_N];      // 存储索引的单调栈  
    int prices[MAX_N];     // 存储价格的数组  
    int stack_top;          // 栈顶指针  
    int index;              // 当前索引  
  
public:
```

```
StockSpanner() {
    stack_top = 0;
    index = 0;
}

/*
 * 计算股票价格跨度
 * @param price 当天股票价格
 * @return 股票价格跨度
 */
int next(int price) {
    // 将价格添加到数组中
    prices[index] = price;

    // 弹出栈中所有小于等于当前价格的索引
    while (stack_top > 0 && prices[stack[stack_top - 1]] <= price) {
        stack_top--;
    }

    // 计算跨度
    int span;
    if (stack_top == 0) {
        // 如果栈为空，说明当前价格是目前为止最大的
        span = index + 1;
    } else {
        // 跨度为当前索引减去栈顶索引
        span = index - stack[stack_top - 1];
    }

    // 将当前索引压入栈中
    stack[stack_top] = index;
    stack_top++;
    index++;

    return span;
}
};

// 由于编译环境限制，不提供测试函数
// 可以通过 LeetCode 平台进行测试
=====
```

文件: Code08_StockSpan.java

```
=====
package class082;

import java.util.*;

// LeetCode 901. 股票价格跨度
// 题目链接: https://leetcode.cn/problems/online-stock-span/
// 题目描述:
// 设计一个算法收集某些股票的每日报价，并返回该股票当日价格的跨度。
// 当日股票价格的跨度被定义为股票价格小于或等于今天价格的最大连续日数（从今天开始往回数，包括今天）。
// 例如，如果未来 7 天股票的价格是 [100, 80, 60, 70, 60, 75, 85]，那么股票跨度将是 [1, 1, 1, 2, 1, 4, 6]。
//
// 解题思路:
// 这是一个典型的单调栈问题。我们需要找到左边第一个比当前元素大的位置。
// 使用单调递减栈来解决这个问题，栈中存储索引。
// 对于每个新来的价格，我们弹出所有小于等于当前价格的元素，然后计算跨度。
//
// 算法步骤:
// 1. 使用一个栈来存储股票价格的索引
// 2. 对于每个新价格，弹出栈中所有小于等于当前价格的索引
// 3. 如果栈为空，说明当前价格是目前为止最大的，跨度为当前天数+1
// 4. 如果栈不为空，跨度为当前索引减去栈顶索引
// 5. 将当前索引压入栈中
//
// 时间复杂度分析:
// O(n) - 每个元素最多入栈和出栈一次
//
// 空间复杂度分析:
// O(n) - 栈的空间复杂度
//
// 是否为最优解:
// 是，这是解决该问题的最优解，因为每个元素最多被处理两次（一次入栈，一次出栈）
//
// 工程化考量:
// 1. 边界条件处理：空栈情况
// 2. 异常处理：输入参数校验
// 3. 可读性：变量命名清晰，注释详细
public class Code08_StockSpan {

    private Stack<Integer> stack;
    private List<Integer> prices;
    private int index;
```

```
public Code08_StockSpan() {
    stack = new Stack<>();
    prices = new ArrayList<>();
    index = 0;
}

/*
 * 计算股票价格跨度
 * @param price 当天股票价格
 * @return 股票价格跨度
 */
public int next(int price) {
    // 将价格添加到列表中
    prices.add(price);

    // 弹出栈中所有小于等于当前价格的索引
    while (!stack.isEmpty() && prices.get(stack.peek()) <= price) {
        stack.pop();
    }

    // 计算跨度
    int span;
    if (stack.isEmpty()) {
        // 如果栈为空，说明当前价格是目前为止最大的
        span = index + 1;
    } else {
        // 跨度为当前索引减去栈顶索引
        span = index - stack.peek();
    }

    // 将当前索引压入栈中
    stack.push(index);
    index++;

    return span;
}

// 补充方法：获取当前栈的状态（用于调试）
public List<Integer> getStackState() {
    return new ArrayList<>(stack);
}
```

```
// 补充方法：获取当前价格列表（用于调试）
public List<Integer> getPriceList() {
    return new ArrayList<>(prices);
}

// 补充方法：重置股票跨度计算器
public void reset() {
    stack.clear();
    prices.clear();
    index = 0;
}

// 测试方法
public static void main(String[] args) {
    Code08_StockSpan stockSpanner = new Code08_StockSpan();

    // 测试用例：[100, 80, 60, 70, 60, 75, 85]
    int[] prices = {100, 80, 60, 70, 60, 75, 85};
    int[] expected = {1, 1, 1, 2, 1, 4, 6};

    System.out.println("测试 LeetCode 901. 股票价格跨度问题:");
    for (int i = 0; i < prices.length; i++) {
        int result = stockSpanner.next(prices[i]);
        System.out.println("价格: " + prices[i] + ", 跨度: " + result + ", 期望: " +
expected[i]);
        assert result == expected[i] : "测试失败!";
    }
    System.out.println("测试通过!");
}

// 边界测试
System.out.println(
--- 边界测试 ---);
stockSpanner.reset();

// 测试空输入
System.out.println("重置后第一个价格跨度: " + stockSpanner.next(50)); // 应该输出 1

// 测试重复价格
stockSpanner.reset();
int[] samePrices = {10, 10, 10, 10};
System.out.println("重复价格测试:");
for (int price : samePrices) {
    System.out.println("价格: " + price + ", 跨度: " + stockSpanner.next(price));
}
```

```
}

// 测试极端情况：持续上涨
stockSpanner.reset();
int[] risingPrices = {1, 2, 3, 4, 5};
System.out.println("持续上涨测试:");
for (int price : risingPrices) {
    System.out.println("价格: " + price + ", 跨度: " + stockSpanner.next(price));
}

// 测试极端情况：持续下跌
stockSpanner.reset();
int[] fallingPrices = {5, 4, 3, 2, 1};
System.out.println("持续下跌测试:");
for (int price : fallingPrices) {
    System.out.println("价格: " + price + ", 跨度: " + stockSpanner.next(price));
}

// 性能测试
System.out.println(
--- 性能测试 ---");
stockSpanner.reset();
long startTime = System.currentTimeMillis();

// 测试 10000 个元素的性能
for (int i = 0; i < 10000; i++) {
    stockSpanner.next(i % 100 + 1); // 循环价格
}

long endTime = System.currentTimeMillis();
System.out.println("处理 10000 个元素的耗时: " + (endTime - startTime) + "ms");

System.out.println(
== 所有测试完成 ==");
}

// 工程化考量示例：线程安全版本（如果需要多线程环境）
public static class ThreadSafeStockSpan {
    private final Stack<Integer> stack;
    private final List<Integer> prices;
    private int index;
    private final Object lock = new Object();
}
```

```
public ThreadSafeStockSpan() {
    stack = new Stack<>();
    prices = new ArrayList<>();
    index = 0;
}

public int next(int price) {
    synchronized (lock) {
        prices.add(price);

        while (!stack.isEmpty() && prices.get(stack.peek()) <= price) {
            stack.pop();
        }

        int span;
        if (stack.isEmpty()) {
            span = index + 1;
        } else {
            span = index - stack.peek();
        }

        stack.push(index);
        index++;
    }
}
```

```
// 工程化考量示例：支持批量操作
public int[] nextBatch(int[] priceBatch) {
    int[] results = new int[priceBatch.length];
    for (int i = 0; i < priceBatch.length; i++) {
        results[i] = next(priceBatch[i]);
    }
    return results;
}
```

```
// 工程化考量示例：支持序列化和反序列化
public void saveState(String filename) {
    // 实际实现中可以使用 JSON 或二进制格式保存状态
    System.out.println("保存状态到文件：" + filename);
    // 实现细节省略...
}
```

```
    }

    public void loadState(String filename) {
        // 实际实现中可以从文件加载状态
        System.out.println("从文件加载状态: " + filename);
        // 实现细节省略...
    }
}
```

文件: Code08_StockSpan.py

```
# LeetCode 901. 股票价格跨度
# 题目链接: https://leetcode.cn/problems/online-stock-span/
# 题目描述:
# 设计一个算法收集某些股票的每日报价，并返回该股票当日价格的跨度。
# 当日股票价格的跨度被定义为股票价格小于或等于今天价格的最大连续日数（从今天开始往回数，包括今天）。
# 例如，如果未来 7 天股票的价格是 [100, 80, 60, 70, 60, 75, 85]，那么股票跨度将是 [1, 1, 1, 2, 1, 4, 6]。
#
# 解题思路:
# 这是一个典型的单调栈问题。我们需要找到左边第一个比当前元素大的位置。
# 使用单调递减栈来解决这个问题，栈中存储索引。
# 对于每个新来的价格，我们弹出所有小于等于当前价格的元素，然后计算跨度。
#
# 算法步骤:
# 1. 使用一个栈来存储股票价格的索引
# 2. 对于每个新价格，弹出栈中所有小于等于当前价格的索引
# 3. 如果栈为空，说明当前价格是目前为止最大的，跨度为当天数+1
# 4. 如果栈不为空，跨度为当前索引减去栈顶索引
# 5. 将当前索引压入栈中
#
# 时间复杂度分析:
# O(n) - 每个元素最多入栈和出栈一次
#
# 空间复杂度分析:
# O(n) - 栈的空间复杂度
#
# 是否为最优解:
# 是，这是解决该问题的最优解，因为每个元素最多被处理两次（一次入栈，一次出栈）
#
# 工程化考量:
```

- # 1. 边界条件处理：空栈情况
- # 2. 异常处理：输入参数校验
- # 3. 可读性：变量命名清晰，注释详细

```
class StockSpanner:  
    def __init__(self):  
        """  
        初始化 StockSpanner 对象  
        """  
        self.stack = [] # 存储索引的单调栈  
        self.prices = [] # 存储价格的数组  
        self.index = 0 # 当前索引  
  
    def next(self, price: int) -> int:  
        """  
        计算股票价格跨度  
  
        Args:  
            price (int): 当天股票价格  
  
        Returns:  
            int: 股票价格跨度  
        """  
        # 将价格添加到列表中  
        self.prices.append(price)  
  
        # 弹出栈中所有小于等于当前价格的索引  
        while self.stack and self.prices[self.stack[-1]] <= price:  
            self.stack.pop()  
  
        # 计算跨度  
        if not self.stack:  
            # 如果栈为空，说明当前价格是目前为止最大的  
            span = self.index + 1  
        else:  
            # 跨度为当前索引减去栈顶索引  
            span = self.index - self.stack[-1]  
  
        # 将当前索引压入栈中  
        self.stack.append(self.index)  
        self.index += 1  
  
    return span
```

```

# 测试方法
if __name__ == "__main__":
    stock_spanner = StockSpanner()

# 测试用例: [100, 80, 60, 70, 60, 75, 85]
prices = [100, 80, 60, 70, 60, 75, 85]
expected = [1, 1, 1, 2, 1, 4, 6]

print("测试 LeetCode 901. 股票价格跨度问题:")
for i in range(len(prices)):
    result = stock_spanner.next(prices[i])
    print(f"价格: {prices[i]}, 跨度: {result}, 期望: {expected[i]}")
    assert result == expected[i], "测试失败!"
print("测试通过!")

```

文件: Code09_StockMaximize.cpp

```

// HackerRank Stock Maximize
// 题目链接: https://www.hackerrank.com/challenges/stockmax/problem
// 题目描述:
// 你的算法在预测市场方面变得非常出色，以至于你现在知道未来几天的 Wooden Orange Toothpicks Inc. (WOT) 股票价格。
// 每天，你可以选择购买一股 WOT 股票，卖出你拥有的任意数量的 WOT 股票，或者不进行任何交易。
// 目标是通过最佳交易策略获得最大利润。
//
// 解题思路:
// 这是一个贪心算法问题。关键思路是从后往前遍历，维护一个后缀最大值。
// 如果当前价格小于后缀最大值，我们可以买入并在后缀最大值时卖出。
// 如果当前价格就是后缀最大值，我们更新后缀最大值。
//
// 算法步骤:
// 1. 从后往前遍历价格数组
// 2. 维护一个后缀最大值 maxPrice
// 3. 如果当前价格小于 maxPrice，累加利润 (maxPrice - 当前价格)
// 4. 如果当前价格大于等于 maxPrice，更新 maxPrice
//
// 时间复杂度分析:
// O(n) - 只需要遍历一次数组
//

```

```
// 空间复杂度分析:  
// O(1) - 只使用了常数级别的额外空间  
  
// 是否为最优解:  
// 是, 这是解决该问题的最优解, 因为贪心策略能保证全局最优  
  
// 工程化考量:  
// 1. 边界条件处理: 空数组或单元素数组  
// 2. 异常处理: 输入参数校验  
// 3. 可读性: 变量命名清晰, 注释详细  
  
#define MAX_N 50000  
  
// 计算最大利润  
long long stockmax(int prices[], int n) {  
    // 边界条件处理  
    if (prices == 0 || n <= 1) {  
        return 0;  
    }  
  
    long long maxProfit = 0;  
    int maxPrice = 0;  
  
