

=====

文件夹: class072\_PrefixSumAlgorithms

=====

[Markdown 文件]

=====

文件: README.md

=====

# 前缀和算法详解与题目扩展

## 算法简介

前缀和 (Prefix Sum) 是一种预处理技术，通过预先计算数组的累积和，可以在常数时间内回答区间和查询问题。它是解决子数组和相关问题的重要工具。

#### 基本思想

对于数组 `nums`，其前缀和数组 `prefix` 定义为:

```

$\text{prefix}[i] = \text{nums}[0] + \text{nums}[1] + \dots + \text{nums}[i]$

```

或者使用偏移形式:

```

$\text{prefix}[0] = 0$

$\text{prefix}[i] = \text{nums}[0] + \text{nums}[1] + \dots + \text{nums}[i-1]$

```

### 应用场景

1. 快速计算区间和:  $\text{sum}(i, j) = \text{prefix}[j+1] - \text{prefix}[i]$
2. 子数组和相关问题
3. 结合哈希表解决特定和的问题

## 已实现题目列表 (已优化)

### 基础前缀和题目

1. [一维数组的动态和] (#扩展题目 1-一维数组的动态和) – LeetCode 1480
  - [Java 实现] (Code01\_RunningSumOf1DArray.java)
  - [Python 实现] (Code01\_RunningSumOf1DArray.py)
  - [C++实现] (Code01\_RunningSumOf1DArray.cpp)
2. [和为 K 的子数组] (#扩展题目 4-和为 k 的子数组个数) – LeetCode 560

- [Java 实现] (Code02\_SubarraySumEqualsK. java) ✓
- [Python 实现] (Code02\_SubarraySumEqualsK. py) ✓
- [C++实现] (Code02\_SubarraySumEqualsK. cpp) ✓

### 3. [连续数组] (#扩展题目 6-连续数组) - LeetCode 525

- [Java 实现] (Code03\_ContiguousArray. java) ✓
- [Python 实现] (Code03\_ContiguousArray. py) ✓
- [C++实现] (Code03\_ContiguousArray. cpp) ✓

### 4. [和可被 K 整除的子数组] (#扩展题目 10-和可被 k 整除的子数组) - LeetCode 974

- [Java 实现] (Code04\_SubarraySumsDivisibleByK. java) ✓
- [Python 实现] (Code04\_SubarraySumsDivisibleByK. py) ✓
- [C++实现] (Code04\_SubarraySumsDivisibleByK. cpp) ✓

### 5. [除自身以外数组的乘积] (#扩展题目 8-除自身以外数组的乘积) - LeetCode 238

- [Java 实现] (Code05\_ProductOfArrayExceptSelf. java) ✓
- [Python 实现] (Code05\_ProductOfArrayExceptSelf. py) ✓
- [C++实现] (Code05\_ProductOfArrayExceptSelf. cpp) ✓

### 6. [数组操作] (#题目 8-数组操作) - HackerRank

- [Java 实现] (Code08\_ArrayManipulation. java) ✓
- [Python 实现] (Code08\_ArrayManipulation. py) ✓
- [C++实现] (Code08\_ArrayManipulation. cpp) ✓

### 7. [二维区域和检索 - 矩阵不可变] (#题目 9-二维区域和检索---矩阵不可变) - LeetCode 304

- [Java 实现] (Code09\_RangeSumQuery2D. java) ✓
- [Python 实现] (Code09\_RangeSumQuery2D. py) ✓
- [C++实现] (Code09\_RangeSumQuery2D. cpp) ✓

## ### 扩展优化特性

所有已优化的文件都包含以下特性:

- ✓ 详细的中文注释和文档说明
- ✓ 完整的时间复杂度和空间复杂度分析
- ✓ 单元测试和边界条件测试
- ✓ 性能测试和优化建议
- ✓ 工程化考量和异常处理
- ✓ 调试技巧和语言特性差异分析

## ## 扩展题目列表

以下是我们将要实现的扩展题目:

## ### 基础前缀和题目

1. [一维数组的动态和] (#扩展题目 1-一维数组的动态和) - LeetCode 1480
2. [区间加法] (#扩展题目 2-区间加法) - LeetCode 370
3. [二维区域和检索] (#扩展题目 3-二维区域和检索) - LeetCode 304

#### #### 哈希表 + 前缀和题目

4. [和为 K 的子数组个数] (#扩展题目 4-和为 k 的子数组个数) - LeetCode 560
5. [最长子数组和等于 K] (#扩展题目 5-最长子数组和等于 k) - LeetCode 325
6. [连续数组] (#扩展题目 6-连续数组) - LeetCode 525
7. [找到和为零的子数组] (#扩展题目 7-找到和为零的子数组) - LintCode 138

#### #### 特殊技巧题目

8. [除自身以外数组的乘积] (#扩展题目 8-除自身以外数组的乘积) - LeetCode 238
9. [统计优美子数组] (#扩展题目 9-统计优美子数组) - LeetCode 1248
10. [和可被 K 整除的子数组] (#扩展题目 10-和可被 k 整除的子数组) - LeetCode 974

### ## 题目详解

#### #### 题目 1：区域和检索 – 数组不可变

##### \*\*题目描述\*\*：

给定一个整数数组 `nums`, 处理以下类型的多个查询:

计算索引 `left` 和 `right` (包含 `left` 和 `right`) 之间的 `nums` 元素的和, 其中 `left <= right`。

##### \*\*实现类\*\*： [Code01\_PrefixSumArray. java] (Code01\_PrefixSumArray. java)

#### #### 题目 2：和为 K 的子数组

##### \*\*题目描述\*\*：

给定一个整数数组和一个整数 `k`, 你需要找到该数组中和为 `k` 的连续的子数组的个数。

##### \*\*实现类\*\*： [Code03\_NumberOfSubarraySumEqualsAim. java] (Code03\_NumberOfSubarraySumEqualsAim. java)

#### #### 题目 3：最长子数组和等于给定值

##### \*\*题目描述\*\*：

给定一个无序数组 `arr`, 其中元素可正、可负、可 0, 给定一个整数 `aim`, 求 `arr` 所有子数组中累加和为 `aim` 的最长子数组长度。

##### \*\*实现类\*\*： [Code02\_LongestSubarraySumEqualsAim. java] (Code02\_LongestSubarraySumEqualsAim. java)

#### #### 题目 4：正负数个数相等的最长子数组

##### \*\*题目描述\*\*：

给定一个无序数组 arr，其中元素可正、可负、可 0，求 arr 所有子数组中正数与负数个数相等的最长子数组的长度。

**\*\*实现类\*\*:**

[Code04\_PositivesEqualsNegativesLongestSubarray. java] (Code04\_PositivesEqualsNegativesLongestSubarray. java)

### 题目 5：表现良好的最长时间段

**\*\*题目描述\*\*:**

给你一份工作时间表 hours，上面记录着某一位员工每天的工作小时数。我们认为当员工一天中的工作小时数大于 8 小时的时候，那么这一天就是劳累的一天。所谓表现良好的时间段，意味在这段时间内，「劳累的天数」是严格大于不劳累的天数。请你返回表现良好时间段的最大长度。

**\*\*实现类\*\*:**

[Code05\_LongestWellPerformingInterval. java] (Code05\_LongestWellPerformingInterval. java)

### 题目 6：使数组和能被 P 整除

**\*\*题目描述\*\*:**

给你一个正整数数组 nums，请你移除最短子数组（可以为空），使得剩余元素的和能被 p 整除。不允许将整个数组都移除。请你返回你需要移除的最短子数组的长度，如果无法满足题目要求，返回 -1。

**\*\*实现类\*\*:** [Code06\_MakeSumDivisibleByP. java] (Code06\_MakeSumDivisibleByP. java)

### 题目 7：每个元音包含偶数次的最长子字符串

**\*\*题目描述\*\*:**

给你一个字符串 s，请你返回满足以下条件的最长子字符串的长度：每个元音字母，即 'a'，'e'，'i'，'o'，'u'，在子字符串中都恰好出现了偶数次。

**\*\*实现类\*\*:** [Code07\_EvenCountsLongestSubarray. java] (Code07\_EvenCountsLongestSubarray. java)

### 题目 13：找到数组中心索引

**\*\*题目描述\*\*:**

给你一个整数数组 nums，请编写一个能够返回数组“中心索引”的方法。

中心索引是数组的一个索引，其左侧所有元素相加的和等于右侧所有元素相加的和。

如果数组不存在中心索引，返回 -1。如果数组有多个中心索引，应该返回最靠近左边的那一个。

**\*\*实现类\*\*:**

- [Code13\_FindPivotIndex. java] (Code13\_FindPivotIndex. java)
- [Code13\_FindPivotIndex. py] (Code13\_FindPivotIndex. py)

- [Code13\_FindPivotIndex. cpp] (Code13\_FindPivotIndex. cpp)

#### #### 题目 14: 连续的子数组和

##### \*\*题目描述\*\*:

给你一个整数数组 `nums` 和一个整数 `k`，编写一个函数来判断该数组是否含有同时满足下述条件的连续子数组：

1. 子数组大小至少为 2
2. 子数组元素总和为 `k` 的倍数

##### \*\*实现类\*\*:

- [Code14\_ContinuousSubarraySum. java] (Code14\_ContinuousSubarraySum. java)
- [Code14\_ContinuousSubarraySum. py] (Code14\_ContinuousSubarraySum. py)
- [Code14\_ContinuousSubarraySum. cpp] (Code14\_ContinuousSubarraySum. cpp)

#### #### 题目 15: 区间和查询 - 不可变

##### \*\*题目描述\*\*:

给定一个整数数组 `nums`，计算索引 `left` 和 `right`（包含 `left` 和 `right`）之间的元素的和，其中 `left <= right`。

##### \*\*实现类\*\*:

- [Code15\_RangeSumQueryImmutable. java] (Code15\_RangeSumQueryImmutable. java)
- [Code15\_RangeSumQueryImmutable. py] (Code15\_RangeSumQueryImmutable. py)
- [Code15\_RangeSumQueryImmutable. cpp] (Code15\_RangeSumQueryImmutable. cpp)

#### #### 题目 16: 二维区域和检索 - 不可变

##### \*\*题目描述\*\*:

给定一个二维矩阵 `matrix`，计算其子矩形范围内元素的总和，该子矩形的左上角为 `(row1, col1)`，右下角为 `(row2, col2)`。

##### \*\*实现类\*\*:

- [Code16\_RangeSumQuery2DImmutable. java] (Code16\_RangeSumQuery2DImmutable. java)
- [Code16\_RangeSumQuery2DImmutable. py] (Code16\_RangeSumQuery2DImmutable. py)
- [Code16\_RangeSumQuery2DImmutable. cpp] (Code16\_RangeSumQuery2DImmutable. cpp)

#### ## 扩展题目实现（已优化）

#### #### 扩展题目 1: 一维数组的动态和

\*\*题目链接\*\*: <https://leetcode.com/problems/running-sum-of-1d-array/>

### \*\*题目描述\*\*:

给你一个数组 `nums`。数组「动态和」的计算公式为:  $\text{runningSum}[i] = \text{sum}(\text{nums}[0] \dots \text{nums}[i])$ 。请返回 `nums` 的动态和。

### \*\*解题思路\*\*:

使用基础前缀和思想，从前向后累加即可。

**\*\*时间复杂度\*\*:**  $O(n)$  – 需要遍历数组一次

**\*\*空间复杂度\*\*:**  $O(1)$  – 如果不算输出数组，原地修改则为  $O(1)$

### \*\*工程化考量\*\*:

- 边界条件处理：空数组、单元素数组
- 原地修改优化：避免额外空间使用
- 性能优化：一次遍历完成计算

### \*\*实现文件\*\*:

- [Code01\_RunningSumOf1DArray.java] (Code01\_RunningSumOf1DArray.java)
- [Code01\_RunningSumOf1DArray.py] (Code01\_RunningSumOf1DArray.py)
- [Code01\_RunningSumOf1DArray.cpp] (Code01\_RunningSumOf1DArray.cpp)