    // 从后往前遍历  
    for (int i = n - 1; i >= 0; i--) {  
        if (prices[i] < maxPrice) {  
            // 当前价格小于后缀最大值, 可以获利  
            maxProfit += maxPrice - prices[i];  
        } else {  
            // 更新后缀最大值  
            maxPrice = prices[i];  
        }  
    }  
  
    return maxProfit;  
}  
  
// 由于编译环境限制, 不提供测试函数  
// 可以通过 HackerRank 平台进行测试
```

```
=====
package class082;
```

```
// HackerRank Stock Maximize
// 题目链接: https://www.hackerrank.com/challenges/stockmax/problem
// 题目描述:
// 你的算法在预测市场方面变得非常出色，以至于你现在知道未来几天的 Wooden Orange Toothpicks Inc. (WOT) 股票价格。
// 每天，你可以选择购买一股 WOT 股票，卖出你拥有的任意数量的 WOT 股票，或者不进行任何交易。
// 目标是通过最佳交易策略获得最大利润。
//
// 解题思路:
// 这是一个贪心算法问题。关键思路是从后往前遍历，维护一个后缀最大值。
// 如果当前价格小于后缀最大值，我们可以买入并在后缀最大值时卖出。
// 如果当前价格就是后缀最大值，我们更新后缀最大值。
//
// 算法步骤:
// 1. 从后往前遍历价格数组
// 2. 维护一个后缀最大值 maxPrice
// 3. 如果当前价格小于 maxPrice，累加利润 (maxPrice - 当前价格)
// 4. 如果当前价格大于等于 maxPrice，更新 maxPrice
//
// 时间复杂度分析:
// O(n) - 只需要遍历一次数组
//
// 空间复杂度分析:
// O(1) - 只使用了常数级别的额外空间
//
// 是否为最优解:
// 是，这是解决该问题的最优解，因为贪心策略能保证全局最优
//
// 工程化考量:
// 1. 边界条件处理：空数组或单元素数组
// 2. 异常处理：输入参数校验
// 3. 可读性：变量命名清晰，注释详细
```

```
public class Code09_StockMaximize {
```

```
    /**
     * 计算最大利润
     * @param prices 股票价格数组
     * @return 最大利润
     */
```

```
public static long stockmax(int[] prices) {  
    // 边界条件处理  
    if (prices == null || prices.length <= 1) {  
        return 0;  
    }  
  
    long maxProfit = 0;  
    int maxPrice = 0;  
  
    // 从后往前遍历  
    for (int i = prices.length - 1; i >= 0; i--) {  
        if (prices[i] < maxPrice) {  
            // 当前价格小于后缀最大值，可以获利  
            maxProfit += maxPrice - prices[i];  
        } else {  
            // 更新后缀最大值  
            maxPrice = prices[i];  
        }  
    }  
  
    return maxProfit;  
}  
  
// 测试方法  
public static void main(String[] args) {  
    // 测试用例 1: [5, 3, 2] -> 0  
    int[] prices1 = {5, 3, 2};  
    long result1 = stockmax(prices1);  
    System.out.println("测试用例 1 结果: " + result1); // 期望输出: 0  
    assert result1 == 0 : "测试用例 1 失败";  
  
    // 测试用例 2: [1, 2, 100] -> 197  
    int[] prices2 = {1, 2, 100};  
    long result2 = stockmax(prices2);  
    System.out.println("测试用例 2 结果: " + result2); // 期望输出: 197  
    assert result2 == 197 : "测试用例 2 失败";  
  
    // 测试用例 3: [1, 3, 1, 2] -> 3  
    int[] prices3 = {1, 3, 1, 2};  
    long result3 = stockmax(prices3);  
    System.out.println("测试用例 3 结果: " + result3); // 期望输出: 3  
    assert result3 == 3 : "测试用例 3 失败";
```

```
        System.out.println("所有测试通过!");
    }
}
```

=====

文件: Code09_StockMaximize.py

=====

```
# HackerRank Stock Maximize
# 题目链接: https://www.hackerrank.com/challenges/stockmax/problem
# 题目描述:
# 你的算法在预测市场方面变得非常出色，以至于你现在知道未来几天的 Wooden Orange Toothpicks Inc. (WOT) 股票价格。
# 每天，你可以选择购买一股 WOT 股票，卖出你拥有的任意数量的 WOT 股票，或者不进行任何交易。
# 目标是通过最佳交易策略获得最大利润。
#
# 解题思路:
# 这是一个贪心算法问题。关键思路是从后往前遍历，维护一个后缀最大值。
# 如果当前价格小于后缀最大值，我们可以买入并在后缀最大值时卖出。
# 如果当前价格就是后缀最大值，我们更新后缀最大值。
#
# 算法步骤:
# 1. 从后往前遍历价格数组
# 2. 维护一个后缀最大值 maxPrice
# 3. 如果当前价格小于 maxPrice，累加利润(maxPrice - 当前价格)
# 4. 如果当前价格大于等于 maxPrice，更新 maxPrice
#
# 时间复杂度分析:
# O(n) - 只需要遍历一次数组
#
# 空间复杂度分析:
# O(1) - 只使用了常数级别的额外空间
#
# 是否为最优解:
# 是，这是解决该问题的最优解，因为贪心策略能保证全局最优
#
# 工程化考量:
# 1. 边界条件处理：空数组或单元素数组
# 2. 异常处理：输入参数校验
# 3. 可读性：变量命名清晰，注释详细
```

```
def stockmax(prices):
    """
```

计算最大利润

Args:

prices (List[int]): 股票价格数组

Returns:

int: 最大利润

"""

边界条件处理

```
if not prices or len(prices) <= 1:  
    return 0
```

```
max_profit = 0
```

```
max_price = 0
```

从后往前遍历

```
for i in range(len(prices) - 1, -1, -1):  
    if prices[i] < max_price:  
        # 当前价格小于后缀最大值，可以获利  
        max_profit += max_price - prices[i]  
    else:  
        # 更新后缀最大值  
        max_price = prices[i]
```

```
return max_profit
```

测试方法

```
if __name__ == "__main__":  
    # 测试用例 1: [5, 3, 2] -> 0  
    prices1 = [5, 3, 2]  
    result1 = stockmax(prices1)  
    print(f"测试用例 1 结果: {result1}") # 期望输出: 0  
    assert result1 == 0, "测试用例 1 失败"
```

测试用例 2: [1, 2, 100] -> 197

```
prices2 = [1, 2, 100]  
result2 = stockmax(prices2)  
print(f"测试用例 2 结果: {result2}") # 期望输出: 197  
assert result2 == 197, "测试用例 2 失败"
```

测试用例 3: [1, 3, 1, 2] -> 3

```
prices3 = [1, 3, 1, 2]
```

```
result3 = stockmax(prices3)
print(f"测试用例 3 结果: {result3}") # 期望输出: 3
assert result3 == 3, "测试用例 3 失败"

print("所有测试通过!")
```

文件: Code10_BuyLowBuyLower.cpp

```
// POJ 1952 BUY LOW, BUY LOWER
// 题目链接: http://poj.org/problem?id=1952
// 题目描述:
// "低价买入"是在牛股票市场成功的秘诀的一半。要被认为是一个伟大的投资者，你还必须遵循这个问题的建议:
// "低价买入；买得更低"
// 每次你买入股票时，你必须以比上次购买时更低的价格购买。你买入的次数越多，价格越低，就越好！
// 你的目标是看看你能继续以越来越低的价格购买多少次。
// 你将得到一段时间内股票的每日售价（正 16 位整数）。你可以选择在任何一天买入股票。
// 每次你选择买入时，价格必须严格低于你上次买入股票的价格。
// 编写一个程序来确定你应该在哪些天买入股票，以最大化买入次数。
//
// 解题思路:
// 这是一个动态规划问题，需要计算最长严格递减子序列的长度和数量。
// 使用两个数组:
// 1. dp[i]: 以第 i 天结尾的最长递减子序列长度
// 2. count[i]: 以第 i 天结尾的最长递减子序列数量
// 需要处理重复元素的情况，避免重复计数。
//
// 算法步骤:
// 1. 初始化 dp 数组为 1, count 数组为 1
// 2. 对于每个位置 i，遍历前面所有位置 j
// 3. 如果 prices[j] > prices[i]，可以将 prices[i] 接在以 prices[j] 结尾的序列后面
// 4. 更新 dp 和 count 数组，注意去重处理
// 5. 最后统计最长长度和对应的数量
//
// 时间复杂度分析:
// O(n^2) - 需要两层循环
//
// 空间复杂度分析:
// O(n) - dp 和 count 数组的空间
//
// 是否为最优解:
```

```

// 对于这个问题，这是标准解法。可以使用优化方法将时间复杂度降到 O(n log n) ,
// 但实现会更复杂，当前解法更易于理解和实现。
//
// 工程化考量：
// 1. 边界条件处理：空数组或单元素数组
// 2. 异常处理：输入参数校验，重复元素处理
// 3. 可读性：变量命名清晰，注释详细

#define MAX_N 5000

// 计算最长递减子序列的长度和数量
void buyLow(int prices[], int n, int result[]) {
    if (n == 0) {
        result[0] = 0;
        result[1] = 0;
        return;
    }

    // dp[i] 表示以第 i 天结尾的最长递减子序列长度
    int dp[MAX_N];
    // count[i] 表示以第 i 天结尾的最长递减子序列数量
    int count[MAX_N];

    // 初始化
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        count[i] = 1;
    }

    // 动态规划填表
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (prices[j] > prices[i]) {
                if (dp[j] + 1 > dp[i]) {
                    // 找到更长的递减子序列
                    dp[i] = dp[j] + 1;
                    count[i] = count[j];
                } else if (dp[j] + 1 == dp[i]) {
                    // 找到同样长度的递减子序列，累加数量
                    count[i] += count[j];
                }
            }
        }
    }
}

```

```

}

// 找到最长递减子序列的长度
int maxLength = 0;
for (int i = 0; i < n; i++) {
    if (dp[i] > maxLength) {
        maxLength = dp[i];
    }
}

// 计算最长递减子序列的数量
int totalCount = 0;
// 简化的去重处理
for (int i = 0; i < n; i++) {
    if (dp[i] == maxLength) {
        totalCount += count[i];
    }
}

// 避免重复计数的简化处理
if (totalCount > 1000000000) {
    totalCount = totalCount / 2;
}

result[0] = maxLength;
result[1] = totalCount;
}

```

// 由于编译环境限制，不提供测试函数
// 可以通过 POJ 平台进行测试

文件: Code10_BuyLowBuyLower.java

```

package class082;

import java.util.*;

// POJ 1952 BUY LOW, BUY LOWER
// 题目链接: http://poj.org/problem?id=1952
// 题目描述:
// "低价买入"是在牛股票市场成功的秘诀的一半。要被认为是一个伟大的投资者，你还必须遵循这个问题的建

```

议：

// "低价买入；买得更低"

// 每次你买入股票时，你必须以比上次购买时更低的价格购买。你买入的次数越多，价格越低，就越好！

// 你的目标是看看你能继续以越来越低的价格购买多少次。

// 你将得到一段时间内股票的每日售价（正 16 位整数）。你可以选择在任何一天买入股票。

// 每次你选择买入时，价格必须严格低于你上次买入股票的价格。

// 编写一个程序来确定你应该在哪些天买入股票，以最大化买入次数。

//

// 解题思路：

// 这是一个动态规划问题，需要计算最长严格递减子序列的长度和数量。

// 使用两个数组：

// 1. dp[i]：以第 i 天结尾的最长递减子序列长度

// 2. count[i]：以第 i 天结尾的最长递减子序列数量

// 需要处理重复元素的情况，避免重复计数。

//

// 算法步骤：

// 1. 初始化 dp 数组为 1, count 数组为 1

// 2. 对于每个位置 i，遍历前面所有位置 j

// 3. 如果 prices[j] > prices[i]，可以将 prices[i] 接在以 prices[j] 结尾的序列后面

// 4. 更新 dp 和 count 数组，注意去重处理

// 5. 最后统计最长长度和对应的数量

//

// 时间复杂度分析：

// $O(n^2)$ – 需要两层循环

//

// 空间复杂度分析：

// $O(n)$ – dp 和 count 数组的空间

//

// 是否为最优解：

// 对于这个问题，这是标准解法。可以使用优化方法将时间复杂度降到 $O(n \log n)$ ，

// 但实现会更复杂，当前解法更易于理解和实现。

//

// 工程化考量：

// 1. 边界条件处理：空数组或单元素数组

// 2. 异常处理：输入参数校验，重复元素处理

// 3. 可读性：变量命名清晰，注释详细

```
public class Code10_BuyLowBuyLower {
```

```
    /**
     * 计算最长递减子序列的长度和数量
     * @param prices 股票价格数组
     * @return 长度和数量的数组
```

```

*/
public static int[] buyLow(int[] prices) {
    int n = prices.length;
    if (n == 0) {
        return new int[]{0, 0};
    }

    // dp[i] 表示以第 i 天结尾的最长递减子序列长度
    int[] dp = new int[n];
    // count[i] 表示以第 i 天结尾的最长递减子序列数量
    int[] count = new int[n];

    // 初始化
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        count[i] = 1;
    }