### \*\*代码特性\*\*:

- 包含详细的单元测试和性能测试
- 支持多种边界条件测试
- 提供调试技巧和优化建议

## ### 扩展题目 2：区间加法

**\*\*题目链接\*\*:** <https://leetcode.com/problems/range-addition/>

### \*\*题目描述\*\*:

假设你有一个长度为  $n$  的数组，初始情况下所有的数字均为 0，你将会被给出  $k$  个更新操作。其中，每个操作会被表示为一个三元组:  $[\text{startIndex}, \text{endIndex}, \text{inc}]$ ，你需要将子数组  $A[\text{startIndex} \dots \text{endIndex}]$  (包括  $\text{startIndex}$  和  $\text{endIndex}$ ) 增加  $\text{inc}$ 。请你返回  $k$  次操作后的数组。

### \*\*解题思路\*\*:

使用差分数组技巧，对区间进行标记，最后通过前缀和还原结果。

## ### 扩展题目 3：二维区域和检索

**\*\*题目链接\*\*:** <https://leetcode.com/problems/range-sum-query-2d-immutable/>

### \*\*题目描述\*\*:

给定一个二维矩阵 `matrix`，处理以下类型的多个查询：计算其子矩形范围内元素的总和，该子矩阵的左上角为

(row1, col1), 右下角为 (row2, col2)。

#### \*\*解题思路\*\*:

使用二维前缀和预处理，然后通过容斥原理计算区域和。

### #### 扩展题目 4：和为 K 的子数组个数

\*\*题目链接\*\*: <https://leetcode.com/problems/subarray-sum-equals-k/>

#### \*\*题目描述\*\*:

给定一个整数数组和一个整数  $k$ ，你需要找到该数组中和为  $k$  的连续的子数组的个数。

#### \*\*解题思路\*\*:

使用哈希表记录前缀和出现的次数，遍历数组时查找是否存在前缀和为（当前前缀和 -  $k$ ）的情况。

\*\*时间复杂度\*\*:  $O(n)$  – 需要遍历数组一次

\*\*空间复杂度\*\*:  $O(n)$  – 哈希表最多存储  $n$  个不同的前缀和

#### \*\*工程化考量\*\*:

- 边界条件处理：空数组、 $k=0$  等特殊情况
- 哈希表优化：使用 HashMap 提高查找效率
- 性能优化：一次遍历完成所有计算

#### \*\*实现文件\*\*:

- [Code02\_SubarraySumEqualsK.java] (Code02\_SubarraySumEqualsK.java)
- [Code02\_SubarraySumEqualsK.py] (Code02\_SubarraySumEqualsK.py)
- [Code02\_SubarraySumEqualsK.cpp] (Code02\_SubarraySumEqualsK.cpp)

#### \*\*代码特性\*\*:

- 包含详细的数学原理和算法推导
- 支持多种边界条件测试
- 提供性能优化和调试技巧

### #### 扩展题目 5：最长子数组和等于 K

\*\*题目链接\*\*: <https://leetcode.com/problems/maximum-size-subarray-sum-equals-k/>

#### \*\*题目描述\*\*:

给定一个数组  $\text{nums}$  和一个目标值  $k$ ，找到和等于  $k$  的最长连续子数组长度。

#### \*\*解题思路\*\*:

使用哈希表记录每个前缀和第一次出现的位置，遍历数组时查找是否存在前缀和为（当前前缀和 -  $k$ ）的情况。

#### #### 扩展题目 6：连续数组

\*\*题目链接\*\*： <https://leetcode.com/problems/contiguous-array/>

\*\*题目描述\*\*：

给定一个二进制数组，找到含有相同数量的 0 和 1 的最长连续子数组。

\*\*解题思路\*\*：

将 0 转换为 -1，问题转化为求和为 0 的最长子数组，使用前缀和 + 哈希表解决。

\*\*时间复杂度\*\*：O(n) – 需要遍历数组一次

\*\*空间复杂度\*\*：O(n) – 哈希表最多存储 n 个不同的前缀和

\*\*工程化考量\*\*：

- 边界条件处理：空数组、单元素数组
- 映射技巧：0 → -1, 1 → 1 的数学变换
- 哈希表初始化：空前缀和为 0 出现在位置 -1
- 性能优化：一次遍历完成所有计算

\*\*实现文件\*\*：

- [Code03\_ContiguousArray.java] (Code03\_ContiguousArray.java)
- [Code03\_ContiguousArray.py] (Code03\_ContiguousArray.py)
- [Code03\_ContiguousArray.cpp] (Code03\_ContiguousArray.cpp)

\*\*代码特性\*\*：

- 包含详细的数学原理和算法推导
- 支持多种边界条件测试
- 提供调试技巧和性能优化建议

#### #### 扩展题目 7：找到和为零的子数组

\*\*题目链接\*\*： <https://www.lintcode.com/problem/138/>

\*\*题目描述\*\*：

给定一个整数数组，找到和为零的子数组。

\*\*解题思路\*\*：

使用前缀和，当两个位置的前缀和相等时，这两个位置之间的子数组和为 0。

#### #### 扩展题目 8：除自身以外数组的乘积

\*\*题目链接\*\*： <https://leetcode.com/problems/product-of-array-except-self/>

### \*\*题目描述\*\*:

给你一个长度为  $n$  的整数数组  $\text{nums}$ , 其中  $n > 1$ , 返回输出数组  $\text{output}$ , 其中  $\text{output}[i]$  等于  $\text{nums}$  中除  $\text{nums}[i]$  之外其余各元素的乘积。

### \*\*解题思路\*\*:

使用两个数组分别存储左侧所有元素的乘积和右侧所有元素的乘积, 然后相乘得到结果。

**\*\*时间复杂度\*\*:**  $O(n)$  – 需要两次遍历数组

**\*\*空间复杂度\*\*:**  $O(1)$  – 不考虑输出数组, 只使用常数额外空间

### \*\*工程化考量\*\*:

- 边界条件处理: 空数组、单元素数组
- 零元素处理: 当数组中有 0 时, 乘积结果的特殊处理
- 性能优化: 两次遍历完成计算, 避免使用除法
- 空间优化: 使用输出数组存储中间结果

### \*\*实现文件\*\*:

- [Code05\_ProductOfArrayExceptSelf.java] (Code05\_ProductOfArrayExceptSelf.java)
- [Code05\_ProductOfArrayExceptSelf.py] (Code05\_ProductOfArrayExceptSelf.py)
- [Code05\_ProductOfArrayExceptSelf.cpp] (Code05\_ProductOfArrayExceptSelf.cpp)

### \*\*代码特性\*\*:

- 包含详细的数学原理和算法推导
- 支持多种边界条件测试
- 提供性能优化和调试技巧

## ### 扩展题目 9: 统计优美子数组

**\*\*题目链接\*\*:** <https://leetcode.com/problems/count-number-of-nice-subarrays/>

### \*\*题目描述\*\*:

给你一个整数数组  $\text{nums}$  和一个整数  $k$ 。如果某个连续子数组中恰好有  $k$  个奇数数字, 我们就认为这个子数组是「优美子数组」。请返回这个数组中「优美子数组」的数目。

### \*\*解题思路\*\*:

将奇数看作 1, 偶数看作 0, 问题转化为求和为  $k$  的子数组个数。

## ### 扩展题目 10: 和可被 K 整除的子数组

**\*\*题目链接\*\*:** <https://leetcode.com/problems/subarray-sums-divisible-by-k/>

### \*\*题目描述\*\*:

给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。

#### \*\*解题思路\*\*:

使用前缀和模 K 的余数，当两个位置的前缀和余数相同时，这两个位置之间的子数组和可被 K 整除。

\*\*时间复杂度\*\*:  $O(n)$  – 需要遍历数组一次

\*\*空间复杂度\*\*:  $O(k)$  – 哈希表最多存储 k 个不同的余数

#### \*\*工程化考量\*\*:

- 负数余数处理：Python 的模运算与数学定义不同，需要特殊处理
- 边界条件：空数组、 $K=0$  等特殊情况
- 性能优化：一次遍历完成所有计算
- 哈希表初始化：空前缀和的余数为 0 出现 1 次

#### \*\*实现文件\*\*:

- [Code04\_SubarraySumsDivisibleByK. java] (Code04\_SubarraySumsDivisibleByK. java)
- [Code04\_SubarraySumsDivisibleByK. py] (Code04\_SubarraySumsDivisibleByK. py)
- [Code04\_SubarraySumsDivisibleByK. cpp] (Code04\_SubarraySumsDivisibleByK. cpp)

#### \*\*代码特性\*\*:

- 包含详细的数学原理和算法推导
- 支持多种边界条件测试
- 提供性能优化和调试技巧

#### 扩展题目 11: Prefix Sum Queries (CSES 2166)

\*\*题目链接\*\*: <https://cses.fi/problemset/task/2166>

#### \*\*题目描述\*\*:

给定一个数组，支持两种操作：

1. 更新位置  $k$  的值为  $u$
2. 查询区间  $[a, b]$  内的最大前缀和

#### \*\*解题思路\*\*:

使用线段树维护区间信息，支持区间最大前缀和查询。

\*\*时间复杂度\*\*:  $O(\log n)$  – 每次操作的时间复杂度

\*\*空间复杂度\*\*:  $O(n)$  – 线段树的空间复杂度

#### \*\*工程化考量\*\*:

- 数据结构选择：线段树提供高效的区间查询和更新
- 边界条件：空数组、单元素数组等特殊情况
- 性能优化：利用线段树的特性优化查询和更新操作

## \*\*实现文件\*\*:

- [Code22\_PrefixSumQueries. java] (Code22\_PrefixSumQueries. java) ✓
- [Code22\_PrefixSumQueries. py] (Code22\_PrefixSumQueries. py) ✓
- [Code22\_PrefixSumQueries. cpp] (Code22\_PrefixSumQueries. cpp) ✓

### 扩展题目 12: Static Range Sum Queries (CSES 1646)

\*\*题目链接\*\*: <https://cses.fi/problemset/task/1646>

## \*\*题目描述\*\*:

给定一个数组，处理多个查询：计算区间  $[a, b]$  内元素的和。

## \*\*解题思路\*\*:

使用基础前缀和技巧，预处理前缀和数组，然后  $O(1)$  时间查询。

## \*\*时间复杂度\*\*:

- 预处理:  $O(n)$  - 需要遍历数组一次
- 查询:  $O(1)$  - 每次查询只需要常数时间

\*\*空间复杂度\*\*:  $O(n)$  - 需要额外的前缀和数组空间

## \*\*工程化考量\*\*:

- 边界条件处理: 空数组、单元素数组
- 性能优化: 预处理前缀和，查询时  $O(1)$  时间
- 空间优化: 必须存储前缀和数组，无法避免

## \*\*实现文件\*\*:

- [Code23\_StaticRangeSumQueries. java] (Code23\_StaticRangeSumQueries. java) ✓
- [Code23\_StaticRangeSumQueries. py] (Code23\_StaticRangeSumQueries. py) ✓
- [Code23\_StaticRangeSumQueries. cpp] (Code23\_StaticRangeSumQueries. cpp) ✓

### 扩展题目 13: Maximum Subarray Sum (CSES 1643)

\*\*题目链接\*\*: <https://cses.fi/problemset/task/1643>

## \*\*题目描述\*\*:

给定一个数组，找到连续子数组的最大和。

## \*\*解题思路\*\*:

使用 Kadane 算法或前缀和思想求最大子数组和。

\*\*时间复杂度\*\*:  $O(n)$  - 需要遍历数组一次

\*\*空间复杂度\*\*:  $O(1)$  - 只需要常数额外空间

## \*\*工程化考量\*\*:

- 边界条件处理: 空数组、全负数数组
- 性能优化: 一次遍历完成计算
- 负数处理: 正确处理全负数情况

## ### 题目 8: 数组操作

\*\*题目链接\*\*: <https://www.hackerrank.com/challenges/crush/problem>

## \*\*题目描述\*\*:

给定一个长度为  $n$  的数组，初始时所有元素都为 0。然后进行  $m$  次操作，每次操作给定三个整数  $a$ ,  $b$ ,  $k$ ，表示将数组中从索引  $a$  到索引  $b$  (包含  $a$  和  $b$ ) 的所有元素都增加  $k$ 。求执行完所有操作后数组中的最大值。

## \*\*解题思路\*\*:

使用差分数组技巧结合前缀和来优化区间更新操作。

\*\*时间复杂度\*\*:  $O(n + m)$  – 需要遍历操作数组和差分数组

\*\*空间复杂度\*\*:  $O(n)$  – 需要存储差分数组

## \*\*工程化考量\*\*:

- 边界条件处理:  $n=0$ 、空操作列表等特殊情况
- 性能优化: 使用差分数组避免  $O(n*m)$  的时间复杂度
- 空间优化: 只存储差分数组，不存储整个数组
- 大数处理:  $k$  可能达到  $10^9$ , 需要确保整数范围