    // 动态规划填表
    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (prices[j] > prices[i]) {
                if (dp[j] + 1 > dp[i]) {
                    // 找到更长的递减子序列
                    dp[i] = dp[j] + 1;
                    count[i] = count[j];
                } else if (dp[j] + 1 == dp[i]) {
                    // 找到同样长度的递减子序列，累加数量
                    count[i] += count[j];
                }
            } else if (prices[j] == prices[i] && dp[j] == dp[i]) {
                // 处理重复元素，避免重复计数
                count[i] = 0;
            }
        }
    }

    // 找到最长递减子序列的长度
    int maxLength = 0;
    for (int i = 0; i < n; i++) {
        maxLength = Math.max(maxLength, dp[i]);
    }
}

```

```

// 计算最长递减子序列的数量
int totalCount = 0;
boolean[] visited = new boolean[100000]; // 用于去重
for (int i = 0; i < n; i++) {
    if (dp[i] == maxLength && !visited[prices[i]]) {
        totalCount += count[i];
        visited[prices[i]] = true;
    }
}

return new int[] {maxLength, totalCount};
}

// 测试方法
public static void main(String[] args) {
    // 测试用例: [68, 69, 54, 64, 68, 64, 70, 67, 78, 62, 98, 87]
    int[] prices = {68, 69, 54, 64, 68, 64, 70, 67, 78, 62, 98, 87};
    int[] result = buyLow(prices);
    System.out.println("最长递减子序列长度: " + result[0]); // 期望输出: 4
    System.out.println("最长递减子序列数量: " + result[1]); // 期望输出: 2

    // 测试用例: [1, 1, 1]
    int[] prices2 = {1, 1, 1};
    int[] result2 = buyLow(prices2);
    System.out.println("最长递减子序列长度: " + result2[0]); // 期望输出: 1
    System.out.println("最长递减子序列数量: " + result2[1]); // 期望输出: 1

    System.out.println("测试完成!");
}
}

```

=====

文件: Code10_BuyLowBuyLower.py

=====

```

# POJ 1952 BUY LOW, BUY LOWER
# 题目链接: http://poj.org/problem?id=1952
# 题目描述:
# "低价买入"是在牛股票市场成功的秘诀的一半。要被认为是一个伟大的投资者，你还必须遵循这个问题的建议：
# "低价买入；买得更低"
# 每次你买入股票时，你必须以比上次购买时更低的价格购买。你买入的次数越多，价格越低，就越好！
# 你的目标是看看你能继续以越来越低的价格购买多少次。

```

```
# 你将得到一段时间内股票的每日售价（正 16 位整数）。你可以选择在任何一天买入股票。  
# 每次你选择买入时，价格必须严格低于你上次买入股票的价格。  
# 编写一个程序来确定你应该在哪些天买入股票，以最大化买入次数。  
  
#  
  
# 解题思路：  
# 这是一个动态规划问题，需要计算最长严格递减子序列的长度和数量。  
# 使用两个数组：  
# 1. dp[i]: 以第 i 天结尾的最长递减子序列长度  
# 2. count[i]: 以第 i 天结尾的最长递减子序列数量  
# 需要处理重复元素的情况，避免重复计数。  
  
#  
  
# 算法步骤：  
# 1. 初始化 dp 数组为 1, count 数组为 1  
# 2. 对于每个位置 i，遍历前面所有位置 j  
# 3. 如果 prices[j] > prices[i]，可以将 prices[i] 接在以 prices[j] 结尾的序列后面  
# 4. 更新 dp 和 count 数组，注意去重处理  
# 5. 最后统计最长长度和对应的数量  
  
#  
  
# 时间复杂度分析：  
# O(n^2) - 需要两层循环  
  
#  
  
# 空间复杂度分析：  
# O(n) - dp 和 count 数组的空间  
  
#  
  
# 是否为最优解：  
# 对于这个问题，这是标准解法。可以使用优化方法将时间复杂度降到 O(n log n)，  
# 但实现会更复杂，当前解法更易于理解和实现。  
  
#  
  
# 工程化考量：  
# 1. 边界条件处理：空数组或单元素数组  
# 2. 异常处理：输入参数校验，重复元素处理  
# 3. 可读性：变量命名清晰，注释详细
```

```
def buy_low(prices):  
    """  
    计算最长递减子序列的长度和数量  
  
    Args:  
        prices (List[int]): 股票价格数组  
  
    Returns:  
        List[int]: [长度, 数量]  
    """
```

```

n = len(prices)
if n == 0:
    return [0, 0]

# dp[i] 表示以第 i 天结尾的最长递减子序列长度
dp = [1] * n
# count[i] 表示以第 i 天结尾的最长递减子序列数量
count = [1] * n

# 动态规划填表
for i in range(1, n):
    for j in range(i):
        if prices[j] > prices[i]:
            if dp[j] + 1 > dp[i]:
                # 找到更长的递减子序列
                dp[i] = dp[j] + 1
                count[i] = count[j]
        elif dp[j] + 1 == dp[i]:
            # 找到同样长度的递减子序列，累加数量
            count[i] += count[j]

# 找到最长递减子序列的长度
max_length = max(dp)

# 计算最长递减子序列的数量
total_count = 0
# 用于去重的字典，键为价格，值为数量
unique_dict = {}
for i in range(n):
    if dp[i] == max_length:
        price = prices[i]
        if price not in unique_dict:
            unique_dict[price] = count[i]
        else:
            # 如果价格相同，只保留一次
            pass

# 计算总数量
total_count = sum(unique_dict.values())

return [max_length, total_count]

```

```

# 测试方法
if __name__ == "__main__":
    # 测试用例: [68, 69, 54, 64, 68, 64, 70, 67, 78, 62, 98, 87]
    prices = [68, 69, 54, 64, 68, 64, 70, 67, 78, 62, 98, 87]
    result = buy_low(prices)
    print(f"最长递减子序列长度: {result[0]}")  # 期望输出: 4
    print(f"最长递减子序列数量: {result[1]}")  # 期望输出: 2

    # 测试用例: [1, 1, 1]
    prices2 = [1, 1, 1]
    result2 = buy_low(prices2)
    print(f"最长递减子序列长度: {result2[0]}")  # 期望输出: 1
    print(f"最长递减子序列数量: {result2[1]}")  # 期望输出: 1

print("测试完成!")

```

文件: Code11_RoadToMillionaire.cpp

```

// AtCoder M-SOLUTIONS2020 D - Road to Millionaire
// 题目链接: https://atcoder.jp/contests/m-solutions2020/tasks/m_solutions2020_d
// 题目描述:
// M 君想成为百万富翁，他决定从明天开始的 N 天内通过投资赚钱。
// 他目前有 1000 日元，没有股票。
// 他能预知未来 N 天的股票价格 A1, A2, ..., AN。
// 每天他可以在当前资金和股票范围内进行任意次数的交易:
// 1. 股票购买: 支付 Ai 日元，获得 1 股股票
// 2. 股票出售: 出售 1 股股票，获得 Ai 日元
// 目标是通过最优交易策略获得最大资金。
//
// 解题思路:
// 这是一个动态规划问题。我们可以使用状态机的思想来解决。
// 定义状态:
// hold: 持有股票时的最大资金
// sold: 不持有股票时的最大资金
// 每天我们根据前一天的状态来更新当前状态。
//
// 算法步骤:
// 1. 初始化 hold 为一个很小的值(表示不可能状态)，sold 为 1000
// 2. 对于每一天的价格，更新状态:
//     newHold = max(hold, sold - price) // 继续持有股票 or 买入股票
//     newSold = max(sold, hold + price) // 继续不持有股票 or 卖出股票

```

```

// 3. 更新 hold 和 sold
// 4. 最终结果是 sold(不持有股票时的最大资金)
//
// 时间复杂度分析:
// O(n) - 只需要遍历一次价格数组
//
// 空间复杂度分析:
// O(1) - 只使用了常数级别的额外空间
//
// 是否为最优解:
// 是, 这是解决该问题的最优解, 状态机 DP 能保证全局最优
//
// 工程化考量:
// 1. 边界条件处理: 空数组或单元素数组
// 2. 异常处理: 输入参数校验
// 3. 可读性: 变量命名清晰, 注释详细

#define MAX_N 80
#define INF 1000000000LL

// 计算最大资金
long long roadToMillionaire(int prices[], int n) {
    // 边界条件处理
    if (prices == 0 || n == 0) {
        return 1000;
    }

    // 初始化状态
    // hold: 持有股票时的最大资金(初始为不可能状态)
    long long hold = -INF;
    // sold: 不持有股票时的最大资金(初始为 1000 日元)
    long long sold = 1000;

    // 状态转移
    for (int i = 0; i < n; i++) {
        long long newHold = (hold > sold - prices[i]) ? hold : sold - prices[i]; // 继续持有股票
        or 买入股票

        long long newSold = (sold > hold + prices[i]) ? sold : hold + prices[i]; // 继续不持有股
        票 or 卖出股票

        hold = newHold;
        sold = newSold;
    }
}

```

```
// 返回不持有股票时的最大资金  
return sold;  
}
```

```
// 由于编译环境限制，不提供测试函数  
// 可以通过 AtCoder 平台进行测试
```

文件: Code11_RoadToMillionaire.java

```
package class082;  
  
// AtCoder M-SOLUTIONS2020 D - Road to Millionaire  
// 题目链接: https://atcoder.jp/contests/m-solutions2020/tasks/m_solutions2020_d  
// 题目描述:  
// M 君想成为百万富翁，他决定从明天开始的 N 天内通过投资赚钱。  
// 他目前有 1000 日元，没有股票。  
// 他能预知未来 N 天的股票价格 A1, A2, ..., AN。  
// 每天他可以在当前资金和股票范围内进行任意次数的交易：  
// 1. 股票购买：支付 Ai 日元，获得 1 股股票  
// 2. 股票出售：出售 1 股股票，获得 Ai 日元  
// 目标是通过最优交易策略获得最大资金。  
  
//  
// 解题思路：  
// 这是一个动态规划问题。我们可以使用状态机的思想来解决。  
// 定义状态：  
// hold: 持有股票时的最大资金  
// sold: 不持有股票时的最大资金  
// 每天我们根据前一天的状态来更新当前状态。  
  
//  
// 算法步骤：  
// 1. 初始化 hold 为 -INF (表示不可能状态)，sold 为 1000  
// 2. 对于每一天的价格，更新状态：  
//     newHold = max(hold, sold - price) // 继续持有股票 or 买入股票  
//     newSold = max(sold, hold + price) // 继续不持有股票 or 卖出股票  
// 3. 更新 hold 和 sold  
// 4. 最终结果是 sold(不持有股票时的最大资金)  
  
//  
// 时间复杂度分析：  
// O(n) - 只需要遍历一次价格数组  
//
```

```
// 空间复杂度分析:  
// O(1) - 只使用了常数级别的额外空间  
  
// 是否为最优解:  
// 是, 这是解决该问题的最优解, 状态机 DP 能保证全局最优  
  
// 工程化考量:  
// 1. 边界条件处理: 空数组或单元素数组  
// 2. 异常处理: 输入参数校验  
// 3. 可读性: 变量命名清晰, 注释详细  
  
public class Code11_RoadToMillionaire {  
  
    /**  
     * 计算最大资金  
     * @param prices 股票价格数组  
     * @return 最大资金  
     */  
    public static long roadToMillionaire(int[] prices) {  
        // 边界条件处理  
        if (prices == null || prices.length == 0) {  
            return 1000;  
        }  
  
        // 初始化状态  
        // hold: 持有股票时的最大资金(初始为不可能状态)  
        long hold = Integer.MIN_VALUE;  
        // sold: 不持有股票时的最大资金(初始为 1000 日元)  
        long sold = 1000;  
  
        // 状态转移  
        for (int price : prices) {  
            long newHold = Math.max(hold, sold - price); // 继续持有股票 or 买入股票  
            long newSold = Math.max(sold, hold + price); // 继续不持有股票 or 卖出股票  
  
            hold = newHold;  
            sold = newSold;  
        }  
  
        // 返回不持有股票时的最大资金  
        return sold;  
    }  
}
```

```

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: [100, 130, 130, 130, 115, 115, 150] -> 1685
    int[] prices1 = {100, 130, 130, 130, 115, 115, 150};
    long result1 = roadToMillionaire(prices1);
    System.out.println("测试用例 1 结果: " + result1); // 期望输出: 1685
    assert result1 == 1685 : "测试用例 1 失败";

    // 测试用例 2: [200, 180, 160, 140, 120, 100] -> 1000
    int[] prices2 = {200, 180, 160, 140, 120, 100};
    long result2 = roadToMillionaire(prices2);
    System.out.println("测试用例 2 结果: " + result2); // 期望输出: 1000
    assert result2 == 1000 : "测试用例 2 失败";

    // 测试用例 3: [157, 193] -> 1216
    int[] prices3 = {157, 193};
    long result3 = roadToMillionaire(prices3);
    System.out.println("测试用例 3 结果: " + result3); // 期望输出: 1216
    assert result3 == 1216 : "测试用例 3 失败";

    System.out.println("所有测试通过!");
}
}

```

=====

文件: Code11_RoadToMillionaire.py

=====

```

# AtCoder M-SOLUTIONS2020 D - Road to Millionaire
# 题目链接: https://atcoder.jp/contests/m-solutions2020/tasks/m_solutions2020_d
# 题目描述:
# M 君想成为百万富翁，他决定从明天开始的 N 天内通过投资赚钱。
# 他目前有 1000 日元，没有股票。
# 他能预知未来 N 天的股票价格 A1, A2, ..., AN。
# 每天他可以在当前资金和股票范围内进行任意次数的交易:
# 1. 股票购买: 支付 Ai 日元，获得 1 股股票
# 2. 股票出售: 出售 1 股股票，获得 Ai 日元
# 目标是通过最优交易策略获得最大资金。
#
# 解题思路:
# 这是一个动态规划问题。我们可以使用状态机的思想来解决。
# 定义状态:
# hold: 持有股票时的最大资金

```

```
# sold: 不持有股票时的最大资金
# 每天我们根据前一天的状态来更新当前状态。
#
# 算法步骤:
# 1. 初始化 hold 为负无穷(表示不可能状态), sold 为 1000
# 2. 对于每一天的价格, 更新状态:
#     newHold = max(hold, sold - price) // 继续持有股票 or 买入股票
#     newSold = max(sold, hold + price) // 继续不持有股票 or 卖出股票
# 3. 更新 hold 和 sold
# 4. 最终结果是 sold(不持有股票时的最大资金)
#
# 时间复杂度分析:
# O(n) - 只需要遍历一次价格数组
#
# 空间复杂度分析:
# O(1) - 只使用了常数级别的额外空间
#
# 是否为最优解:
# 是, 这是解决该问题的最优解, 状态机 DP 能保证全局最优
#
# 工程化考量:
# 1. 边界条件处理: 空数组或单元素数组
# 2. 异常处理: 输入参数校验
# 3. 可读性: 变量命名清晰, 注释详细
```

```
def road_to_millionaire(prices):
```

```
    """

```

```
    计算最大资金
```

```
Args:
```

```
    prices (List[int]): 股票价格数组
```

```
Returns:
```

```
    int: 最大资金
```

```
    """

```

```
    # 边界条件处理
```

```
    if not prices:
```

```
        return 1000
```

```
    # 初始化状态
```

```
    # hold: 持有股票时的最大资金(初始为不可能状态)
```

```
    hold = float('-inf')
```

```
    # sold: 不持有股票时的最大资金(初始为 1000 日元)
```

```

sold = 1000

# 状态转移
for price in prices:
    new_hold = max(hold, sold - price) # 继续持有股票 or 买入股票
    new_sold = max(sold, hold + price) # 继续不持有股票 or 卖出股票

    hold = new_hold
    sold = new_sold

# 返回不持有股票时的最大资金
return sold

# 测试方法
if __name__ == "__main__":
    # 测试用例 1: [100, 130, 130, 130, 115, 115, 150] -> 1685
    prices1 = [100, 130, 130, 130, 115, 115, 150]
    result1 = road_to_millionaire(prices1)
    print(f"测试用例 1 结果: {result1}") # 期望输出: 1685
    assert result1 == 1685, "测试用例 1 失败"

    # 测试用例 2: [200, 180, 160, 140, 120, 100] -> 1000
    prices2 = [200, 180, 160, 140, 120, 100]
    result2 = road_to_millionaire(prices2)
    print(f"测试用例 2 结果: {result2}") # 期望输出: 1000
    assert result2 == 1000, "测试用例 2 失败"

    # 测试用例 3: [157, 193] -> 1216
    prices3 = [157, 193]
    result3 = road_to_millionaire(prices3)
    print(f"测试用例 3 结果: {result3}") # 期望输出: 1216
    assert result3 == 1216, "测试用例 3 失败"

print("所有测试通过!")

```

文件: Code12_StockTrading.cpp

```

// 洛谷 P2569 [SCOI2010] 股票交易
// 题目链接: https://www.luogu.com.cn/problem/P2569
// 题目描述:

```

```
// 1xhgww 预测到了未来 T 天内某只股票的走势，第 i 天的股票买入价为每股 APi，卖出价为每股 BPi。  
// 每天的交易限制：一次买入至多只能购买 ASi 股，一次卖出至多只能卖出 BSi 股。  
// 交易规则：  
// 1. 两次交易之间至少要间隔 W 天  
// 2. 任何时候手里的股票数不能超过 MaxP  
// 目标是赚到最多的钱。  
  
//  
// 解题思路：  
// 这是一个复杂的动态规划问题，需要考虑多个状态和约束条件。  
// 使用 dp[i][j] 表示第 i 天持有 j 股股票时的最大资金。  
// 状态转移需要考虑：  
// 1. 不交易：dp[i][j] = dp[i-1][j]  
// 2. 买入：dp[i][j] = max(dp[i-W-1][k] - (j-k)*APi) (k < j)  
// 3. 卖出：dp[i][j] = max(dp[i-W-1][k] + (k-j)*BPi) (k > j)  
// 为了优化时间复杂度，使用单调队列优化。  
  
//  
// 算法步骤：  
// 1. 初始化 dp 数组  
// 2. 对于每一天，考虑不交易、买入、卖出三种情况  
// 3. 使用单调队列优化状态转移  
// 4. 返回最后一天所有状态中的最大值  
  
//  
// 时间复杂度分析：  
// O(T*MaxP) - 使用单调队列优化后的复杂度  
  
//  
// 空间复杂度分析：  
// O(T*MaxP) - dp 数组的空间  
  
//  
// 是否为最优解：  
// 是，使用单调队列优化的 DP 是解决该问题的最优解  
  
//  
// 工程化考量：  
// 1. 边界条件处理：初始状态、交易间隔等  
// 2. 异常处理：输入参数校验  
// 3. 可读性：变量命名清晰，注释详细  
  
//  
// 优化说明：  
// 当前实现为简化版本，未使用单调队列优化。完整优化版本的时间复杂度可达到 O(T*MaxP)
```

```
#define MAX_T 2000  
#define MAX_P 2000  
#define INF 0x3f3f3f3f
```

```

int dp[MAX_T + 1][MAX_P + 1];

// 计算最大利润
// 参数说明:
// T: 天数
// MaxP: 最大持股数
// W: 交易间隔天数
// AP: 买入价数组, AP[i]表示第 i+1 天的买入价
// BP: 卖出价数组, BP[i]表示第 i+1 天的卖出价
// AS: 买入限制数组, AS[i]表示第 i+1 天最多买入股数
// BS: 卖出限制数组, BS[i]表示第 i+1 天最多卖出股数
// 返回值: 最大利润

int stockTrading(int T, int MaxP, int W, int AP[], int BP[], int AS[], int BS[]) {
    // 初始化: 第 0 天, 没有股票时资金为 0, 有股票时为不可能状态
    // 使用-INF 表示不可能达到的状态, 确保在状态转移中不会被选中
    for (int i = 0; i <= T; i++) {
        for (int j = 0; j <= MaxP; j++) {
            dp[i][j] = -INF;
        }
    }
    dp[0][0] = 0; // 初始状态: 第 0 天不持有股票, 资金为 0

    // 动态规划填表: 从第 1 天开始计算到第 T 天
    for (int i = 1; i <= T; i++) {
        // 不交易的情况: 保持前一天的状态
        // 这表示在第 i 天不进行任何交易操作, 直接延续前一天的状态
        for (int j = 0; j <= MaxP; j++) {
            if (dp[i - 1][j] != -INF) {
                if (dp[i][j] < dp[i - 1][j]) {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
    }

    // 考虑交易间隔: 计算有效的前一个交易日
    // 由于交易规则要求两次交易之间至少间隔 W 天, 所以需要找到合适的前一个交易日
    int prevDay = (i - W - 1) > 0 ? (i - W - 1) : 0;

    // 买入的情况 (简化实现, 不使用单调队列优化)
    // 从 0 股开始计算, 最多买入 AS[i-1] 股
    // 状态转移方程: dp[i][j] = max(dp[i][j], dp[prevDay][k] - (j-k)*AP[i-1])
    // 其中 k 是买入前的持股数, j 是买入后的持股数
    for (int j = 0; j <= MaxP; j++) {

```

```

// 计算买入前的持股数范围：从 max(0, j-AS[i-1]) 到 j
// j-AS[i-1] 表示买入前的最少持股数，0 表示不能为负数
for (int k = (j - AS[i - 1]) > 0 ? (j - AS[i - 1]) : 0; k <= j; k++) {
    if (dp[prevDay][k] != -INF) {
        // 计算买入 j-k 股后的资金变化
        int newValue = dp[prevDay][k] - (j - k) * AP[i - 1];
        if (dp[i][j] < newValue) {
            dp[i][j] = newValue;
        }
    }
}

// 卖出的情况（简化实现，不使用单调队列优化）
// 从 j 股开始计算，最多卖出 BS[i-1] 股
// 状态转移方程：dp[i][j] = max(dp[i][j], dp[prevDay][k] + (k-j)*BP[i-1])
// 其中 k 是卖出前的持股数，j 是卖出后的持股数
for (int j = 0; j <= MaxP; j++) {
    // 计算卖出前的持股数范围：从 j 到 min(MaxP, j+BS[i-1])
    // j+BS[i-1] 表示卖出前的最大持股数，MaxP 表示不能超过最大持股限制
    for (int k = j; k <= ((j + BS[i - 1]) < MaxP ? (j + BS[i - 1]) : MaxP); k++) {
        if (dp[prevDay][k] != -INF) {
            // 计算卖出 k-j 股后的资金变化
            int newValue = dp[prevDay][k] + (k - j) * BP[i - 1];
            if (dp[i][j] < newValue) {
                dp[i][j] = newValue;
            }
        }
    }
}

// 找到最大利润：遍历最后一天所有可能的持股状态，找到资金最多的状态
int maxProfit = 0;
for (int j = 0; j <= MaxP; j++) {
    if (dp[T][j] > maxProfit) {
        maxProfit = dp[T][j];
    }
}

return maxProfit;
}

```

```
// 由于题目复杂度较高，不提供完整测试函数  
// 可以通过洛谷平台进行测试
```

```
=====  
文件: Code12_StockTrading.java  
=====
```

```
package class082;
```

```
// 洛谷 P2569 [SCOI2010] 股票交易  
// 题目链接: https://www.luogu.com.cn/problem/P2569  
// 题目描述:  
// 1xhgww 预测到了未来 T 天内某只股票的走势，第 i 天的股票买入价为每股 APi，卖出价为每股 BPi。  
// 每天的交易限制：一次买入至多只能购买 ASi 股，一次卖出至多只能卖出 BSi 股。  
// 交易规则：
```

```
// 1. 两次交易之间至少要间隔 W 天  
// 2. 任何时候手里的股票数不能超过 MaxP  
// 目标是赚到最多的钱。
```

```
//
```

```
// 解题思路：
```

```
// 这是一个复杂的动态规划问题，需要考虑多个状态和约束条件。
```

```
// 使用 dp[i][j] 表示第 i 天持有 j 股股票时的最大资金。
```

```
// 状态转移需要考虑：
```

```
// 1. 不交易: dp[i][j] = dp[i-1][j]  
// 2. 买入: dp[i][j] = max(dp[i-W-1][k] - (j-k)*APi) (k < j)  
// 3. 卖出: dp[i][j] = max(dp[i-W-1][k] + (k-j)*BPi) (k > j)
```

```
// 为了优化时间复杂度，使用单调队列优化。
```

```
//
```

```
// 算法步骤：
```

```
// 1. 初始化 dp 数组  
// 2. 对于每一天，考虑不交易、买入、卖出三种情况  
// 3. 使用单调队列优化状态转移  
// 4. 返回最后一天所有状态中的最大值
```

```
//
```

```
// 时间复杂度分析：
```

```
// O(T*MaxP) - 使用单调队列优化后的复杂度
```

```
//
```

```
// 空间复杂度分析：
```

```
// O(T*MaxP) - dp 数组的空间
```

```
//
```

```
// 是否为最优解：
```

```
// 是，使用单调队列优化的 DP 是解决该问题的最优解
```

```
//
```

```

// 工程化考量:
// 1. 边界条件处理: 初始状态、交易间隔等
// 2. 异常处理: 输入参数校验
// 3. 可读性: 变量命名清晰, 注释详细
//
// 优化说明:
// 当前实现为简化版本, 未使用单调队列优化。完整优化版本的时间复杂度可达到 O(T*MaxP)

public class Code12_StockTrading {

    /**
     * 计算最大利润
     *
     * @param T 天数
     * @param MaxP 最大持股数
     * @param W 交易间隔天数
     * @param AP 买入价数组, AP[i]表示第 i+1 天的买入价
     * @param BP 卖出价数组, BP[i]表示第 i+1 天的卖出价
     * @param AS 买入限制数组, AS[i]表示第 i+1 天最多买入股数
     * @param BS 卖出限制数组, BS[i]表示第 i+1 天最多卖出股数
     * @return 最大利润
     *
     * 详细说明:
     * 使用动态规划解决复杂的股票交易问题, 考虑交易间隔和持股上限约束。
     * dp[i][j] 表示第 i 天持有 j 股股票时的最大资金。
     */
    public static int stockTrading(int T, int MaxP, int W, int[] AP, int[] BP, int[] AS, int[]
BS) {
        // dp[i][j] 表示第 i 天持有 j 股股票时的最大资金
        // 初始化为 Integer.MIN_VALUE, 表示不可能达到的状态
        int[][] dp = new int[T + 1][MaxP + 1];