## \*\*实现文件\*\*:

- [Code08\_ArrayManipulation.java] (Code08\_ArrayManipulation.java)
- [Code08\_ArrayManipulation.py] (Code08\_ArrayManipulation.py)
- [Code08\_ArrayManipulation.cpp] (Code08\_ArrayManipulation.cpp)

## \*\*代码特性\*\*:

- 包含详细的数学原理和算法推导
- 支持多种边界条件测试
- 提供性能优化和调试技巧

## ### 题目 9: 二维区域和检索 - 矩阵不可变

\*\*题目链接\*\*: <https://leetcode.com/problems/range-sum-query-2d-immutable/>

## \*\*题目描述\*\*:

给定一个二维矩阵  $matrix$ , 处理以下类型的多个查询: 计算其子矩形范围内元素的总和, 该子矩阵的左上角为  $(row1, col1)$ , 右下角为  $(row2, col2)$ 。

## \*\*解题思路\*\*:

使用二维前缀和预处理技术，利用容斥原理计算任意子矩阵的和。

## \*\*时间复杂度\*\*:

- 初始化:  $O(m \times n)$  - 需要遍历整个矩阵构建前缀和数组
- 查询:  $O(1)$  - 每次查询只需要常数时间

\*\*空间复杂度\*\*:  $O(m \times n)$  - 需要额外的前缀和数组空间

## \*\*工程化考量\*\*:

- 边界条件处理: 空矩阵、单元素矩阵
- 性能优化: 预处理前缀和，查询时  $O(1)$  时间
- 空间优化: 必须存储前缀和数组，无法避免
- 大数处理: 元素值可能很大，需要确保整数范围

## \*\*实现文件\*\*:

- [Code09\_RangeSumQuery2D.java] (Code09\_RangeSumQuery2D.java)
- [Code09\_RangeSumQuery2D.py] (Code09\_RangeSumQuery2D.py)
- [Code09\_RangeSumQuery2D.cpp] (Code09\_RangeSumQuery2D.cpp)

## \*\*代码特性\*\*:

- 包含详细的数学原理和算法推导
- 支持多种边界条件测试
- 提供性能优化和调试技巧

## ### 题目 10: Rikka with Prefix Sum

### \*\*题目链接\*\*: 牛客网

## \*\*题目描述\*\*:

给定一个长度为  $n$  初始全为 0 的数列  $A$ 。 $m$  次操作，要求支持区间加、全局前缀和、区间求和三种操作。

## \*\*解题思路\*\*:

使用差分数组和组合数学来优化操作，结合前缀和的高级应用。

## \*\*实现文件\*\*:

- [Code10\_RikkaWithPrefixSum.java] (Code10\_RikkaWithPrefixSum.java)
- [Code10\_RikkaWithPrefixSum.py] (Code10\_RikkaWithPrefixSum.py)

## ### 题目 11: Good Subarrays

### \*\*题目链接\*\*: <https://codeforces.com/contest/1398/problem/C>

## \*\*题目描述\*\*:

给定一个由数字字符组成的字符串  $s$ , 定义“好数组”为: 数组中所有元素的和等于元素个数。求字符串  $s$  的所有连续子串中, 有多少个“好数组”。

## \*\*解题思路\*\*:

将问题转换为前缀和问题, 通过数学变换使用哈希表统计满足条件的前缀和。

## \*\*实现文件\*\*:

- [Code11\_GoodSubarrays. java] (Code11\_GoodSubarrays. java)
- [Code11\_GoodSubarrays. py] (Code11\_GoodSubarrays. py)

## ### 题目 12: Subarray Divisibility

\*\*题目链接\*\*: <https://cses.fi/problemset/task/1662>

## \*\*题目描述\*\*:

给定一个包含  $n$  个整数的数组和一个正整数  $m$ , 计算有多少个连续子数组的元素和可以被  $m$  整除。

## \*\*解题思路\*\*:

使用前缀和与模运算的性质, 统计具有相同模  $m$  值的前缀和的个数。

## \*\*实现文件\*\*:

- [Code12\_SubarrayDivisibility. java] (Code12\_SubarrayDivisibility. java)
- [Code12\_SubarrayDivisibility. py] (Code12\_SubarrayDivisibility. py)

## ## 算法复杂度分析 (详细版)

### ### 基础前缀和

- \*\*时间复杂度\*\*: 预处理  $O(n)$ , 查询  $O(1)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*最优解\*\*: 是, 必须预处理才能实现  $O(1)$  查询
- \*\*应用场景\*\*: 频繁区间和查询的场景

### ### 哈希表 + 前缀和

- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*最优解\*\*: 是, 必须遍历所有元素才能找到所有满足条件的子数组
- \*\*应用场景\*\*: 子数组和等于特定值的问题

### ### 差分数组 + 前缀和

- \*\*时间复杂度\*\*:  $O(n + m)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*最优解\*\*: 是, 对于大规模区间更新操作, 必须使用差分数组

- **应用场景**: 区间更新操作频繁的场景

#### 二维前缀和

- **时间复杂度**: 预处理  $O(m \times n)$ , 查询  $O(1)$
- **空间复杂度**:  $O(m \times n)$
- **最优解**: 是, 对于频繁的二维区域查询, 预处理是必要的
- **应用场景**: 二维矩阵区域和查询

## 工程化最佳实践

#### 1. 边界条件处理

- 空数组、单元素数组等特殊情况
- 索引越界检查
- 输入验证和异常处理

#### 2. 性能优化策略

- 预处理和缓存计算结果
- 避免重复计算
- 使用合适的数据结构

#### 3. 内存优化

- 复用数组空间
- 及时释放不需要的资源
- 考虑大规模数据的内存使用

#### 4. 代码可读性

- 清晰的变量命名
- 适当的注释和文档
- 模块化的代码结构

#### 5. 测试策略

- 单元测试覆盖边界条件
- 性能测试验证大规模数据表现
- 集成测试确保功能正确性

## 项目优化总结

### 已完成的工作

#### 1. 代码优化和扩展

- **7个核心算法文件**已全面优化:
  - Code01\_RunningSumOf1DArray (一维数组动态和)
  - Code02\_SubarraySumEqualsK (和为 K 的子数组)

- Code03\_ContiguousArray (连续数组)
  - Code04\_SubarraySumsDivisibleByK (和可被 K 整除的子数组)
  - Code05\_ProductOfArrayExceptSelf (除自身以外数组的乘积)
  - Code08\_ArrayManipulation (数组操作)
  - Code09\_RangeSumQuery2D (二维区域和检索)
- 
- **多语言支持**: 每个算法都提供了 Java、Python、C++三种语言的实现
  - **详细注释**: 每个文件都包含详细的中文注释和文档说明

#### #### 2. 工程化特性

- **复杂度分析**: 详细的时间复杂度和空间复杂度分析
- **单元测试**: 完整的测试用例覆盖边界条件
- **性能测试**: 大规模数据下的性能表现测试
- **异常处理**: 完善的边界条件和异常场景处理
- **调试技巧**: 提供调试方法和性能优化建议

#### #### 3. 文档完善

- **README.md**: 全面更新，包含所有优化内容
- **算法总结**: 创建了前缀和算法全面总结文档
- **扩展题目**: 整理了扩展题目列表和解题思路

### ## 技术亮点

#### #### 1. 算法深度优化

- **最优解保证**: 所有算法都是最优解，时间复杂度达到理论最优
- **数学原理**: 详细推导了每个算法的数学原理和公式
- **边界处理**: 完善处理了各种边界条件和异常场景

#### #### 2. 工程化最佳实践

- **代码可读性**: 清晰的变量命名和模块化结构
- **测试覆盖**: 全面的单元测试和性能测试
- **性能优化**: 针对大规模数据的优化策略

#### #### 3. 多语言实现

- **语言特性**: 针对不同语言的特性进行优化
- **一致性**: 保持三种语言实现的功能一致性
- **可移植性**: 代码可以在不同环境中运行

### ## 学习价值

#### #### 1. 算法掌握

- **前缀和核心**: 深入理解前缀和的基本原理和应用
- **变种算法**: 掌握前缀和的各种变种和扩展应用

- **优化技巧**: 学习算法优化的各种技巧和方法

#### #### 2. 工程能力

- **代码质量**: 学习如何编写高质量的工程代码
- **测试驱动**: 掌握测试驱动的开发方法
- **性能分析**: 学习性能分析和优化的方法

#### #### 3. 面试准备

- **高频考点**: 覆盖了前缀和相关的高频面试题
- **解题思路**: 提供了清晰的解题思路和模板
- **实战经验**: 通过实际代码实现积累实战经验

### ### 后续学习建议

#### #### 1. 算法扩展

- 学习更多前缀和的变种和应用场景
- 探索前缀和与其他算法的结合使用
- 研究前缀和在机器学习和大数据中的应用

#### #### 2. 工程实践

- 在实际项目中应用前缀和算法
- 学习更多工程化最佳实践
- 参与开源项目积累经验

#### #### 3. 深入学习

- 研究算法的时间复杂度分析理论
- 学习更多数据结构和算法
- 探索算法在系统设计中的应用

### ### 项目文件结构

```

```
class046/
├── README.md                      # 项目主文档
├── 前缀和算法全面总结.md            # 算法理论总结
├── 扩展题目列表.md                # 扩展题目整理
├── Code01_RunningSumOf1DArray.java # Java 实现
├── Code01_RunningSumOf1DArray.py    # Python 实现
├── Code01_RunningSumOf1DArray.cpp   # C++ 实现
├── Code02_SubarraySumEqualsK.java   # Java 实现
├── Code02_SubarraySumEqualsK.py     # Python 实现
├── Code02_SubarraySumEqualsK.cpp   # C++ 实现
├── Code03_ContiguousArray.java     # Java 实现
└── Code03_ContiguousArray.py       # Python 实现
```

```
└── Code03_ContiguousArray.cpp      # C++实现  
└── Code04_SubarraySumsDivisibleByK.java # Java 实现  
└── Code04_SubarraySumsDivisibleByK.py # Python 实现  
└── Code04_SubarraySumsDivisibleByK.cpp # C++实现  
└── Code05_ProductOfArrayExceptSelf.java # Java 实现  
└── Code05_ProductOfArrayExceptSelf.py # Python 实现  
└── Code05_ProductOfArrayExceptSelf.cpp # C++实现  
└── Code08_ArrayManipulation.java    # Java 实现  
└── Code08_ArrayManipulation.py     # Python 实现  
└── Code08_ArrayManipulation.cpp    # C++实现  
└── Code09_RangeSumQuery2D.java     # Java 实现  
└── Code09_RangeSumQuery2D.py       # Python 实现  
└── Code09_RangeSumQuery2D.cpp      # C++实现  
...
```

#### #### 总结

本项目全面优化了前缀和算法的实现，提供了高质量的代码和详细的文档。通过这个项目，学习者可以：

1. 深入理解前缀和算法的原理和应用
2. 掌握算法优化和工程化实践
3. 积累多语言编程经验
4. 为算法面试和实际项目开发做好准备

前缀和作为一种基础而强大的算法技巧，在算法学习和工程实践中都有重要价值。掌握好前缀和，将为学习更复杂的算法打下坚实基础。

---

文件：SUMMARY.md

---

## # 前缀和算法全面总结

### ## 核心思想

前缀和是一种重要的算法技巧，通过预处理数组的累积和，可以快速回答区间查询问题。其核心思想是空间换时间，通过预算算来加速查询。

### ## 基本形式

#### #### 一维前缀和

...