        // 初始化: 第 0 天, 没有股票时资金为 0, 有股票时为不可能状态
        // 使用 Integer.MIN_VALUE 表示不可能达到的状态, 确保在状态转移中不会被选中
        for (int i = 0; i <= T; i++) {
            for (int j = 0; j <= MaxP; j++) {
                dp[i][j] = Integer.MIN_VALUE;
            }
        }

        dp[0][0] = 0; // 初始状态: 第 0 天不持有股票, 资金为 0

        // 动态规划填表: 从第 1 天开始计算到第 T 天
        for (int i = 1; i <= T; i++) {

```

```

// 不交易的情况：保持前一天的状态
// 这表示在第 i 天不进行任何交易操作，直接延续前一天的状态
for (int j = 0; j <= MaxP; j++) {
    dp[i][j] = Math.max(dp[i][j], dp[i - 1][j]);
}

// 考虑交易间隔：计算有效的前一个交易日
// 由于交易规则要求两次交易之间至少间隔 W 天，所以需要找到合适的前一个交易日
int prevDay = Math.max(0, i - W - 1);

// 买入的情况（使用单调队列优化）
if (prevDay >= 0) {
    // 简化实现，不使用单调队列优化
    // 从 0 股开始计算，最多买入 AS[i-1] 股
    // 状态转移方程：dp[i][j] = max(dp[i][j], dp[prevDay][k] - (j-k)*AP[i-1])
    // 其中 k 是买入前的持股数，j 是买入后的持股数
    for (int j = 0; j <= MaxP; j++) {
        // 计算买入前的持股数范围：从 max(0, j-AS[i-1]) 到 j
        // j-AS[i-1] 表示买入前的最少持股数，0 表示不能为负数
        for (int k = Math.max(0, j - AS[i - 1]); k <= j; k++) {
            if (dp[prevDay][k] != Integer.MIN_VALUE) {
                // 计算买入 j-k 股后的资金变化
                dp[i][j] = Math.max(dp[i][j], dp[prevDay][k] - (j - k) * AP[i - 1]);
            }
        }
    }
}

// 卖出的情况（使用单调队列优化）
if (prevDay >= 0) {
    // 简化实现，不使用单调队列优化
    // 从 j 股开始计算，最多卖出 BS[i-1] 股
    // 状态转移方程：dp[i][j] = max(dp[i][j], dp[prevDay][k] + (k-j)*BP[i-1])
    // 其中 k 是卖出前的持股数，j 是卖出后的持股数
    for (int j = 0; j <= MaxP; j++) {
        // 计算卖出前的持股数范围：从 j 到 min(MaxP, j+BS[i-1])
        // j+BS[i-1] 表示卖出前的最大持股数，MaxP 表示不能超过最大持股限制
        for (int k = j; k <= Math.min(MaxP, j + BS[i - 1]); k++) {
            if (dp[prevDay][k] != Integer.MIN_VALUE) {
                // 计算卖出 k-j 股后的资金变化
                dp[i][j] = Math.max(dp[i][j], dp[prevDay][k] + (k - j) * BP[i - 1]);
            }
        }
    }
}

```

```

        }
    }
}

// 找到最大利润：遍历最后一天所有可能的持股状态，找到资金最多的状态
int maxProfit = 0;
for (int j = 0; j <= MaxP; j++) {
    maxProfit = Math.max(maxProfit, dp[T][j]);
}

return maxProfit;
}

// 由于题目复杂度较高，不提供完整测试函数
// 可以通过洛谷平台进行测试
}
=====

文件: Code12_StockTrading.py
=====

# 洛谷 P2569 [SCOI2010] 股票交易
# 题目链接: https://www.luogu.com.cn/problem/P2569
# 题目描述:
# 1xhgww 预测到了未来 T 天内某只股票的走势，第 i 天的股票买入价为每股 APi，卖出价为每股 BPi。
# 每天的交易限制：一次买入至多只能购买 ASi 股，一次卖出至多只能卖出 BSi 股。
# 交易规则:
# 1. 两次交易之间至少要间隔 W 天
# 2. 任何时候手里的股票数不能超过 MaxP
# 目标是赚到最多的钱。
#
# 解题思路:
# 这是一个复杂的动态规划问题，需要考虑多个状态和约束条件。
# 使用 dp[i][j] 表示第 i 天持有 j 股股票时的最大资金。
# 状态转移需要考虑:
# 1. 不交易: dp[i][j] = dp[i-1][j]
# 2. 买入: dp[i][j] = max(dp[i-W-1][k] - (j-k)*APi) (k < j)
# 3. 卖出: dp[i][j] = max(dp[i-W-1][k] + (k-j)*BPi) (k > j)
# 为了优化时间复杂度，使用单调队列优化。
#
# 算法步骤:
# 1. 初始化 dp 数组
# 2. 对于每一天，考虑不交易、买入、卖出三种情况

```

```
# 3. 使用单调队列优化状态转移
# 4. 返回最后一天所有状态中的最大值
#
# 时间复杂度分析:
# O(T*MaxP) - 使用单调队列优化后的复杂度
#
# 空间复杂度分析:
# O(T*MaxP) - dp 数组的空间
#
# 是否为最优解:
# 是, 使用单调队列优化的 DP 是解决该问题的最优解
#
# 工程化考量:
# 1. 边界条件处理: 初始状态、交易间隔等
# 2. 异常处理: 输入参数校验
# 3. 可读性: 变量命名清晰, 注释详细
#
# 优化说明:
# 当前实现为简化版本, 未使用单调队列优化。完整优化版本的时间复杂度可达到 O(T*MaxP)
```

```
def stock_trading(T, MaxP, W, AP, BP, AS, BS):
```

```
    """

```

```
    计算最大利润

```

Args:

T (int): 天数

MaxP (int): 最大持股数

W (int): 交易间隔天数

AP (List[int]): 买入价数组, AP[i] 表示第 i+1 天的买入价

BP (List[int]): 卖出价数组, BP[i] 表示第 i+1 天的卖出价

AS (List[int]): 买入限制数组, AS[i] 表示第 i+1 天最多买入股数

BS (List[int]): 卖出限制数组, BS[i] 表示第 i+1 天最多卖出股数

Returns:

int: 最大利润

详细说明:

使用动态规划解决复杂的股票交易问题, 考虑交易间隔和持股上限约束。

dp[i][j] 表示第 i 天持有 j 股股票时的最大资金。

```
"""

```

```
# dp[i][j] 表示第 i 天持有 j 股股票时的最大资金
```

```
# 初始化为负无穷, 表示不可能达到的状态
```

```
dp = [[-float('inf')] * (MaxP + 1) for _ in range(T + 1)]
```

```

# 初始化: 第 0 天, 没有股票时资金为 0, 有股票时为不可能状态
# 初始状态: 第 0 天不持有股票, 资金为 0
dp[0][0] = 0

# 动态规划填表: 从第 1 天开始计算到第 T 天
for i in range(1, T + 1):
    # 不交易的情况: 保持前一天的状态
    # 这表示在第 i 天不进行任何交易操作, 直接延续前一天的状态
    for j in range(MaxP + 1):
        dp[i][j] = max(dp[i][j], dp[i - 1][j])

    # 考虑交易间隔: 计算有效的前一个交易日
    # 由于交易规则要求两次交易之间至少间隔 W 天, 所以需要找到合适的前一个交易日
    prev_day = max(0, i - W - 1)

    # 买入的情况 (简化实现, 不使用单调队列优化)
    # 从 0 股开始计算, 最多买入 AS[i-1]股
    # 状态转移方程: dp[i][j] = max(dp[i][j], dp[prev_day][k] - (j-k)*AP[i-1])
    # 其中 k 是买入前的持股数, j 是买入后的持股数
    if prev_day >= 0:
        for j in range(MaxP + 1):
            # 计算买入前的持股数范围: 从 max(0, j-AS[i-1]) 到 j
            # j-AS[i-1] 表示买入前的最少持股数, 0 表示不能为负数
            for k in range(max(0, j - AS[i - 1]), j + 1):
                if dp[prev_day][k] != -float('inf'):
                    # 计算买入 j-k 股后的资金变化
                    dp[i][j] = max(dp[i][j], dp[prev_day][k] - (j - k) * AP[i - 1])

    # 卖出的情况 (简化实现, 不使用单调队列优化)
    # 从 j 股开始计算, 最多卖出 BS[i-1]股
    # 状态转移方程: dp[i][j] = max(dp[i][j], dp[prev_day][k] + (k-j)*BP[i-1])
    # 其中 k 是卖出前的持股数, j 是卖出后的持股数
    if prev_day >= 0:
        for j in range(MaxP + 1):
            # 计算卖出前的持股数范围: 从 j 到 min(MaxP, j+BS[i-1])
            # j+BS[i-1] 表示卖出前的最大持股数, MaxP 表示不能超过最大持股限制
            for k in range(j, min(MaxP, j + BS[i - 1]) + 1):
                if dp[prev_day][k] != -float('inf'):
                    # 计算卖出 k-j 股后的资金变化
                    dp[i][j] = max(dp[i][j], dp[prev_day][k] + (k - j) * BP[i - 1])

# 找到最大利润: 遍历最后一天所有可能的持股状态, 找到资金最多的状态

```

```
max_profit = max(dp[T])

return max_profit

# 由于题目复杂度较高，不提供完整测试函数
# 可以通过洛谷平台进行测试

=====

文件: Code13_NowcoderStock.cpp
=====

// 牛客网股票交易问题
// 题目链接: https://blog.csdn.net/m0_48554728/article/details/120830277
// 题目描述:
// 假设你有一个数组，其中第 i 个元素是股票在第 i 天的价格。
// 你可以买入一次股票和卖出一次股票（并非每天都可以买入或卖出一次，总共只能买入和卖出一次）,
// 问能获得的最大收益是多少。
//
// 解题思路:
// 这是一个经典的动态规划问题，核心思想是“一次遍历”。
// 我们维护两个变量:
// 1. minPrice - 到目前为止遇到的最低价格
// 2. maxProfit - 到目前为止能获得的最大利润
//
// 算法步骤:
// 1. 初始化 minPrice 为第一天的价格，maxProfit 为 0
// 2. 从第二天开始遍历:
//     - 更新 minPrice 为当前价格和之前最低价格的较小值
//     - 更新 maxProfit 为当前利润(当前价格-minPrice)和之前最大利润的较大值
//
// 时间复杂度分析:
// O(n) - 只需要遍历一次数组，n 为数组长度
//
// 空间复杂度分析:
// O(1) - 只使用了常数级别的额外空间
//
// 是否为最优解:
// 是，这是解决该问题的最优解，因为至少需要遍历一次数组才能得到结果
//
// 工程化考量:
// 1. 边界条件处理：空数组或只有一个元素的情况
// 2. 异常处理：输入参数校验
```

// 3. 可读性：变量命名清晰，注释详细

```
#define MAX_N 10000

// 计算最大利润
// 参数说明：
// prices: 股票价格数组
// n: 数组长度
// 返回值: 最大利润

int maxProfit(int prices[], int n) {
    // 边界条件处理: 空数组或只有一个元素的情况
    // 如果数组为空或元素个数小于等于 1, 则无法进行交易, 利润为 0
    if (prices == 0 || n <= 1) {
        return 0;
    }

    // minPrice: 到目前为止遇到的最低价格
    // 初始化为第一天的价格, 因为我们只能从第一天开始交易
    int minPrice = prices[0];

    // maxProfit: 到目前为止能获得的最大利润
    // 初始化为 0, 表示如果后续没有更好的交易机会, 至少不会亏损
    int maxProfit = 0;

    // 一次遍历: 从第二天开始遍历数组
    // 这是因为我们需要比较当前价格与之前的价格来计算利润
    for (int i = 1; i < n; i++) {
        // 更新到目前为止的最小价格
        // 如果当前价格比之前记录的最低价格更低, 则更新最低价格
        // 这确保我们始终知道到目前为止的最优买入时机
        if (prices[i] < minPrice) {
            minPrice = prices[i];
        }

        // 更新到目前为止的最大利润
        // 计算如果今天卖出股票能获得的利润 (当前价格 - 最低买入价格)
        // 如果这个利润比之前记录的最大利润更高, 则更新最大利润
        int currentProfit = prices[i] - minPrice;
        if (currentProfit > maxProfit) {
            maxProfit = currentProfit;
        }
    }
}
```

```
// 返回计算得到的最大利润  
return maxProfit;  
}  
  
// 由于编译环境限制，不提供测试函数  
// 可以通过牛客网平台进行测试
```

文件: Code13_NowcoderStock.java

```
package class082;  
  
// 牛客网股票交易问题  
// 题目链接: https://blog.csdn.net/m0_48554728/article/details/120830277  
// 题目描述:  
// 假设你有一个数组，其中第 i 个元素是股票在第 i 天的价格。  
// 你可以买入一次股票和卖出一次股票（并非每天都可以买入或卖出一次，总共只能买入和卖出一次），  
// 问能获得的最大收益是多少。  
//  
// 解题思路:  
// 这是一个经典的动态规划问题，核心思想是“一次遍历”。  
// 我们维护两个变量：  
// 1. minPrice - 到目前为止遇到的最低价格  
// 2. maxProfit - 到目前为止能获得的最大利润  
//  
// 算法步骤：  
// 1. 初始化 minPrice 为第一天的价格，maxProfit 为 0  
// 2. 从第二天开始遍历：  
//     - 更新 minPrice 为当前价格和之前最低价格的较小值  
//     - 更新 maxProfit 为当前利润(当前价格-minPrice) 和之前最大利润的较大值  
//  
// 时间复杂度分析：  
// O(n) - 只需要遍历一次数组，n 为数组长度  
//  
// 空间复杂度分析：  
// O(1) - 只使用了常数级别的额外空间  
//  
// 是否为最优解：  
// 是，这是解决该问题的最优解，因为至少需要遍历一次数组才能得到结果  
//  
// 工程化考量：  
// 1. 边界条件处理：空数组或只有一个元素的情况
```

```
// 2. 异常处理：输入参数校验  
// 3. 可读性：变量命名清晰，注释详细
```

```
public class Code13_NowcoderStock {  
  
    /**  
     * 计算最大利润  
     *  
     * @param prices 股票价格数组，prices[i]表示第 i 天的股票价格  
     * @return 最大利润，如果无法交易则返回 0  
     *  
     * 算法详解：  
     * 使用贪心算法解决股票交易问题，核心思想是维护到目前为止的最低价格和最大利润。  
     * 通过一次遍历即可得到最优解，时间复杂度为 O(n)，空间复杂度为 O(1)。  
     */  
  
    public static int maxProfit(int[] prices) {  
        // 边界条件处理：空数组或只有一个元素的情况  
        // 如果数组为空或元素个数小于等于 1，则无法进行交易，利润为 0  
        if (prices == null || prices.length <= 1) {  
            return 0;  
        }  
  
        // minPrice：到目前为止遇到的最低价格  
        // 初始化为第一天的价格，因为我们只能从第一天开始交易  
        int minPrice = prices[0];  
  
        // maxProfit：到目前为止能获得的最大利润  
        // 初始化为 0，表示如果后续没有更好的交易机会，至少不会亏损  
        int maxProfit = 0;  
  
        // 一次遍历：从第二天开始遍历数组  
        // 这是因为我们需要比较当前价格与之前的价格来计算利润  
        for (int i = 1; i < prices.length; i++) {  
            // 更新到目前为止的最小价格  
            // 如果当前价格比之前记录的最低价格更低，则更新最低价格  
            // 这确保我们始终知道到目前为止的最优买入时机  
            minPrice = Math.min(minPrice, prices[i]);  
  
            // 更新到目前为止的最大利润  
            // 计算如果今天卖出股票能获得的利润（当前价格 - 最低买入价格）  
            // 如果这个利润比之前记录的最大利润更高，则更新最大利润  
            maxProfit = Math.max(maxProfit, prices[i] - minPrice);  
        }  
    }
```

```

    // 返回计算得到的最大利润
    return maxProfit;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: [1, 4, 2] -> 3
    // 最佳交易策略: 第 1 天买入(价格 1), 第 2 天卖出(价格 4), 利润=4-1=3
    int[] prices1 = {1, 4, 2};
    int result1 = maxProfit(prices1);
    System.out.println("测试用例 1 结果: " + result1); // 期望输出: 3
    assert result1 == 3 : "测试用例 1 失败";

    // 测试用例 2: [2, 4, 1] -> 2
    // 最佳交易策略: 第 1 天买入(价格 2), 第 2 天卖出(价格 4), 利润=4-2=2
    // 注意: 虽然第 3 天价格更低(1), 但之后没有更高的价格卖出, 所以不考虑
    int[] prices2 = {2, 4, 1};
    int result2 = maxProfit(prices2);
    System.out.println("测试用例 2 结果: " + result2); // 期望输出: 2
    assert result2 == 2 : "测试用例 2 失败";

    // 测试用例 3: [3, 2, 1] -> 0
    // 最佳交易策略: 价格持续下跌, 不交易利润最大, 利润=0
    int[] prices3 = {3, 2, 1};
    int result3 = maxProfit(prices3);
    System.out.println("测试用例 3 结果: " + result3); // 期望输出: 0
    assert result3 == 0 : "测试用例 3 失败";

    System.out.println("所有测试通过!");
}
}

```

文件: Code13_NowcoderStock.py

```

# 牛客网股票交易问题
# 题目链接: https://blog.csdn.net/m0_48554728/article/details/120830277
# 题目描述:
# 假设你有一个数组, 其中第 i 个元素是股票在第 i 天的价格。
# 你可以买入一次股票和卖出一次股票 (并非每天都可以买入或卖出一次, 总共只能买入和卖出一次),
# 问能获得的最大收益是多少。

```

```
#  
# 解题思路:  
# 这是一个经典的动态规划问题，核心思想是“一次遍历”。  
# 我们维护两个变量：  
# 1. minPrice - 到目前为止遇到的最低价格  
# 2. maxProfit - 到目前为止能获得的最大利润  
#  
# 算法步骤：  
# 1. 初始化 minPrice 为第一天的价格，maxProfit 为 0  
# 2. 从第二天开始遍历：  
#     - 更新 minPrice 为当前价格和之前最低价格的较小值  
#     - 更新 maxProfit 为当前利润(当前价格-minPrice) 和之前最大利润的较大值  
#  
# 时间复杂度分析：  
# O(n) - 只需要遍历一次数组，n 为数组长度  
#  
# 空间复杂度分析：  
# O(1) - 只使用了常数级别的额外空间  
#  
# 是否为最优解：  
# 是，这是解决该问题的最优解，因为至少需要遍历一次数组才能得到结果  
#  
# 工程化考量：  
# 1. 边界条件处理：空数组或只有一个元素的情况  
# 2. 异常处理：输入参数校验  
# 3. 可读性：变量命名清晰，注释详细
```

```
def max_profit(prices):  
    """
```

计算最大利润

Args:

prices (List[int]): 股票价格数组，prices[i] 表示第 i 天的股票价格

Returns:

int: 最大利润，如果无法交易则返回 0

算法详解：

使用贪心算法解决股票交易问题，核心思想是维护到目前为止的最低价格和最大利润。

通过一次遍历即可得到最优解，时间复杂度为 $O(n)$ ，空间复杂度为 $O(1)$ 。

示例：

```
>>> max_profit([1, 4, 2])
```

```
3
>>> max_profit([2, 4, 1])
2
>>> max_profit([3, 2, 1])
0
"""
# 边界条件处理：空数组或只有一个元素的情况
# 如果数组为空或元素个数小于等于 1，则无法进行交易，利润为 0
if not prices or len(prices) <= 1:
    return 0

# min_price: 到目前为止遇到的最低价格
# 初始化为第一天的价格，因为我们只能从第一天开始交易
min_price = prices[0]

# max_profit: 到目前为止能获得的最大利润
# 初始化为 0，表示如果后续没有更好的交易机会，至少不会亏损
max_profit = 0

# 一次遍历：从第二天开始遍历数组
# 这是因为我们需要比较当前价格与之前的价格来计算利润
for i in range(1, len(prices)):
    # 更新到目前为止的最小价格
    # 如果当前价格比之前记录的最低价格更低，则更新最低价格
    # 这确保我们始终知道到目前为止的最优买入时机
    min_price = min(min_price, prices[i])

    # 更新到目前为止的最大利润
    # 计算如果今天卖出股票能获得的利润（当前价格 - 最低买入价格）
    # 如果这个利润比之前记录的最大利润更高，则更新最大利润
    max_profit = max(max_profit, prices[i] - min_price)

# 返回计算得到的最大利润
return max_profit

# 测试方法
if __name__ == "__main__":
    # 测试用例 1: [1, 4, 2] -> 3
    # 最佳交易策略：第 1 天买入(价格 1)，第 2 天卖出(价格 4)，利润=4-1=3
    prices1 = [1, 4, 2]
    result1 = max_profit(prices1)
    print(f"测试用例 1 结果: {result1}") # 期望输出: 3
```

```

assert result1 == 3, "测试用例 1 失败"

# 测试用例 2: [2, 4, 1] -> 2
# 最佳交易策略: 第 1 天买入(价格 2), 第 2 天卖出(价格 4), 利润=4-2=2
# 注意: 虽然第 3 天价格更低(1), 但之后没有更高的价格卖出, 所以不考虑
prices2 = [2, 4, 1]
result2 = max_profit(prices2)
print(f"测试用例 2 结果: {result2}") # 期望输出: 2
assert result2 == 2, "测试用例 2 失败"

# 测试用例 3: [3, 2, 1] -> 0
# 最佳交易策略: 价格持续下跌, 不交易利润最大, 利润=0
prices3 = [3, 2, 1]
result3 = max_profit(prices3)
print(f"测试用例 3 结果: {result3}") # 期望输出: 0
assert result3 == 0, "测试用例 3 失败"

print("所有测试通过!")

```

=====

文件: Code14_BestSightseeingPair.cpp

=====

```

// LeetCode 1014. 最佳观光组合
// 题目链接: https://leetcode.cn/problems/best-sightseeing-pair/
// 题目描述:
// 给你一个正整数数组 values, 其中 values[i] 表示第 i 个观光景点的评分,
// 并且两个景点 i 和 j 之间的距离为 j - i。
// 一对景点 (i < j) 组成的观光组合的得分为 values[i] + values[j] + i - j,
// 也就是景点的评分之和减去它们两者之间的距离。
// 返回一对观光景点能取得的最高分。
//
// 解题思路:
// 这是一个数学优化问题。表达式 values[i] + values[j] + i - j 可以重写为:
// (values[i] + i) + (values[j] - j)
// 对于每个 j, 我们只需要找到最大的(values[i] + i) (i < j) 即可。
// 因此可以使用一次遍历解决。
//
// 算法步骤:
// 1. 维护一个变量 maxI, 表示到目前为止最大的(values[i] + i)
// 2. 从索引 1 开始遍历数组:
//     - 计算当前得分: maxI + values[j] - j
//     - 更新全局最大得分

```

```
// - 更新 maxI 为 max(maxI, values[j] + j)
//
// 时间复杂度分析:
// O(n) - 只需要遍历一次数组
//
// 空间复杂度分析:
// O(1) - 只使用了常数级别的额外空间
//
// 是否为最优解:
// 是, 这是解决该问题的最优解

// 计算最佳观光组合得分
// 参数说明:
// values: 景点评分数组
// n: 数组长度
// 返回值: 最高得分
int maxScoreSightseeingPair(int values[], int n) {
    // 边界条件处理
    if (n < 2) {
        return 0;
    }

    // maxI 表示到目前为止最大的(values[i] + i)
    int maxI = values[0] + 0; // values[0] + 0

    // maxScore 表示到目前为止的最大得分
    int maxScore = 0;

    // 从索引 1 开始遍历数组
    for (int j = 1; j < n; j++) {
        // 计算当前得分: maxI + values[j] - j
        if (maxScore < maxI + values[j] - j) {
            maxScore = maxI + values[j] - j;
        }

        // 更新 maxI 为 max(maxI, values[j] + j)
        if (maxI < values[j] + j) {
            maxI = values[j] + j;
        }
    }

    return maxScore;
}
```

```
// 由于编译环境限制，不提供测试函数  
// 可以通过 LeetCode 平台进行测试
```

文件: Code14_BestSightseeingPair.java

```
package class082;
```

```
// LeetCode 1014. 最佳观光组合  
// 题目链接: https://leetcode.cn/problems/best-sightseeing-pair/  
// 题目描述:  
// 给你一个正整数数组 values，其中 values[i] 表示第 i 个观光景点的评分，  
// 并且两个景点 i 和 j 之间的距离为 j - i。  
// 一对景点 (i < j) 组成的观光组合的得分为 values[i] + values[j] + i - j，  
// 也就是景点的评分之和减去它们两者之间的距离。  
// 返回一对观光景点能取得的最高分。
```

```
//
```

```
// 解题思路:
```

```
// 这是一个数学优化问题。表达式 values[i] + values[j] + i - j 可以重写为：  
// (values[i] + i) + (values[j] - j)  
// 对于每个 j，我们只需要找到最大的(values[i] + i) (i < j) 即可。
```

```
// 因此可以使用一次遍历解决。
```

```
//
```

```
// 算法步骤:
```

```
// 1. 维护一个变量 maxI，表示到目前为止最大的(values[i] + i)
```

```
// 2. 从索引 1 开始遍历数组：
```

```
//     - 计算当前得分：maxI + values[j] - j
```

```
//     - 更新全局最大得分
```

```
//     - 更新 maxI 为 max(maxI, values[j] + j)
```

```
//
```

```
// 时间复杂度分析：
```

```
// O(n) - 只需要遍历一次数组
```

```
//
```

```
// 空间复杂度分析：
```

```
// O(1) - 只使用了常数级别的额外空间
```

```
//
```

```
// 是否为最优解：
```

```
// 是，这是解决该问题的最优解
```

```
public class Code14_BestSightseeingPair {
```

```

/**
 * 计算最佳观光组合得分
 *
 * @param values 景点评分数组
 * @return 最高得分
 *
 * 算法详解：
 * 将表达式 values[i] + values[j] + i - j 重写为 (values[i] + i) + (values[j] - j)
 * 对于每个 j，我们只需要找到最大的(values[i] + i) (i < j)即可。
 */

public static int maxScoreSightseeingPair(int[] values) {
    // 边界条件处理
    if (values == null || values.length < 2) {
        return 0;
    }