```
prefix[i] = nums[0] + nums[1] + ... + nums[i]  
// 或者
```

```
prefix[0] = 0
prefix[i] = nums[0] + nums[1] + ... + nums[i-1]
```

```

区间和计算:

```
```
sum(i, j) = prefix[j+1] - prefix[i] // 对应第二种定义
// 或者
sum(i, j) = prefix[j] - prefix[i-1] // 对应第一种定义
```

```

### 二维前缀和

```
```
prefix[i][j] = sum of all elements in rectangle from (0,0) to (i-1, j-1)
```

```

区域和计算 (容斥原理):

```
```
sum((r1, c1), (r2, c2)) = prefix[r2+1][c2+1] - prefix[r1][c2+1] - prefix[r2+1][c1] + prefix[r1][c1]
```

```

## 常见题型及解法

### 1. 基础前缀和

**\*\*特点\*\*:** 直接计算区间和

**\*\*技巧\*\*:** 预处理前缀和数组, O(1)时间查询

**\*\*代表题目\*\*:**

- 区域和检索 - 数组不可变 (LeetCode 303)
- 一维数组的动态和 (LeetCode 1480)
- 找到数组中心索引 (LeetCode 724)
- 区间和查询 - 不可变 (LeetCode 303)
- 二维区域和检索 - 不可变 (LeetCode 304)

### 2. 哈希表 + 前缀和

**\*\*特点\*\*:** 查找满足特定条件的子数组

**\*\*技巧\*\*:** 用哈希表记录前缀和的状态, 通过数学变换找到目标条件

**\*\*关键变换\*\*:**

- 和为 k 的子数组:  $\text{sum}[j] - \text{sum}[i] = k \rightarrow \text{sum}[i] = \text{sum}[j] - k$
- 和为 0 的子数组:  $\text{sum}[j] = \text{sum}[i]$
- 和可被 K 整除:  $(\text{sum}[j] - \text{sum}[i]) \% K = 0 \rightarrow \text{sum}[j] \% K = \text{sum}[i] \% K$

**\*\*代表题目\*\*:**

- 和为 K 的子数组 (LeetCode 560)

- 连续数组 (LeetCode 525)
- 和可被 K 整除的子数组 (LeetCode 974)
- 找到和为零的子数组 (LintCode 138)
- 连续的子数组和 (LeetCode 523)

#### #### 3. 特殊映射 + 前缀和

- \*\*特点\*\*:** 需要将原问题转换为前缀和问题
- \*\*技巧\*\*:** 通过适当的映射将问题转化为前缀和形式
- \*\*常见映射\*\*:**
- 0/1 数组 → -1/1 数组 (便于处理相等数量问题)
  - 字符状态 → 位运算状态 (处理奇偶性问题)

**\*\*代表题目\*\*:**

- 连续数组 (LeetCode 525):  $0 \rightarrow -1, 1 \rightarrow 1$
- 每个元音包含偶数次的最长子字符串 (LeetCode 1371): 元音状态用位运算表示
- 表现良好的最长时间段 (LeetCode 1124):  $>8 \rightarrow 1, \leq 8 \rightarrow -1$

#### #### 4. 前缀积及其他运算

- \*\*特点\*\*:** 使用乘法或其他运算替代加法
- \*\*技巧\*\*:** 左右扫描分别计算前缀和后缀, 避免使用除法
- \*\*代表题目\*\*:**
- 除自身以外数组的乘积 (LeetCode 238)

## ## 关键技巧总结

### #### 1. 哈希表的初始化

```

// 和相关问题

```
map.put(0, 1); // 表示空前缀和为 0 出现 1 次
```

// 位置相关问题

```
map.put(0, -1); // 表示前缀和 0 最早出现在位置-1
```

```

### #### 2. 负数取模处理

``` java

```
int remainder = sum % K;
if (remainder < 0) {
    remainder += K;
}
```

```

#### #### 3. 状态压缩

对于有限状态问题（如元音字母奇偶性），使用位运算压缩状态：

```
```java
// 5个元音字母用5位二进制表示奇偶性
int status = 0;
status ^= (1 << k); // 切换第k位的状态
```

```

## ## 时间与空间复杂度

| 类型      | 时间复杂度                 | 空间复杂度           |
|---------|-----------------------|-----------------|
| 基础前缀和   | $O(n)$ 预处理, $O(1)$ 查询 | $O(n)$          |
| 哈希表+前缀和 | $O(n)$                | $O(n)$          |
| 前缀积     | $O(n)$                | $O(1)$ (不计输出数组) |

## ## 工程化考虑

### #### 1. 边界条件处理

- 空数组/空指针检查
- 单元素数组处理
- 整数溢出检查

### #### 2. 优化技巧

- 复用输出数组减少空间使用
- 使用静态数组避免动态内存分配（在大小确定的情况下）
- 选择合适的数据结构（HashMap vs Array）

### #### 3. 可读性增强

- 详细注释说明数学变换过程
- 变量命名清晰表达含义
- 分步骤实现，便于调试

## ## 面试要点

### #### 1. 本质理解

- 前缀和本质上是缓存思想的应用
- 通过预计算避免重复计算
- 哈希表用于快速查找历史状态

### #### 2. 问题识别

能在以下场景中识别出前缀和的适用性：

- 区间查询问题
- 子数组和相关问题

- 需要统计满足特定和条件的子数组个数或长度

#### #### 3. 变体处理

- 能处理一维到二维的扩展
- 能处理加法到其他运算的扩展
- 能处理直接计算到需要数学变换的扩展

#### ## 学习路径建议

1. **基础掌握**: 熟练实现一维、二维前缀和，理解其原理
2. **典型应用**: 掌握和为 K 的子数组、找到数组中心索引、连续的子数组和等经典题目
3. **变体练习**: 练习各种变体，如和可被 K 整除、状态压缩等
4. **综合应用**: 在更复杂问题中识别和应用前缀和技巧
5. **工程实践**: 关注边界处理、性能优化等工程化细节

#### ## 常见误区

1. **忽略边界**: 忘记处理空数组、单元素等情况
2. **负数取模**: Java 中负数取模结果为负，需要特殊处理
3. **状态表示**: 状态压缩时位运算使用错误
4. **初始化错误**: 哈希表初始化值不正确导致结果错误
5. **溢出问题**: 大数运算时未考虑整数溢出

通过系统学习和练习这些前缀和相关题目，可以建立起对这一重要算法技巧的全面理解，并能在实际面试和工程中灵活应用。

=====

文件: 前缀和算法全面总结.md

=====

## # 前缀和算法全面总结

### ## 核心思想

前缀和是一种重要的算法技巧，通过预处理数组的累积和，可以快速回答区间查询问题。其核心思想是空间换时间，通过预算算来加速查询。

### ## 基本形式

#### #### 一维前缀和

```

```
prefix[i] = nums[0] + nums[1] + ... + nums[i]
// 或者
```

```
prefix[0] = 0
prefix[i] = nums[0] + nums[1] + ... + nums[i-1]
```

```

区间和计算:

```
```
sum(i, j) = prefix[j+1] - prefix[i] // 对应第二种定义
// 或者
sum(i, j) = prefix[j] - prefix[i-1] // 对应第一种定义
```

```

### 二维前缀和

```
```
prefix[i][j] = sum of all elements in rectangle from (0,0) to (i-1, j-1)
```

```

区域和计算 (容斥原理):

```
```
sum((r1, c1), (r2, c2)) = prefix[r2+1][c2+1] - prefix[r1][c2+1] - prefix[r2+1][c1] + prefix[r1][c1]
```

```

## 常见题型及解法

### 1. 基础前缀和

**\*\*特点\*\*:** 直接计算区间和

**\*\*技巧\*\*:** 预处理前缀和数组, O(1)时间查询

**\*\*代表题目\*\*:**

- 区域和检索 - 数组不可变 (LeetCode 303)
- 一维数组的动态和 (LeetCode 1480)
- 找到数组中心索引 (LeetCode 724)
- 区间和查询 - 不可变 (LeetCode 303)
- 二维区域和检索 - 不可变 (LeetCode 304)

### 2. 哈希表 + 前缀和

**\*\*特点\*\*:** 查找满足特定条件的子数组

**\*\*技巧\*\*:** 用哈希表记录前缀和的状态, 通过数学变换找到目标条件

**\*\*关键变换\*\*:**

- 和为 k 的子数组:  $\text{sum}[j] - \text{sum}[i] = k \rightarrow \text{sum}[i] = \text{sum}[j] - k$
- 和为 0 的子数组:  $\text{sum}[j] = \text{sum}[i]$
- 和可被 K 整除:  $(\text{sum}[j] - \text{sum}[i]) \% K = 0 \rightarrow \text{sum}[j] \% K = \text{sum}[i] \% K$

**\*\*代表题目\*\*:**

- 和为 K 的子数组 (LeetCode 560)

- 连续数组 (LeetCode 525)
- 和可被 K 整除的子数组 (LeetCode 974)
- 找到和为零的子数组 (LintCode 138)
- 连续的子数组和 (LeetCode 523)

#### #### 3. 特殊映射 + 前缀和

- \*\*特点\*\*:** 需要将原问题转换为前缀和问题
- \*\*技巧\*\*:** 通过适当的映射将问题转化为前缀和形式
- \*\*常见映射\*\*:**
- 0/1 数组 → -1/1 数组 (便于处理相等数量问题)
  - 字符状态 → 位运算状态 (处理奇偶性问题)

**\*\*代表题目\*\*:**

- 连续数组 (LeetCode 525):  $0 \rightarrow -1, 1 \rightarrow 1$
- 每个元音包含偶数次的最长子字符串 (LeetCode 1371): 元音状态用位运算表示
- 表现良好的最长时间段 (LeetCode 1124):  $>8 \rightarrow 1, \leq 8 \rightarrow -1$

#### #### 4. 前缀积及其他运算

- \*\*特点\*\*:** 使用乘法或其他运算替代加法
- \*\*技巧\*\*:** 左右扫描分别计算前缀和后缀, 避免使用除法
- \*\*代表题目\*\*:**
- 除自身以外数组的乘积 (LeetCode 238)

#### #### 5. 差分数组 + 前缀和

- \*\*特点\*\*:** 高效处理区间更新操作
- \*\*技巧\*\*:** 使用差分数组标记区间变化, 然后通过前缀和还原结果
- \*\*代表题目\*\*:**
- 数组操作 (HackerRank)
  - 区间加法 (LeetCode 370)

## ## 关键技巧总结

#### #### 1. 哈希表的初始化

```

// 和相关问题

```
map.put(0, 1); // 表示空前缀和为 0 出现 1 次
```

// 位置相关问题

```
map.put(0, -1); // 表示前缀和 0 最早出现在位置-1
```

```

#### #### 2. 负数取模处理

``` java

```
int remainder = sum % K;  
if (remainder < 0) {  
    remainder += K;  
}  
...  
...
```

### ### 3. 状态压缩

对于有限状态问题（如元音字母奇偶性），使用位运算压缩状态：

```
``` java  
// 5个元音字母用5位二进制表示奇偶性  
int status = 0;  
status ^= (1 << k); // 切换第k位的状态  
...  
...
```

## ## 时间与空间复杂度

类型	时间复杂度	空间复杂度
基础前缀和	$O(n)$ 预处理, $O(1)$ 查询	$O(n)$
哈希表+前缀和	$O(n)$	$O(n)$
前缀积	$O(n)$	$O(1)$ (不计输出数组)
差分数组+前缀和	$O(n + m)$	$O(n)$
二维前缀和	$O(m*n)$ 预处理, $O(1)$ 查询	$O(m*n)$

## ## 工程化考虑

### ### 1. 边界条件处理

- 空数组/空指针检查
- 单元素数组处理
- 整数溢出检查
- 索引越界检查

### ### 2. 优化技巧

- 复用输出数组减少空间使用
- 使用静态数组避免动态内存分配（在大小确定的情况下）
- 选择合适的数据结构（HashMap vs Array）
- 原地修改节省空间

### ### 3. 可读性增强

- 详细注释说明数学变换过程
- 变量命名清晰表达含义
- 分步骤实现，便于调试
- 单元测试覆盖各种边界情况

#### #### 4. 性能优化

- 预处理阶段优化
- 查询阶段优化
- 内存使用优化
- 算法选择优化

### ## 面试要点

#### #### 1. 本质理解

- 前缀和本质上是缓存思想的应用
- 通过预计算避免重复计算
- 哈希表用于快速查找历史状态

#### #### 2. 问题识别

能在以下场景中识别出前缀和的适用性：

- 区间查询问题
- 子数组和相关问题
- 需要统计满足特定和条件的子数组个数或长度
- 需要高效处理区间更新的问题

#### #### 3. 变体处理

- 能处理一维到二维的扩展
- 能处理加法到其他运算的扩展
- 能处理直接计算到需要数学变换的扩展
- 能处理基础查询到复杂统计的扩展

#### #### 4. 代码实现

- 清晰的变量命名
- 完整的边界处理
- 详细的注释说明
- 充分的测试用例

### ## 学习路径建议

1. **基础掌握**: 熟练实现一维、二维前缀和，理解其原理
2. **典型应用**: 掌握和为 K 的子数组、找到数组中心索引、连续的子数组和等经典题目
3. **变体练习**: 练习各种变体，如和可被 K 整除、状态压缩等
4. **综合应用**: 在更复杂问题中识别和应用前缀和技巧
5. **工程实践**: 关注边界处理、性能优化等工程化细节

### ## 常见误区

1. \*\*忽略边界\*\*: 忘记处理空数组、单元素等情况
2. \*\*负数取模\*\*: Java 中负数取模结果为负, 需要特殊处理
3. \*\*状态表示\*\*: 状态压缩时位运算使用错误
4. \*\*初始化错误\*\*: 哈希表初始化值不正确导致结果错误
5. \*\*溢出问题\*\*: 大数运算时未考虑整数溢出
6. \*\*索引混淆\*\*: 1-based 和 0-based 索引混淆

## ## 多语言实现差异

### #### Java

- 强类型语言, 需要显式声明类型
- 自动内存管理, 无需手动释放
- HashMap 提供  $O(1)$  的平均查找时间
- 需要处理负数取模问题

### #### C++

- 需要手动管理内存
- unordered\_map 提供  $O(1)$  的平均查找时间
- 需要包含必要的头文件
- 需要处理编译错误和链接问题

### #### Python

- 动态类型语言, 语法简洁
- 字典自动处理哈希冲突
- 自动处理大整数, 无需担心溢出
- 支持负索引和切片操作

## ## 实际应用场景

1. \*\*数据分析\*\*: 快速计算数据区间统计量
2. \*\*图像处理\*\*: 二维前缀和用于图像滤波和特征提取
3. \*\*游戏开发\*\*: 区域统计和碰撞检测
4. \*\*数据库优化\*\*: 预计算查询结果加速查询
5. \*\*机器学习\*\*: 特征工程中的统计特征计算

## ## 进阶学习方向

1. \*\*树状数组\*\*: 支持动态更新的前缀和结构
2. \*\*线段树\*\*: 更通用的区间查询数据结构
3. \*\*稀疏表\*\*: 静态区间查询的优化结构
4. \*\*莫队算法\*\*: 离线区间查询算法
5. \*\*分块算法\*\*: 平衡预处理和查询的折中方案

通过系统学习和练习这些前缀和相关题目，可以建立起对这一重要算法技巧的全面理解，并能在实际面试和工程中灵活应用。

---

文件：扩展题目列表.md

---

## # 前缀和算法扩展题目列表

### ## 基础前缀和题目

#### ### 1. 一维数组的动态和 (LeetCode 1480)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/running-sum-of-1d-array/>

**\*\*难度\*\*:** 简单

**\*\*解题思路\*\*:** 基础前缀和应用，直接累加即可

#### ### 2. 区域和检索 - 数组不可变 (LeetCode 303)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/range-sum-query-immutable/>

**\*\*难度\*\*:** 简单

**\*\*解题思路\*\*:** 预处理前缀和数组， $O(1)$ 时间查询

#### ### 3. 二维区域和检索 - 矩阵不可变 (LeetCode 304)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/range-sum-query-2d-immutable/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 二维前缀和，使用容斥原理计算区域和

#### ### 4. 找到数组中心索引 (LeetCode 724)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/find-pivot-index/>

**\*\*难度\*\*:** 简单

**\*\*解题思路\*\*:** 前缀和+后缀和，找到左右和相等的位置

### ## 哈希表 + 前缀和题目

#### ### 5. 和为 K 的子数组 (LeetCode 560)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/subarray-sum-equals-k/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 前缀和+哈希表，统计满足条件的子数组个数

#### ### 6. 连续数组 (LeetCode 525)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/contiguous-array/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 将 0 映射为 -1，问题转化为求和为 0 的最长子数组

### ### 7. 和可被 K 整除的子数组 (LeetCode 974)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/subarray-sums-divisible-by-k/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 前缀和模 K, 统计相同余数的前缀和对数

### ### 8. 最长子数组和等于 K (LeetCode 325)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/maximum-size-subarray-sum-equals-k/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 前缀和+哈希表, 记录每个前缀和第一次出现的位置

### ### 9. 找到和为零的子数组 (LintCode 138)

**\*\*题目链接\*\*:** <https://www.lintcode.com/problem/138/>

**\*\*难度\*\*:** 简单

**\*\*解题思路\*\*:** 前缀和+哈希表, 找到两个相同的前缀和

## ## 特殊技巧题目

### ### 10. 除自身以外数组的乘积 (LeetCode 238)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/product-of-array-except-self/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 前缀积+后缀积, 避免使用除法

### ### 11. 表现良好的最长时间段 (LeetCode 1124)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/longest-well-performing-interval/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 将 $>8$  映射为 1,  $\leq 8$  映射为 -1, 问题转化为和大于 0 的最长子数组

### ### 12. 每个元音包含偶数次的最长子字符串 (LeetCode 1371)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/find-the-longest-substring-containing-vowels-in-even-counts/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 状态压缩+前缀和, 用位运算表示元音奇偶性

### ### 13. 统计优美子数组 (LeetCode 1248)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/count-number-of-nice-subarrays/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 将奇数映射为 1, 偶数映射为 0, 问题转化为和为 k 的子数组个数

## ## 差分数组题目

### ### 14. 数组操作 (HackerRank)

**\*\*题目链接\*\*:** <https://www.hackerrank.com/challenges/crush/problem>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 差分数组+前缀和，高效处理区间更新

#### #### 15. 区间加法 (LeetCode 370)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/range-addition/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 差分数组技巧，标记区间变化后通过前缀和还原

## ## 竞赛题目

#### #### 16. Good Subarrays (Codeforces 1398C)

**\*\*题目链接\*\*:** <https://codeforces.com/contest/1398/problem/C>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 数学变换，将问题转化为前缀和问题

#### #### 17. Subarray Divisibility (CSES 1662)

**\*\*题目链接\*\*:** <https://cses.fi/problemset/task/1662>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 前缀和模运算，统计相同余数的前缀和对数

#### #### 18. Rikka with Prefix Sum (牛客网)

**\*\*题目链接\*\*:** 牛客网竞赛题目

**\*\*难度\*\*:** 困难

**\*\*解题思路\*\*:** 差分数组+组合数学，高级前缀和应用

## ## 其他平台题目

#### #### 19. 正负数个数相等的最长子数组 (牛客网)

**\*\*解题思路\*\*:** 将正数映射为 1，负数映射为 -1，问题转化为和为 0 的最长子数组

#### #### 20. 使数组和能被 P 整除 (LeetCode 1590)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/make-sum-divisible-by-p/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 前缀和模运算，找到需要移除的最短子数组

#### #### 21. 连续的子数组和 (LeetCode 523)

**\*\*题目链接\*\*:** <https://leetcode.com/problems/continuous-subarray-sum/>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 前缀和模运算，检查是否存在长度至少为 2 的满足条件的子数组

#### #### 22. Prefix Sum Queries (CSES 2166)

**\*\*题目链接\*\*:** <https://cses.fi/problemset/task/2166>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 使用线段树或类似数据结构维护前缀和，支持区间最大前缀和查询

**\*\*实现文件\*\*:**

- [Code22\_PrefixSumQueries. java] (Code22\_PrefixSumQueries. java)
- [Code22\_PrefixSumQueries. py] (Code22\_PrefixSumQueries. py)
- [Code22\_PrefixSumQueries. cpp] (Code22\_PrefixSumQueries. cpp)

### ### 23. Static Range Sum Queries (CSES 1646)

**\*\*题目链接\*\*:** <https://cses.fi/problemset/task/1646>

**\*\*难度\*\*:** 简单

**\*\*解题思路\*\*:** 基础前缀和应用，预处理前缀和数组， $O(1)$ 时间查询区间和

**\*\*实现文件\*\*:**

- [Code23\_StaticRangeSumQueries. java] (Code23\_StaticRangeSumQueries. java)
- [Code23\_StaticRangeSumQueries. py] (Code23\_StaticRangeSumQueries. py)
- [Code23\_StaticRangeSumQueries. cpp] (Code23\_StaticRangeSumQueries. cpp)

### ### 24. Maximum Subarray Sum (CSES 1643)

**\*\*题目链接\*\*:** <https://cses.fi/problemset/task/1643>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 使用 Kadane 算法或前缀和思想求最大子数组和

### ### 25. Prefix Sum Primes (Codeforces 1149A)

**\*\*题目链接\*\*:** <https://codeforces.com/problemset/problem/1149/A>

**\*\*难度\*\*:** 简单

**\*\*解题思路\*\*:** 构造前缀和序列，使得尽可能多的前缀和为素数

### ### 26. Prefix Sum Addicts (Codeforces 1738B)

**\*\*题目链接\*\*:** <https://codeforces.com/problemset/problem/1738/B>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 根据给定的后  $k$  个前缀和，构造原数组

### ### 27. 最大前缀和 (洛谷 P5369)

**\*\*题目链接\*\*:** <https://www.luogu.com.cn/problem/P5369>

**\*\*难度\*\*:** 困难

**\*\*解题思路\*\*:** 使用动态规划或组合数学计算所有排列中最大前缀和的总和

### ### 28. 前缀和 (洛谷 P9532)

**\*\*题目链接\*\*:** <https://www.luogu.com.cn/problem/P9532>

**\*\*难度\*\*:** 中等

**\*\*解题思路\*\*:** 构造满足特定规律的前缀和序列

## ## 题目分类总结

### ### 基础应用类

- 一维数组的动态和

- 区域和检索
- 二维区域和检索
- 找到数组中心索引

#### #### 哈希表应用类

- 和为 K 的子数组
- 连续数组
- 和可被 K 整除的子数组
- 最长子数组和等于 K

#### #### 特殊技巧类

- 除自身以外数组的乘积
- 表现良好的最长时间段
- 每个元音包含偶数次的最长子字符串
- 统计优美子数组

#### #### 差分数组类

- 数组操作
- 区间加法

#### #### 竞赛进阶类

- Good Subarrays
- Subarray Divisibility
- Rikka with Prefix Sum

### ## 学习建议

1. \*\*从基础开始\*\*: 先掌握基础的前缀和应用，理解其核心思想
2. \*\*循序渐进\*\*: 从简单题目开始，逐步挑战中等和困难题目
3. \*\*分类练习\*\*: 按题目类型分类练习，掌握每种类型的解题技巧
4. \*\*多语言实现\*\*: 尝试用 Java、C++、Python 等多种语言实现，理解语言特性差异
5. \*\*总结归纳\*\*: 每做完一类题目后总结解题思路和技巧
6. \*\*实战应用\*\*: 在实际项目中寻找前缀和的应用场景

### ## 资源推荐

#### #### 在线评测平台

- LeetCode: 题目丰富，社区活跃
- LintCode: 中文界面，题目分类清晰
- Codeforces: 竞赛题目，难度较高
- CSES: 经典算法题目集合

#### #### 学习资料

- 《算法导论》：经典算法教材
- 《编程珠玑》：算法思维训练
- 各大算法博客和视频教程

通过系统练习这些题目，可以全面掌握前缀和算法及其各种变体应用。

[代码文件]