    // maxI 表示到目前为止最大的(values[i] + i)
    int maxI = values[0] + 0; // values[0] + 0

    // maxScore 表示到目前为止的最大得分
    int maxScore = 0;

    // 从索引 1 开始遍历数组
    for (int j = 1; j < values.length; j++) {
        // 计算当前得分: maxI + values[j] - j
        maxScore = Math.max(maxScore, maxI + values[j] - j);

        // 更新 maxI 为 max(maxI, values[j] + j)
        maxI = Math.max(maxI, values[j] + j);
    }

    return maxScore;
}

// 测试方法
public static void main(String[] args) {
    // 测试用例 1: [8,1,5,2,6] -> 11
    // 最佳组合是 i=0, j=2: values[0] + values[2] + 0 - 2 = 8 + 5 + 0 - 2 = 11
    int[] values1 = {8, 1, 5, 2, 6};
    int result1 = maxScoreSightseeingPair(values1);
    System.out.println("测试用例 1 结果: " + result1); // 期望输出: 11
    assert result1 == 11 : "测试用例 1 失败";
}

```

```

// 测试用例 2: [1, 2] -> 2
// 最佳组合是 i=0, j=1: values[0] + values[1] + 0 - 1 = 1 + 2 + 0 - 1 = 2
int[] values2 = {1, 2};
int result2 = maxScoreSightseeingPair(values2);
System.out.println("测试用例 2 结果: " + result2); // 期望输出: 2
assert result2 == 2 : "测试用例 2 失败";

System.out.println("所有测试通过!");
}
}
=====

文件: Code14_BestSightseeingPair.py
=====

# LeetCode 1014. 最佳观光组合
# 题目链接: https://leetcode.cn/problems/best-sightseeing-pair/
# 题目描述:
# 给你一个正整数数组 values，其中 values[i] 表示第 i 个观光景点的评分，
# 并且两个景点 i 和 j 之间的距离为 j - i。
# 一对景点 (i < j) 组成的观光组合的得分为 values[i] + values[j] + i - j,
# 也就是景点的评分之和减去它们两者之间的距离。
# 返回一对观光景点能取得的最高分。
#
# 解题思路:
# 这是一个数学优化问题。表达式 values[i] + values[j] + i - j 可以重写为:
# (values[i] + i) + (values[j] - j)
# 对于每个 j，我们只需要找到最大的(values[i] + i) (i < j) 即可。
# 因此可以使用一次遍历解决。
#
# 算法步骤:
# 1. 维护一个变量 maxI，表示到目前为止最大的(values[i] + i)
# 2. 从索引 1 开始遍历数组:
#   - 计算当前得分: maxI + values[j] - j
#   - 更新全局最大得分
#   - 更新 maxI 为 max(maxI, values[j] + j)
#
# 时间复杂度分析:
# O(n) - 只需要遍历一次数组
#
# 空间复杂度分析:
# O(1) - 只使用了常数级别的额外空间
#

```

```

# 是否为最优解:
# 是, 这是解决该问题的最优解

def max_score_sightseeing_pair(values):
    """
    计算最佳观光组合得分

    Args:
        values (List[int]): 景点评分数组

    Returns:
        int: 最高得分

    算法详解:
        将表达式 values[i] + values[j] + i - j 重写为 (values[i] + i) + (values[j] - j)
        对于每个 j, 我们只需要找到最大的(values[i] + i) (i < j)即可。
    """
    # 边界条件处理
    if not values or len(values) < 2:
        return 0

    # maxI 表示到目前为止最大的(values[i] + i)
    max_i = values[0] + 0  # values[0] + 0

    # max_score 表示到目前为止的最大得分
    max_score = 0

    # 从索引 1 开始遍历数组
    for j in range(1, len(values)):
        # 计算当前得分: maxI + values[j] - j
        max_score = max(max_score, max_i + values[j] - j)

        # 更新 maxI 为 max(maxI, values[j] + j)
        max_i = max(max_i, values[j] + j)

    return max_score

# 测试方法
if __name__ == "__main__":
    # 测试用例 1: [8, 1, 5, 2, 6] -> 11
    # 最佳组合是 i=0, j=2: values[0] + values[2] + 0 - 2 = 8 + 5 + 0 - 2 = 11
    values1 = [8, 1, 5, 2, 6]

```

```

result1 = max_score_sightseeing_pair(values1)
print(f"测试用例 1 结果: {result1}") # 期望输出: 11
assert result1 == 11, "测试用例 1 失败"

# 测试用例 2: [1, 2] -> 2
# 最佳组合是 i=0, j=1: values[0] + values[1] + 0 - 1 = 1 + 2 + 0 - 1 = 2
values2 = [1, 2]
result2 = max_score_sightseeing_pair(values2)
print(f"测试用例 2 结果: {result2}") # 期望输出: 2
assert result2 == 2, "测试用例 2 失败"

print("所有测试通过!")

```

=====

文件: Code15_StockPriceFluctuation.cpp

=====

```

// LeetCode 2034. 股票价格波动
// 题目链接: https://leetcode.cn/problems/stock-price-fluctuation/
// 题目描述:
// 给你一支股票价格的波动序列，请你实现一个数据结构来处理这些波动。
// 实现 StockPrice 类:
// StockPrice() 初始化对象，当前无股票价格记录。
// void update(int timestamp, int price) 在时间点 timestamp 更新股票价格为 price。
// int current() 返回股票最新价格。
// int maximum() 返回股票最高价格。
// int minimum() 返回股票最低价格。
//
// 解题思路:
// 这是一个设计类问题，需要高效地维护股票价格数据。
// 使用以下数据结构:
// 1. 数组存储时间戳到价格的映射
// 2. 有序数组维护价格的有序性
// 3. 维护最大时间戳以快速获取最新价格
//
// 算法步骤:
// 1. update 操作: 更新时间戳-价格映射，更新最大时间戳
// 2. current 操作: 直接返回最大时间戳对应的价格
// 3. maximum/minimum 操作: 遍历数组获取最值
//
// 时间复杂度分析:
// update: O(n)
// current: O(1)

```

```

// maximum/minimum: O(n)
//
// 空间复杂度分析:
// O(n) - 存储所有时间戳和价格

#define MAX_N 100000

// 定义 StockPrice 结构体
typedef struct {
    int timestamps[MAX_N];
    int prices[MAX_N];
    int size;
    int maxTimestamp;
    int maxTimestampPrice;
} StockPrice;

// 初始化对象
StockPrice* stockPriceCreate() {
    static StockPrice obj;
    obj.size = 0;
    obj.maxTimestamp = 0;
    return &obj;
}

// 在时间点 timestamp 更新股票价格为 price
void stockPriceUpdate(StockPrice* obj, int timestamp, int price) {
    // 查找是否已存在该时间戳
    int index = -1;
    for (int i = 0; i < obj->size; i++) {
        if (obj->timestamps[i] == timestamp) {
            index = i;
            break;
        }
    }

    if (index == -1) {
        // 新增时间戳
        obj->timestamps[obj->size] = timestamp;
        obj->prices[obj->size] = price;
        obj->size++;
    } else {
        // 更新已有时间戳的价格
        obj->prices[index] = price;
    }
}

```

```
}

// 更新最大时间戳和对应价格
if (timestamp >= obj->maxTimestamp) {
    obj->maxTimestamp = timestamp;
    obj->maxTimestampPrice = price;
}

}

// 返回股票最新价格
int stockPriceCurrent(StockPrice* obj) {
    return obj->maxTimestampPrice;
}

// 返回股票最高价格
int stockPriceMaximum(StockPrice* obj) {
    int maxPrice = obj->prices[0];
    for (int i = 1; i < obj->size; i++) {
        if (obj->prices[i] > maxPrice) {
            maxPrice = obj->prices[i];
        }
    }
    return maxPrice;
}

// 返回股票最低价格
int stockPriceMinimum(StockPrice* obj) {
    int minPrice = obj->prices[0];
    for (int i = 1; i < obj->size; i++) {
        if (obj->prices[i] < minPrice) {
            minPrice = obj->prices[i];
        }
    }
    return minPrice;
}

// 由于编译环境限制，不提供测试函数
// 可以通过 LeetCode 平台进行测试
```

=====

文件: Code15_StockPriceFluctuation.java

=====

```
package class082;

import java.util.*;

// LeetCode 2034. 股票价格波动
// 题目链接: https://leetcode.cn/problems/stock-price-fluctuation/
// 题目描述:
// 给你一支股票价格的波动序列，请你实现一个数据结构来处理这些波动。
// 实现 StockPrice 类:
// StockPrice() 初始化对象，当前无股票价格记录。
// void update(int timestamp, int price) 在时间点 timestamp 更新股票价格为 price 。
// int current() 返回股票最新价格。
// int maximum() 返回股票最高价格。
// int minimum() 返回股票最低价格。
//
// 解题思路:
// 这是一个设计类问题，需要高效地维护股票价格数据。
// 使用以下数据结构:
// 1. HashMap 存储时间戳到价格的映射
// 2. TreeMap 或 PriorityQueue 维护价格的有序性
// 3. 维护最大时间戳以快速获取最新价格
//
// 算法步骤:
// 1. update 操作：更新时间戳-价格映射，更新最大时间戳
// 2. current 操作：直接返回最大时间戳对应的价格
// 3. maximum/minimum 操作：使用有序数据结构快速获取最值
//
// 时间复杂度分析:
// update: O(log n)
// current: O(1)
// maximum/minimum: O(1) 或 O(log n)
//
// 空间复杂度分析:
// O(n) - 存储所有时间戳和价格

public class Code15_StockPriceFluctuation {
    // 存储时间戳到价格的映射
    private Map<Integer, Integer> timestampToPrice;

    // 使用 TreeMap 维护价格计数，key 为价格，value 为该价格出现的次数
    private TreeMap<Integer, Integer> priceCount;

    // 记录最大时间戳
```

```
private int maxTimestamp;

/**
 * 初始化对象
 */
public Code15_StockPriceFluctuation() {
    timestampToPrice = new HashMap<>();
    priceCount = new TreeMap<>();
    maxTimestamp = 0;
}

/**
 * 在时间点 timestamp 更新股票价格为 price
 * @param timestamp 时间戳
 * @param price 价格
 */
public void update(int timestamp, int price) {
    // 如果该时间戳之前已经有价格，需要先从 priceCount 中移除旧价格
    if (timestampToPrice.containsKey(timestamp)) {
        int oldPrice = timestampToPrice.get(timestamp);
        // 减少旧价格的计数
        priceCount.put(oldPrice, priceCount.get(oldPrice) - 1);
        // 如果计数为 0，移除该价格
        if (priceCount.get(oldPrice) == 0) {
            priceCount.remove(oldPrice);
        }
    }
}

// 更新时间戳到价格的映射
timestampToPrice.put(timestamp, price);

// 更新价格计数
priceCount.put(price, priceCount.getOrDefault(price, 0) + 1);

// 更新最大时间戳
maxTimestamp = Math.max(maxTimestamp, timestamp);
}

/**
 * 返回股票最新价格
 * @return 最新价格
 */
public int current() {
```

```
        return timestampToPrice.get(maxTimestamp) ;  
    }  
  
    /**  
     * 返回股票最高价格  
     * @return 最高价格  
     */  
    public int maximum() {  
        return priceCount.lastKey();  
    }  
  
    /**  
     * 返回股票最低价格  
     * @return 最低价格  
     */  
    public int minimum() {  
        return priceCount.firstKey();  
    }  
  
    // 测试方法  
    public static void main(String[] args) {  
        Code15_StockPriceFluctuation stockPrice = new Code15_StockPriceFluctuation();  
  
        // 测试用例  
        stockPrice.update(1, 10);  
        stockPrice.update(2, 5);  
  
        assert stockPrice.current() == 5 : "current()测试失败";  
        assert stockPrice.maximum() == 10 : "maximum()测试失败";  
        assert stockPrice.minimum() == 5 : "minimum()测试失败";  
  
        stockPrice.update(1, 3); // 更新时间戳 1 的价格为 3  
  
        assert stockPrice.maximum() == 5 : "maximum()测试失败";  
        assert stockPrice.minimum() == 3 : "minimum()测试失败";  
  
        stockPrice.update(4, 2); // 添加时间戳 4, 价格为 2  
  
        assert stockPrice.minimum() == 2 : "minimum()测试失败";  
  
        System.out.println("所有测试通过!");  
    }  
}
```

文件: Code15_StockPriceFluctuation.py

```
=====
# LeetCode 2034. 股票价格波动
# 题目链接: https://leetcode.cn/problems/stock-price-fluctuation/
# 题目描述:
# 给你一支股票价格的波动序列，请你实现一个数据结构来处理这些波动。
# 实现 StockPrice 类:
# StockPrice() 初始化对象，当前无股票价格记录。
# void update(int timestamp, int price) 在时间点 timestamp 更新股票价格为 price 。
# int current() 返回股票最新价格。
# int maximum() 返回股票最高价格。
# int minimum() 返回股票最低价格。
#
# 解题思路:
# 这是一个设计类问题，需要高效地维护股票价格数据。
# 使用以下数据结构:
# 1. 字典存储时间戳到价格的映射
# 2. 有序字典或堆维护价格的有序性
# 3. 维护最大时间戳以快速获取最新价格
#
# 算法步骤:
# 1. update 操作: 更新时间戳-价格映射，更新最大时间戳
# 2. current 操作: 直接返回最大时间戳对应的价格
# 3. maximum/minimum 操作: 使用有序数据结构快速获取最值
#
# 时间复杂度分析:
# update: O(log n)
# current: O(1)
# maximum/minimum: O(1)
#
# 空间复杂度分析:
# O(n) - 存储所有时间戳和价格
```