文件: Code01\_PrefixSumArray.java

```
package class046;
```

```
/**  
 * 区域和检索 - 数组不可变 (Range Sum Query - Immutable)  
 *  
 * 题目描述:  
 * 给定一个整数数组 nums，处理以下类型的多个查询：  
 * 计算索引 left 和 right (包含 left 和 right) 之间的 nums 元素的和，其中 left <= right。  
 *  
 * 实现 NumArray 类：  
 * - NumArray(int[] nums) 使用数组 nums 初始化对象  
 * - int sumRange(int i, int j) 返回数组 nums 中索引 left 和 right 之间的元素的总和  
 *  
 * 示例：  
 * 输入：  
 * ["NumArray", "sumRange", "sumRange", "sumRange"]  
 * [[[ -2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]  
 * 输出：  
 * [null, 1, -1, -3]  
 *  
 * 解释：  
 * NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);  
 * numArray.sumRange(0, 2); // return 1 ((-2) + 0 + 3)  
 * numArray.sumRange(2, 5); // return -1 (3 + (-5) + 2 + (-1))  
 * numArray.sumRange(0, 5); // return -3 ((-2) + 0 + 3 + (-5) + 2 + (-1))  
 *  
 * 提示：  
 * 1 <= nums.length <= 10^4  
 * -10^5 <= nums[i] <= 10^5  
 * 0 <= i <= j < nums.length  
 * 最多调用 10^4 次 sumRange 方法
```

```
*  
* 题目链接: https://leetcode.cn/problems/range-sum-query-immutable/  
*  
* 解题思路:  
* 使用前缀和技巧预处理数组，使得每次查询可以在 O(1) 时间内完成。  
* 1. 构建前缀和数组 sum，其中 sum[i] 表示 nums[0] 到 nums[i-1] 的和  
* 2. 区间 [left, right] 的和等于 sum[right+1] - sum[left]  
*  
* 时间复杂度:  
* - 初始化: O(n) - 需要遍历数组一次构建前缀和数组  
* - 查询: O(1) - 每次查询只需要常数时间  
* 空间复杂度: O(n) - 需要额外的前缀和数组空间  
*  
* 工程化考量:  
* 1. 边界条件处理: 空数组、单元素数组  
* 2. 性能优化: 预处理前缀和，查询时 O(1) 时间  
* 3. 空间优化: 必须存储前缀和数组，无法避免  
* 4. 大数处理: 元素值可能很大，需要确保整数范围  
*  
* 最优解分析:  
* 这是最优解，因为查询次数可能很多，预处理后可以实现 O(1) 查询时间。  
* 对于频繁查询的场景，预处理是必要的。  
*  
* 算法核心:  
* 前缀和公式: sum[i] = sum[i-1] + nums[i-1]  
* 区间和公式: sumRange(left, right) = sum[right+1] - sum[left]  
*  
* 算法调试技巧:  
* 1. 打印中间过程: 显示前缀和数组的计算过程  
* 2. 边界测试: 测试空数组、单元素数组等特殊情况  
* 3. 性能测试: 测试大规模数组下的性能表现  
*  
* 语言特性差异:  
* Java 中数组是对象，可以直接访问 length 属性。  
* 与 C++ 相比，Java 有自动内存管理，无需手动释放数组内存。  
* 与 Python 相比，Java 是静态类型语言，需要显式声明数组类型。  
*/  
  
public class Code01_PrefixSumArray {  
  
    /**  
     * NumArray 类用于处理区域和查询  
     */  
    class NumArray {
```

```
public int[] sum; // 前缀和数组

/**
 * 构造函数，初始化前缀和数组
 *
 * @param nums 输入数组
 *
 * 异常场景处理：
 * - 空数组：创建空的前缀和数组
 * - 单元素数组：正常处理
 *
 * 边界条件：
 * - 数组为空
 * - 数组只有一行或一列
 * - 查询范围超出数组边界
 */

public NumArray(int[] nums) {
    // 创建前缀和数组，大小为 nums.length+1
    // 使用 nums.length+1 可以避免边界检查
    sum = new int[nums.length + 1];

    // 计算前缀和，时间复杂度 O(n)
    for (int i = 1; i <= nums.length; i++) {
        // 前缀和公式：当前前缀和 = 前一个前缀和 + 当前元素
        sum[i] = sum[i - 1] + nums[i - 1];
        // 调试打印：显示前缀和计算过程
        // System.out.println("位置 " + i + ": sum[" + i + "] = " + sum[i]);
    }
}

/**
 * 计算子数组区域和
 *
 * @param left 左边界（包含）
 * @param right 右边界（包含）
 * @return 子数组元素和
 *
 * 边界条件：
 * - 坐标超出范围（题目保证有效）
 * - 单元素查询
 * - 整个数组查询
 */
```

```

public int sumRange(int left, int right) {
    // 使用前缀和公式计算区域和，时间复杂度 O(1)
    // 公式：区间和 = 右边界前缀和 - 左边界前缀和
    int result = sum[right + 1] - sum[left];

    // 调试打印：显示查询过程
    // System.out.println("查询区域 [" + left + ", " + right + "]: 结果 = " + result);

    return result;
}

}

}

=====

文件: Code01_RunningSumOf1DArray.cpp
=====

/***
 * 一维数组的动态和 (Running Sum of 1d Array)
 *
 * 题目描述：
 * 给你一个数组 nums 。数组「动态和」的计算公式为：runningSum[i] = sum(nums[0]…nums[i]) 。
 * 请返回 nums 的动态和。
 *
 * 示例：
 * 输入：nums = [1, 2, 3, 4]
 * 输出：[1, 3, 6, 10]
 * 解释：动态和计算过程为 [1, 1+2, 1+2+3, 1+2+3+4] 。
 *
 * 输入：nums = [1, 1, 1, 1, 1]
 * 输出：[1, 2, 3, 4, 5]
 * 解释：动态和计算过程为 [1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1] 。
 *
 * 输入：nums = [3, 1, 2, 10, 1]
 * 输出：[3, 4, 6, 16, 17]
 *
 * 提示：
 * 1 <= nums.length <= 1000
 * -10^6 <= nums[i] <= 10^6
 *
 * 题目链接: https://leetcode.com/problems/running-sum-of-1d-array/
 */

```

- \* 解题思路：
- \* 使用前缀和的思想，从前向后累加即可。
- \*
- \* 时间复杂度：O(n) - 需要遍历数组一次
- \* 空间复杂度：O(1) - 不考虑输出数组，只使用常数额外空间
- \*
- \* 工程化考量：
  1. 边界条件处理：空数组、单元素数组
  2. 原地修改：节省空间，避免创建新数组
  3. 整数溢出：虽然题目保证在 32 位整数范围内，但实际工程中需要考虑
  4. 代码可读性：清晰的变量命名和注释
- \*
- \* 最优解分析：
  - 这是最优解，因为必须遍历所有元素才能计算前缀和，时间复杂度 O(n) 无法优化。
  - 空间复杂度 O(1) 也是最优的（不考虑输出数组）。
- \*/

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

using namespace std;

class Solution {
public:
    /**
     * 计算数组的动态和
     *
     * @param nums 输入数组
     * @return 动态和数组
     */
    vector<int> runningSum(vector<int>& nums) {
        // 边界情况处理：空数组或单元素数组直接返回
        if (nums.empty()) {
            return nums;
        }

        // 直接在原数组上进行修改，节省空间
        // 从第二个元素开始，每个位置的值等于前一个位置的前缀和加上当前位置的原始值
        for (int i = 1; i < nums.size(); i++) {
            nums[i] += nums[i - 1];
        }

        return nums;
    }
}
```

```
}

};

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

    // 测试用例 1: 正常情况
    vector<int> nums1 = {1, 2, 3, 4};
    vector<int> result1 = solution.runningSum(nums1);
    cout << "测试用例 1: ";
    for (int num : result1) {
        cout << num << " ";
    }
    cout << " (预期: 1 3 6 10)" << endl;

    // 测试用例 2: 全 1 数组
    vector<int> nums2 = {1, 1, 1, 1, 1};
    vector<int> result2 = solution.runningSum(nums2);
    cout << "测试用例 2: ";
    for (int num : result2) {
        cout << num << " ";
    }
    cout << " (预期: 1 2 3 4 5)" << endl;

    // 测试用例 3: 混合数值
    vector<int> nums3 = {3, 1, 2, 10, 1};
    vector<int> result3 = solution.runningSum(nums3);
    cout << "测试用例 3: ";
    for (int num : result3) {
        cout << num << " ";
    }
    cout << " (预期: 3 4 6 16 17)" << endl;

    // 测试用例 4: 空数组
    vector<int> nums4 = {};
    vector<int> result4 = solution.runningSum(nums4);
    cout << "测试用例 4: ";
    for (int num : result4) {
        cout << num << " ";
    }
}
```

```

cout << " (预期: 空数组)" << endl;

// 测试用例 5: 单元素数组
vector<int> nums5 = {5};
vector<int> result5 = solution.runningSum(nums5);
cout << "测试用例 5: ";
for (int num : result5) {
    cout << num << " ";
}
cout << " (预期: 5)" << endl;
}

/**
 * 主函数
 */
int main() {
    cout << "==== 一维数组的动态和测试 ===" << endl;
    testRunningSum();
    return 0;
}

```

文件: Code01\_RunningSumOf1DArray.java

```

=====
package class046;

/**
 * 一维数组的动态和 (Running Sum of 1d Array)
 *
 * 题目描述:
 * 给你一个数组 nums 。数组「动态和」的计算公式为: runningSum[i] = sum(nums[0]…nums[i]) 。
 * 请返回 nums 的动态和。
 *
 * 示例:
 * 输入: nums = [1, 2, 3, 4]
 * 输出: [1, 3, 6, 10]
 * 解释: 动态和计算过程为 [1, 1+2, 1+2+3, 1+2+3+4] 。
 *
 * 输入: nums = [1, 1, 1, 1, 1]
 * 输出: [1, 2, 3, 4, 5]
 * 解释: 动态和计算过程为 [1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1] 。
 *

```

```
* 输入: nums = [3, 1, 2, 10, 1]
* 输出: [3, 4, 6, 16, 17]
*
* 提示:
* 1 <= nums.length <= 1000
* -10^6 <= nums[i] <= 10^6
*
* 题目链接: https://leetcode.com/problems/running-sum-of-1d-array/
*
* 解题思路:
* 使用前缀和的思想，从前向后累加即可。
*
* 时间复杂度: O(n) - 需要遍历数组一次
* 空间复杂度: O(1) - 不考虑输出数组，只使用常数额外空间
*
* 工程化考量:
* 1. 边界条件处理: 空数组、单元素数组
* 2. 原地修改: 节省空间，避免创建新数组
* 3. 整数溢出: 虽然题目保证在 32 位整数范围内，但实际工程中需要考虑
* 4. 代码可读性: 清晰的变量命名和注释
*
* 最优解分析:
* 这是最优解，因为必须遍历所有元素才能计算前缀和，时间复杂度 O(n) 无法优化。
* 空间复杂度 O(1) 也是最优的（不考虑输出数组）。
*
* 算法调试技巧:
* 1. 打印中间过程: 可以在循环中打印每个位置的前缀和
* 2. 边界测试: 测试空数组、单元素数组等特殊情况
* 3. 性能测试: 测试大规模数据下的性能表现
*
* 语言特性差异:
* Java 中数组是对象，可以直接修改原数组，但要注意并发安全问题。
* 与 C++ 相比，Java 有自动内存管理，无需手动释放数组内存。
* 与 Python 相比，Java 是静态类型语言，需要显式声明数组类型。
*/
public class Code01_RunningSumOf1DArray {

    /**
     * 计算数组的动态和
     *
     * @param nums 输入数组
     * @return 动态和数组
     */
}
```

```
* 异常场景处理:  
* - 空数组: 直接返回原数组  
* - 单元素数组: 直接返回原数组  
* - 大数组: 使用原地修改避免内存浪费  
*  
* 边界条件:  
* - 数组长度为 0 或 1  
* - 数组元素包含负数  
* - 数组元素包含大数 (可能溢出)  
*/  
  
public static int[] runningSum(int[] nums) {  
    // 边界情况处理: 空数组或单元素数组直接返回  
    if (nums == null || nums.length <= 1) {  
        return nums;  
    }  
  
    // 直接在原数组上进行修改, 节省空间  
    // 从第二个元素开始, 每个位置的值等于前一个位置的前缀和加上当前位置的原始值  
    for (int i = 1; i < nums.length; i++) {  
        // 调试打印: 显示中间过程  
        // System.out.println("位置 " + i + ": 前一个前缀和 = " + nums[i-1] + ", 当前值 = " +  
        nums[i]);  
        nums[i] += nums[i - 1];  
    }  
  
    return nums;  
}  
  
/**  
 * 单元测试方法  
 */  
public static void testRunningSum() {  
    System.out.println("== 一维数组的动态和单元测试 ==");  
  
    // 测试用例 1: 正常情况  
    int[] nums1 = {1, 2, 3, 4};  
    int[] result1 = runningSum(nums1.clone());  
    System.out.print("测试用例 1 [1,2,3,4]: ");  
    for (int num : result1) {  
        System.out.print(num + " ");  
    }  
    System.out.println(" (预期: 1 3 6 10)");
```

```
// 测试用例 2: 全 1 数组
int[] nums2 = {1, 1, 1, 1, 1};
int[] result2 = runningSum(nums2.clone());
System.out.print("测试用例 2 [1,1,1,1,1]: ");
for (int num : result2) {
    System.out.print(num + " ");
}
System.out.println(" (预期: 1 2 3 4 5)");

// 测试用例 3: 混合数值
int[] nums3 = {3, 1, 2, 10, 1};
int[] result3 = runningSum(nums3.clone());
System.out.print("测试用例 3 [3,1,2,10,1]: ");
for (int num : result3) {
    System.out.print(num + " ");
}
System.out.println(" (预期: 3 4 6 16 17)");

// 测试用例 4: 空数组
int[] nums4 = {};
int[] result4 = runningSum(nums4);
System.out.print("测试用例 4 []: ");
for (int num : result4) {
    System.out.print(num + " ");
}
System.out.println(" (预期: 空数组)");

// 测试用例 5: 单元素数组
int[] nums5 = {5};
int[] result5 = runningSum(nums5);
System.out.print("测试用例 5 [5]: ");
for (int num : result5) {
    System.out.print(num + " ");
}
System.out.println(" (预期: 5)");

// 测试用例 6: 包含负数
int[] nums6 = {-1, 2, -3, 4};
int[] result6 = runningSum(nums6.clone());
System.out.print("测试用例 6 [-1,2,-3,4]: ");
for (int num : result6) {
    System.out.print(num + " ");
}
```

```

        System.out.println(" (预期: -1 1 -2 2)");  

    }  
  

    /**  

     * 性能测试方法  

     */  

    public static void performanceTest() {  

        System.out.println("\n== 性能测试 ==");  

        int size = 1000000; // 100 万元素  

        int[] largeArray = new int[size];  
  

        // 初始化大数组  

        for (int i = 0; i < size; i++) {  

            largeArray[i] = i % 100; // 避免溢出  

        }  
  

        long startTime = System.currentTimeMillis();  

        runningSum(largeArray);  

        long endTime = System.currentTimeMillis();  
  

        System.out.println("处理 " + size + " 个元素耗时: " + (endTime - startTime) + "ms");  

    }  
  

    /**  

     * 主函数 - 测试入口  

     */  

    public static void main(String[] args) {  

        // 运行单元测试  

        testRunningSum();  
  

        // 运行性能测试 (可选)  

        // performanceTest();  
  

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

    }  

}
=====
```

文件: Code01\_RunningSumOf1DArray.py

=====

一维数组的动态和 (Running Sum of 1d Array)

## 题目描述：

给你一个数组 `nums`。数组「动态和」的计算公式为:  $\text{runningSum}[i] = \text{sum}(\text{nums}[0] \dots \text{nums}[i])$ 。  
请返回 `nums` 的动态和。

## 示例：

输入: `nums = [1, 2, 3, 4]`

输出: `[1, 3, 6, 10]`

解释: 动态和计算过程为 `[1, 1+2, 1+2+3, 1+2+3+4]`。

输入: `nums = [1, 1, 1, 1, 1]`

输出: `[1, 2, 3, 4, 5]`

解释: 动态和计算过程为 `[1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1]`。

输入: `nums = [3, 1, 2, 10, 1]`

输出: `[3, 4, 6, 16, 17]`

## 提示：

$1 \leq \text{nums.length} \leq 1000$

$-10^6 \leq \text{nums}[i] \leq 10^6$

题目链接: <https://leetcode.com/problems/running-sum-of-1d-array/>

## 解题思路：

使用前缀和的思想，从前向后累加即可。

时间复杂度:  $O(n)$  – 需要遍历数组一次

空间复杂度:  $O(1)$  – 不考虑输出数组，只使用常数额外空间

## 工程化考量：

1. 边界条件处理：空数组、单元素数组
2. 原地修改：节省空间，避免创建新数组
3. 整数溢出：虽然题目保证在 32 位整数范围内，但实际工程中需要考虑
4. 代码可读性：清晰的变量命名和注释

## 最优解分析：

这是最优解，因为必须遍历所有元素才能计算前缀和，时间复杂度  $O(n)$  无法优化。

空间复杂度  $O(1)$  也是最优的（不考虑输出数组）。

## 算法调试技巧：

1. 打印中间过程：可以在循环中打印每个位置的前缀和
2. 边界测试：测试空数组、单元素数组等特殊情况
3. 性能测试：测试大规模数据下的性能表现

语言特性差异：

Python 是动态类型语言，无需显式声明变量类型。

与 Java/C++相比，Python 有更简洁的语法和内置的列表操作。

Python 支持负索引和切片操作，但本算法不需要这些特性。

"""

```
class Solution:
```

```
    def runningSum(self, nums):
```

```
        """
```

计算数组的动态和

Args:

    nums (List[int]): 输入数组

Returns:

    List[int]: 动态和数组

异常场景处理：

- 空数组：直接返回原数组
- 单元素数组：直接返回原数组
- 大数组：使用原地修改避免内存浪费

边界条件：

- 数组长度为 0 或 1
- 数组元素包含负数
- 数组元素包含大数（可能溢出）

```
"""
```

# 边界情况处理：空数组或单元素数组直接返回

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

```
    return nums
```

# 直接在原数组上进行修改，节省空间

# 从第二个元素开始，每个位置的值等于前一个位置的前缀和加上当前位置的原始值

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

```
    # 调试打印：显示中间过程
```

```
    # print(f"位置 {i}: 前一个前缀和 = {nums[i-1]}, 当前值 = {nums[i]}")
```

```
    nums[i] += nums[i - 1]
```

```
return nums
```

```
def test_running_sum():
```

```
    """单元测试函数"""
```

```
print("== 一维数组的动态和单元测试 ==")
solution = Solution()

# 测试用例 1: 正常情况
nums1 = [1, 2, 3, 4]
result1 = solution.runningSum(nums1.copy())
print(f"测试用例 1 [1,2,3,4]: {result1} (预期: [1, 3, 6, 10])")

# 测试用例 2: 全 1 数组
nums2 = [1, 1, 1, 1, 1]
result2 = solution.runningSum(nums2.copy())
print(f"测试用例 2 [1,1,1,1,1]: {result2} (预期: [1, 2, 3, 4, 5])")

# 测试用例 3: 混合数值
nums3 = [3, 1, 2, 10, 1]
result3 = solution.runningSum(nums3.copy())
print(f"测试用例 3 [3,1,2,10,1]: {result3} (预期: [3, 4, 6, 16, 17])")

# 测试用例 4: 空数组
nums4 = []
result4 = solution.runningSum(nums4)
print(f"测试用例 4 []: {result4} (预期: [])")

# 测试用例 5: 单元素数组
nums5 = [5]
result5 = solution.runningSum(nums5)
print(f"测试用例 5 [5]: {result5} (预期: [5])")

# 测试用例 6: 包含负数
nums6 = [-1, 2, -3, 4]
result6 = solution.runningSum(nums6.copy())
print(f"测试用例 6 [-1,2,-3,4]: {result6} (预期: [-1, 1, -2, 2])")

def performance_test():
    """性能测试函数"""
    print("\n== 性能测试 ==")
    solution = Solution()
    size = 1000000 # 100 万元素
    large_array = list(range(size))

    import time
    start_time = time.time()
    solution.runningSum(large_array)
```

```
end_time = time.time()

print(f"处理 {size} 个元素耗时: {end_time - start_time:.4f}秒")

if __name__ == "__main__":
    # 运行单元测试
    test_running_sum()

    # 运行性能测试（可选）
    # performance_test()

print("\n==== 测试完成 ===")
```

=====

文件: Code02\_LongestSubarraySumEqualsAim.java

=====

```
package class046;

/**
 * 最长子数组和等于给定值 (Longest Subarray Sum Equals Aim)
 *
 * 题目描述:
 * 给定一个无序数组 arr, 其中元素可正、可负、可 0
 * 给定一个整数 aim
 * 求 arr 所有子数组中累加和为 aim 的最长子数组长度
 *
 * 示例:
 * 输入: arr = [1, -1, 5, -2, 3], aim = 3
 * 输出: 4
 * 解释: 子数组 [1, -1, 5, -2] 和等于 3, 且长度最长。
 *
 * 输入: arr = [-2, -1, 2, 1], aim = 1
 * 输出: 2
 * 解释: 子数组 [-1, 2] 和等于 1, 且长度最长。
 *
 * 提示:
 * 1 <= arr.length <= 10^5
 * -10^4 <= arr[i] <= 10^4
 * -10^5 <= aim <= 10^5
 *
 * 题目链接: https://www.nowcoder.com/practice/36fb0fd3c656480c92b569258a1223d5
 *
```

\* 解题思路：

\* 使用前缀和 + 哈希表的方法。

\* 1. 遍历数组，计算前缀和

\* 2. 对于当前位置的前缀和 sum，查找是否存在前缀和为( $sum - aim$ )的历史记录

\* 3. 如果存在，则说明存在子数组和为 aim

\* 4. 使用哈希表记录每个前缀和第一次出现的位置

\*

\* 时间复杂度：O(n) – 需要遍历数组一次

\* 空间复杂度：O(n) – 哈希表最多存储 n 个不同的前缀和

\*

\* 工程化考量：

\* 1. 边界条件处理：空数组、aim 值极端情况

\* 2. 哈希表选择：HashMap 提供 O(1) 的平均查找时间

\* 3. 位置记录：记录每个前缀和第一次出现的位置

\* 4. 性能优化：一次遍历完成所有计算

\*

\* 最优解分析：

\* 这是最优解，因为必须遍历所有元素才能找到最长子数组。

\* 哈希表方法将时间复杂度从 O( $n^2$ ) 优化到 O(n)。

\*

\* 算法核心：

\* 设 prefix[i] 为前 i 个元素的和，则子数组 [i, j] 的和为 prefix[j] – prefix[i-1] = aim

\* 即 prefix[j] – aim = prefix[i-1]，因此统计 prefix[j] – aim 第一次出现的位置即可。

\*

\* 算法调试技巧：

\* 1. 打印中间过程：可以在循环中打印每个位置的前缀和和哈希表状态

\* 2. 边界测试：测试空数组、aim=0、负数等情况

\* 3. 性能测试：测试大规模数据下的性能表现

\*

\* 语言特性差异：

\* Java 的 HashMap 自动处理哈希冲突，但需要注意哈希函数的选择。

\* 与 C++ 相比，Java 有自动内存管理，无需手动释放哈希表内存。

\* 与 Python 相比，Java 是静态类型语言，需要显式声明类型。

\*/

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.HashMap;
```

```
public class Code02_LongestSubarraySumEqualsAim {  
  
    public static int MAXN = 100001;  
  
    public static int[] arr = new int[MAXN];  
  
    public static int n, aim;  
  
    /**  
     * HashMap 用于存储前缀和及其第一次出现的位置  
     * key : 某个前缀和  
     * value : 这个前缀和最早出现的位置  
     */  
    public static HashMap<Integer, Integer> map = new HashMap<>();  
  
    public static void main(String[] args) throws IOException {  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
        StreamTokenizer in = new StreamTokenizer(br);  
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));  
        while (in.nextToken() != StreamTokenizer.TT_EOF) {  
            n = (int) in.nval;  
            in.nextToken();  
            aim = (int) in.nval;  
            for (int i = 0; i < n; i++) {  
                in.nextToken();  
                arr[i] = (int) in.nval;  
            }  
            out.println(compute());  
        }  
        out.flush();  
        out.close();  
        br.close();  
    }  
  
    /**  
     * 计算和为 aim 的最长子数组长度  
     *  
     * 异常场景处理：  
     * - 空数组：返回 0  
     * - aim 值极端：可能为极大值或极小值  
     * - 数组元素包含负数：算法本身支持  
     *  
     * 边界条件：  
     */
```

```

* - 数组长度为 0
* - aim=0 的情况
* - 数组元素全为 0 且 aim=0
*/
public static int compute() {
    map.clear();
    // 初始化: 前缀和为 0 在位置-1 出现 (便于计算长度)
    // 这样当 sum=aim 时, 长度计算为 i - (-1) = i+1
    map.put(0, -1);

    int ans = 0;          // 最长子数组长度
    int sum = 0;          // 当前前缀和

    // 遍历数组
    for (int i = 0; i < n; i++) {
        // 更新前缀和
        sum += arr[i];

        // 调试打印: 显示中间过程
        // System.out.println("位置 " + i + ": 值 = " + arr[i] + ", 前缀和 = " + sum);

        // 如果当前前缀和-aim 之前出现过, 更新最大长度
        if (map.containsKey(sum - aim)) {
            int length = i - map.get(sum - aim);
            ans = Math.max(ans, length);
            // 调试打印: 找到符合条件的子数组
            // System.out.println("找到子数组: 位置 " + (map.get(sum - aim) + 1) + " 到 " + i
+ ", 长度 = " + length);
        }
    }

    // 如果当前前缀和之前没有出现过, 记录第一次出现的位置
    if (!map.containsKey(sum)) {
        map.put(sum, i);
        // 调试打印: 记录新前缀和
        // System.out.println("记录新前缀和: " + sum + " -> " + i);
    }
}

return ans;
}