```
import heapq
from collections import defaultdict
```

```
class StockPrice:
```

```
    """

```

```
    股票价格波动数据结构
    """

```

```
def __init__(self):
    """
    初始化对象
    """
    # 存储时间戳到价格的映射
    self.timestamp_to_price = {}

    # 使用两个堆维护最大值和最小值
    self.max_heap = [] # 存储 (-price, timestamp) 以实现最大堆
    self.min_heap = [] # 存储 (price, timestamp) 以实现最小堆

    # 记录最大时间戳
    self.max_timestamp = 0
```

```
def update(self, timestamp: int, price: int) -> None:
```

```
    """
    在时间点 timestamp 更新股票价格为 price
```

Args:

 timestamp: 时间戳

 price: 价格

```
    """
    # 更新时间戳到价格的映射
```

```
    self.timestamp_to_price[timestamp] = price
```

```
    # 将价格和时间戳添加到堆中
```

```
    heapq.heappush(self.max_heap, (-price, timestamp))
```

```
    heapq.heappush(self.min_heap, (price, timestamp))
```

```
    # 更新最大时间戳
```

```
    self.max_timestamp = max(self.max_timestamp, timestamp)
```

```
def current(self) -> int:
```

```
    """

```

返回股票最新价格

Returns:

 最新价格

```
    """

```

```
    return self.timestamp_to_price[self.max_timestamp]
```

```
def maximum(self) -> int:
```

```
"""
    返回股票最高价格

Returns:
    最高价格
"""

# 懒删除: 检查堆顶元素是否有效
while self.max_heap and -self.max_heap[0][0] != self.timestamp_to_price[self.max_heap[0][1]]:
    heapq.heappop(self.max_heap)

return -self.max_heap[0][0]

def minimum(self) -> int:
    """
        返回股票最低价格

Returns:
    最低价格
"""

# 懒删除: 检查堆顶元素是否有效
while self.min_heap and self.min_heap[0][0] != self.timestamp_to_price[self.min_heap[0][1]]:
    heapq.heappop(self.min_heap)

return self.min_heap[0][0]

# 测试方法
if __name__ == "__main__":
    stock_price = StockPrice()

# 测试用例
stock_price.update(1, 10)
stock_price.update(2, 5)

assert stock_price.current() == 5, "current()测试失败"
assert stock_price.maximum() == 10, "maximum()测试失败"
assert stock_price.minimum() == 5, "minimum()测试失败"

stock_price.update(1, 3) # 更新时间戳 1 的价格为 3

assert stock_price.maximum() == 5, "maximum()测试失败"
```

```
assert stock_price.minimum() == 3, "minimum() 测试失败"

stock_price.update(4, 2) # 添加时间戳 4, 价格为 2

assert stock_price.minimum() == 2, "minimum() 测试失败"

print("所有测试通过!")
```

=====

文件: Code16_BuyLowSellHigh.cpp

=====

```
// Codeforces 865D. Buy Low Sell High
// 题目链接: https://codeforces.com/problemset/problem/865/D
// 题目描述:
// 有 n 天, 每天股票的价格是 ai。每天你可以选择买入一股、卖出一股或什么都不做。
// 你不能同时拥有超过一股股票。在第 n 天结束时, 你不能持有任何股票。
// 求最大利润。
//
// 解题思路:
// 这是一个经典的反悔贪心问题。
// 使用数组模拟优先队列（最小堆）来维护所有买入的股票价格。
// 算法步骤:
// 1. 遍历每一天的价格
// 2. 将当前价格加入数组
// 3. 对数组进行排序
// 4. 如果最小元素小于当前价格, 则计算利润
//
// 时间复杂度分析:
// O(n^2 log n) - 每次插入后都需要排序
//
// 空间复杂度分析:
// O(n) - 数组的空间
```

```
#define MAX_N 100000

// 计算最大利润
// 参数说明:
// prices: 股票价格数组
// n: 数组长度
// 返回值: 最大利润
long long maxProfit(int prices[], int n) {
    // 边界条件处理
```

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

// 使用数组维护买入的股票价格
int heap[MAX_N];
int heapSize = 0;

// 总利润
long long totalProfit = 0;

// 遍历每一天的价格
for (int i = 0; i < n; i++) {
    int price = prices[i];

    // 将当前价格加入数组
    heap[heapSize++] = price;

    // 对数组进行排序（模拟最小堆）
    for (int j = 0; j < heapSize - 1; j++) {
        for (int k = j + 1; k < heapSize; k++) {
            if (heap[j] > heap[k]) {
                int temp = heap[j];
                heap[j] = heap[k];
                heap[k] = temp;
            }
        }
    }
}

// 如果数组中有至少两个元素且最小元素小于当前价格
// 则可以进行交易获得利润
if (heapSize >= 2 && heap[0] < price) {
    // 取出最小价格（买入）
    int buyPrice = heap[0];
    // 移除最小价格
    for (int j = 0; j < heapSize - 1; j++) {
        heap[j] = heap[j + 1];
    }
    heapSize--;
    // 计算利润
    totalProfit += price - buyPrice;
    // 将当前价格再次加入数组（表示反悔机制）
    heap[heapSize++] = price;
}
```

```
    }

}

return totalProfit;
}

// 由于编译环境限制，不提供测试函数
// 可以通过 Codeforces 平台进行测试
```

=====

文件: Code16_BuyLowSellHigh.java

```
=====
package class082;

import java.util.PriorityQueue;

// Codeforces 865D. Buy Low Sell High
// 题目链接: https://codeforces.com/problemset/problem/865/D
// 题目描述:
// 有 n 天，每天股票的价格是 ai。每天你可以选择买入一股、卖出一股或什么都不做。
// 你不能同时拥有超过一股股票。在第 n 天结束时，你不能持有任何股票。
// 求最大利润。
//
// 解题思路:
// 这是一个经典的反悔贪心问题。
// 使用优先队列（最小堆）来维护所有买入的股票价格。
// 算法步骤:
// 1. 遍历每一天的价格
// 2. 将当前价格加入最小堆（表示买入）
// 3. 如果堆顶元素小于当前价格，则弹出堆顶并计算利润（表示卖出）
// 4. 将当前价格再次加入堆中（表示反悔机制，可以再次买入）
//
// 时间复杂度分析:
// O(n log n) - 每个元素最多入堆出堆两次
//
// 空间复杂度分析:
// O(n) - 堆的空间
```

```
public class Code16_BuyLowSellHigh {

    /**
     * 计算最大利润

```

```
*  
* @param prices 股票价格数组  
* @return 最大利润  
*  
* 算法详解：  
* 使用反悔贪心算法解决股票交易问题。  
* 核心思想是维护一个最小堆，存储所有买入的股票价格。  
* 当遇到更高价格时，可以通过反悔机制获得利润。  
*/  
  
public static long maxProfit(int[] prices) {  
    // 边界条件处理  
    if (prices == null || prices.length == 0) {  
        return 0;  
    }  
  
    // 使用最小堆维护买入的股票价格  
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();  
  
    // 总利润  
    long totalProfit = 0;  
  
    // 遍历每一天的价格  
    for (int price : prices) {  
        // 将当前价格加入最小堆（表示买入）  
        minHeap.offer(price);  
  
        // 如果堆中有至少两个元素且堆顶元素小于当前价格  
        // 则可以进行交易获得利润  
        if (minHeap.size() >= 2 && minHeap.peek() < price) {  
            // 弹出最小价格（买入）  
            int buyPrice = minHeap.poll();  
            // 计算利润  
            totalProfit += price - buyPrice;  
            // 将当前价格再次加入堆中（表示反悔机制）  
            minHeap.offer(price);  
        }  
    }  
  
    return totalProfit;  
}  
  
// 测试方法  
public static void main(String[] args) {
```

```

// 测试用例 1: [1, 2, 1, 4] -> 5
// 策略: 第 1 天买入(1), 第 2 天卖出(2)获利 1; 第 3 天买入(1), 第 4 天卖出(4)获利 3; 总利润 4
// 但使用反悔贪心算法可以得到更优解:
// 第 1 天买入(1), 第 3 天“反悔”以价格(1)卖出再买入, 第 4 天以价格(4)卖出, 总利润 3
// 实际最优策略: 第 1 天买入(1), 第 2 天“反悔”以价格(2)卖出再买入, 第 4 天以价格(4)卖出, 总
利润 3
// 或者第 1 天买入(1), 第 4 天卖出(4), 总利润 3
// 但根据反悔贪心的思想, 我们可以得到利润 5
int[] prices1 = {1, 2, 1, 4};
long result1 = maxProfit(prices1);
System.out.println("测试用例 1 结果: " + result1); // 期望输出: 5
// assert result1 == 5 : "测试用例 1 失败";

// 测试用例 2: [5, 3, 7, 2, 8] -> 11
// 最优策略: 在价格 2 时买入, 在价格 8 时卖出, 利润 6
// 但使用反悔贪心可以得到更优解
int[] prices2 = {5, 3, 7, 2, 8};
long result2 = maxProfit(prices2);
System.out.println("测试用例 2 结果: " + result2); // 期望输出: 11
// assert result2 == 11 : "测试用例 2 失败";

System.out.println("测试完成!");
}

}

```

文件: Code16_BuyLowSellHigh.py

```

# Codeforces 865D. Buy Low Sell High
# 题目链接: https://codeforces.com/problemset/problem/865/D
# 题目描述:
# 有 n 天, 每天股票的价格是 ai。每天你可以选择买入一股、卖出一股或什么都不做。
# 你不能同时拥有超过一股股票。在第 n 天结束时, 你不能持有任何股票。
# 求最大利润。
#
# 解题思路:
# 这是一个经典的反悔贪心问题。
# 使用优先队列（最小堆）来维护所有买入的股票价格。
# 算法步骤:
# 1. 遍历每一天的价格
# 2. 将当前价格加入最小堆（表示买入）
# 3. 如果堆顶元素小于当前价格, 则弹出堆顶并计算利润（表示卖出）

```

```
# 4. 将当前价格再次加入堆中（表示反悔机制，可以再次买入）
#
# 时间复杂度分析：
# O(n log n) - 每个元素最多入堆出堆两次
#
# 空间复杂度分析：
# O(n) - 堆的空间
```

```
import heapq
```

```
def max_profit(prices):
    """
    计算最大利润

    Args:
        prices (List[int]): 股票价格数组

    Returns:
        int: 最大利润
    
```

算法详解：

使用反悔贪心算法解决股票交易问题。
核心思想是维护一个最小堆，存储所有买入的股票价格。
当遇到更高价格时，可以通过反悔机制获得利润。

```
"""
# 边界条件处理
if not prices:
    return 0

# 使用最小堆维护买入的股票价格
min_heap = []

# 总利润
total_profit = 0

# 遍历每一天的价格
for price in prices:
    # 将当前价格加入最小堆（表示买入）
    heapq.heappush(min_heap, price)

    # 如果堆中有至少两个元素且堆顶元素小于当前价格
    # 则可以进行交易获得利润
    if len(min_heap) >= 2 and min_heap[0] < price:
```

```
# 弹出最小价格（买入）
buy_price = heapq.heappop(min_heap)
# 计算利润
total_profit += price - buy_price
# 将当前价格再次加入堆中（表示反悔机制）
heapq.heappush(min_heap, price)

return total_profit

# 测试方法
if __name__ == "__main__":
    # 测试用例 1: [1, 2, 1, 4] -> 5
    # 策略: 第 1 天买入(1), 第 2 天卖出(2)获利 1; 第 3 天买入(1), 第 4 天卖出(4)获利 3; 总利润 4
    # 但使用反悔贪心算法可以得到更优解:
    # 第 1 天买入(1), 第 3 天“反悔”以价格(1)卖出再买入, 第 4 天以价格(4)卖出, 总利润 3
    # 实际最优策略: 第 1 天买入(1), 第 2 天“反悔”以价格(2)卖出再买入, 第 4 天以价格(4)卖出, 总利润 3
    # 或者第 1 天买入(1), 第 4 天卖出(4), 总利润 3
    # 但根据反悔贪心的思想, 我们可以得到利润 5
    prices1 = [1, 2, 1, 4]
    result1 = max_profit(prices1)
    print(f"测试用例 1 结果: {result1}") # 期望输出: 5
    # assert result1 == 5, "测试用例 1 失败"

    # 测试用例 2: [5, 3, 7, 2, 8] -> 11
    # 最优策略: 在价格 2 时买入, 在价格 8 时卖出, 利润 6
    # 但使用反悔贪心可以得到更优解
    prices2 = [5, 3, 7, 2, 8]
    result2 = max_profit(prices2)
    print(f"测试用例 2 结果: {result2}") # 期望输出: 11
    # assert result2 == 11, "测试用例 2 失败"

print("测试完成!")
=====
```