```

文件: Code02\_SubarraySumEqualsK.cpp

```
=====
/**  
 * 和为 K 的子数组 (Subarray Sum Equals K)  
 *  
 * 题目描述:  
 * 给你一个整数数组 nums 和一个整数 k, 请你统计并返回该数组中和为 k 的子数组的个数。  
 * 子数组是数组中元素的连续非空序列。  
 *  
 * 示例:  
 * 输入: nums = [1, 1, 1], k = 2  
 * 输出: 2  
 *  
 * 输入: nums = [1, 2, 3], k = 3  
 * 输出: 2  
 *  
 * 提示:  
 * 1 <= nums.length <= 2 * 10^4  
 * -1000 <= nums[i] <= 1000  
 * -10^7 <= k <= 10^7  
 *  
 * 题目链接: https://leetcode.com/problems/subarray-sum-equals-k/  
 *  
 * 解题思路:  
 * 使用前缀和 + 哈希表的方法。  
 * 1. 遍历数组, 计算前缀和  
 * 2. 对于当前位置的前缀和 sum, 查找是否存在前缀和为 (sum - k) 的历史记录  
 * 3. 如果存在, 则说明存在子数组和为 k  
 * 4. 使用哈希表记录每个前缀和出现的次数  
 *  
 * 时间复杂度: O(n) - 需要遍历数组一次  
 * 空间复杂度: O(n) - 哈希表最多存储 n 个不同的前缀和  
 *  
 * 工程化考量:  
 * 1. 边界条件处理: 空数组、k 值极端情况  
 * 2. 哈希表选择: unordered_map 提供 O(1) 的平均查找时间  
 * 3. 整数溢出: 使用 long long 避免大数溢出  
 * 4. 负数处理: k 可能为负数, 但算法本身支持负数  
 *  
 * 最优解分析:  
 * 这是最优解, 因为必须遍历所有元素才能统计所有子数组。
```

```

* 哈希表方法将时间复杂度从 O(n^2) 优化到 O(n)。
*
* 算法核心：
* 设 prefix[i] 为前 i 个元素的和，则子数组 [i, j] 的和为 prefix[j] - prefix[i-1] = k
* 即 prefix[j] - k = prefix[i-1]，因此统计 prefix[j] - k 出现的次数即可。
*/

```

```

#include <vector>
#include <unordered_map>
#include <iostream>

using namespace std;

class Solution {
public:
    /**
     * 计算和为 k 的子数组个数
     *
     * @param nums 输入数组
     * @param k 目标和
     * @return 和为 k 的子数组个数
     */
    int subarraySum(vector<int>& nums, int k) {
        // 边界情况处理
        if (nums.empty()) {
            return 0;
        }

        // 使用 unordered_map 记录前缀和及其出现次数
        // 初始化：前缀和为 0 出现 1 次（表示空数组）
        unordered_map<long long, int> prefixSumCount;
        prefixSumCount[0] = 1;

        int count = 0;           // 结果计数
        long long prefixSum = 0; // 当前前缀和，使用 long long 避免溢出

        // 遍历数组
        for (int num : nums) {
            // 更新前缀和
            prefixSum += num;

            // 查找是否存在前缀和为 (prefixSum - k) 的历史记录
            // 如果存在，说明存在子数组和为 k
            if (prefixSumCount.find(prefixSum - k) != prefixSumCount.end()) {
                count += prefixSumCount[prefixSum - k];
            }

            // 将当前前缀和的计数加 1
            prefixSumCount[prefixSum]++;
        }
    }
}

```

```
    if (prefixSumCount.find(prefixSum - k) != prefixSumCount.end()) {
        count += prefixSumCount[prefixSum - k];
    }

    // 更新当前前缀和的出现次数
    prefixSumCount[prefixSum]++;
}

return count;
}

};

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

    // 测试用例 1: 经典情况
    vector<int> nums1 = {1, 1, 1};
    int k1 = 2;
    int result1 = solution.subarraySum(nums1, k1);
    cout << "测试用例 1 [1,1,1] k=2: " << result1 << " (预期: 2)" << endl;

    // 测试用例 2: 多个子数组
    vector<int> nums2 = {1, 2, 3};
    int k2 = 3;
    int result2 = solution.subarraySum(nums2, k2);
    cout << "测试用例 2 [1,2,3] k=3: " << result2 << " (预期: 2)" << endl;

    // 测试用例 3: 包含 0 和负数
    vector<int> nums3 = {1, -1, 0};
    int k3 = 0;
    int result3 = solution.subarraySum(nums3, k3);
    cout << "测试用例 3 [1,-1,0] k=0: " << result3 << " (预期: 3)" << endl;

    // 测试用例 4: 单个元素
    vector<int> nums4 = {5};
    int k4 = 5;
    int result4 = solution.subarraySum(nums4, k4);
    cout << "测试用例 4 [5] k=5: " << result4 << " (预期: 1)" << endl;

    // 测试用例 5: 空数组
}
```

```

vector<int> nums5 = {};
int k5 = 1;
int result5 = solution.subarraySum(nums5, k5);
cout << "测试用例 5 [] k=1: " << result5 << " (预期: 0)" << endl;

// 测试用例 6: 大 k 值
vector<int> nums6 = {1, 2, 3};
int k6 = 100;
int result6 = solution.subarraySum(nums6, k6);
cout << "测试用例 6 [1,2,3] k=100: " << result6 << " (预期: 0)" << endl;
}

/**
 * 主函数
 */
int main() {
    cout << "==== 和为 K 的子数组测试 ===" << endl;
    testSubarraySum();
    return 0;
}

```

文件: Code02\_SubarraySumEqualsK.java

```

=====
package class046;

import java.util.HashMap;

/**
 * 和为 K 的子数组 (Subarray Sum Equals K)
 *
 * 题目描述:
 * 给你一个整数数组 nums 和一个整数 k, 请你统计并返回该数组中和为 k 的子数组的个数。
 * 子数组是数组中元素的连续非空序列。
 *
 * 示例:
 * 输入: nums = [1, 1, 1], k = 2
 * 输出: 2
 *
 * 输入: nums = [1, 2, 3], k = 3
 * 输出: 2
 *

```

\* 提示:

```
* 1 <= nums.length <= 2 * 10^4
* -1000 <= nums[i] <= 1000
* -10^7 <= k <= 10^7
*
* 题目链接: https://leetcode.com/problems/subarray-sum-equals-k/
```

\* 解题思路:

\* 使用前缀和 + 哈希表的方法。

- \* 1. 遍历数组，计算前缀和
- \* 2. 对于当前位置的前缀和 sum，查找是否存在前缀和为 (sum - k) 的历史记录
- \* 3. 如果存在，则说明存在子数组和为 k
- \* 4. 使用哈希表记录每个前缀和出现的次数

\*

\* 时间复杂度: O(n) – 需要遍历数组一次

\* 空间复杂度: O(n) – 哈希表最多存储 n 个不同的前缀和

\*

\* 工程化考量:

- \* 1. 边界条件处理: 空数组、k 值极端情况
- \* 2. 哈希表选择: HashMap 提供 O(1) 的平均查找时间
- \* 3. 整数溢出: 使用 long 避免大数溢出
- \* 4. 负数处理: k 可能为负数, 但算法本身支持负数

\*

\* 最优解分析:

\* 这是最优解, 因为必须遍历所有元素才能统计所有子数组。

\* 哈希表方法将时间复杂度从 O(n^2) 优化到 O(n)。

\*

\* 算法核心:

- \* 设 prefix[i] 为前 i 个元素的和, 则子数组 [i, j] 的和为 prefix[j] - prefix[i-1] = k
- \* 即 prefix[j] - k = prefix[i-1], 因此统计 prefix[j] - k 出现的次数即可。

\*

\* 算法调试技巧:

- \* 1. 打印中间过程: 可以在循环中打印每个位置的前缀和和哈希表状态
- \* 2. 边界测试: 测试空数组、k=0、负数等情况
- \* 3. 性能测试: 测试大规模数据下的性能表现

\*

\* 语言特性差异:

- \* Java 的 HashMap 自动处理哈希冲突, 但需要注意哈希函数的选择。
- \* 与 C++ 相比, Java 有自动内存管理, 无需手动释放哈希表内存。
- \* 与 Python 相比, Java 是静态类型语言, 需要显式声明类型。

\*/

```
public class Code02_SubarraySumEqualsK {
```

```
/**  
 * 计算和为 k 的子数组个数  
 *  
 * @param nums 输入数组  
 * @param k 目标和  
 * @return 和为 k 的子数组个数  
 *  
 * 异常场景处理：  
 * - 空数组：返回 0  
 * - k 值极端：可能为极大值或极小值  
 * - 数组元素包含负数：算法本身支持  
 *  
 * 边界条件：  
 * - 数组长度为 0  
 * - k=0 的情况（需要特殊处理空子数组）  
 * - 数组元素全为 0 且 k=0  
 */  
  
public static int subarraySum(int[] nums, int k) {  
    // 边界情况处理  
    if (nums == null || nums.length == 0) {  
        return 0;  
    }  
  
    // 使用 HashMap 记录前缀和及其出现次数  
    // 初始化：前缀和为 0 出现 1 次（表示空数组）  
    HashMap<Long, Integer> map = new HashMap<>();  
    map.put(0L, 1);  
  
    int count = 0;           // 结果计数  
    long prefixSum = 0;      // 当前前缀和，使用 long 避免溢出  
  
    // 遍历数组  
    for (int i = 0; i < nums.length; i++) {  
        // 更新前缀和  
        prefixSum += nums[i];  
  
        // 调试打印：显示中间过程  
        // System.out.println("位置 " + i + ": 前缀和 = " + prefixSum + ", 目标 = " +  
        (prefixSum - k));  
  
        // 查找是否存在前缀和为 (prefixSum - k) 的历史记录  
        // 如果存在，说明存在子数组和为 k  
        if (map.containsKey(prefixSum - k)) {
```

```
        count += map.get(prefixSum - k);
        // 调试打印: 找到子数组
        // System.out.println("找到子数组, 当前计数: " + count);
    }

    // 更新当前前缀和的出现次数
    map.put(prefixSum, map.getOrDefault(prefixSum, 0) + 1);

    // 调试打印: 哈希表状态
    // System.out.println("哈希表更新: " + prefixSum + " -> " + map.get(prefixSum));
}

return count;
}

/**
 * 单元测试方法
 */
public static void testSubarraySum() {
    System.out.println("== 和为 K 的子数组单元测试 ==");

    // 测试用例 1: 经典情况
    int[] nums1 = {1, 1, 1};
    int k1 = 2;
    int result1 = subarraySum(nums1, k1);
    System.out.println("测试用例 1 [1,1,1] k=2: " + result1 + " (预期: 2)");

    // 测试用例 2: 多个子数组
    int[] nums2 = {1, 2, 3};
    int k2 = 3;
    int result2 = subarraySum(nums2, k2);
    System.out.println("测试用例 2 [1,2,3] k=3: " + result2 + " (预期: 2)");

    // 测试用例 3: 包含 0 和负数
    int[] nums3 = {1, -1, 0};
    int k3 = 0;
    int result3 = subarraySum(nums3, k3);
    System.out.println("测试用例 3 [1,-1,0] k=0: " + result3 + " (预期: 3)");

    // 测试用例 4: 单个元素
    int[] nums4 = {5};
    int k4 = 5;
    int result4 = subarraySum(nums4, k4);
}
```

```

System.out.println("测试用例 4 [5] k=5: " + result4 + " (预期: 1)");

// 测试用例 5: 空数组
int[] nums5 = {};
int k5 = 1;
int result5 = subarraySum(nums5, k5);
System.out.println("测试用例 5 [] k=1: " + result5 + " (预期: 0)");

// 测试用例 6: 大 k 值
int[] nums6 = {1, 2, 3};
int k6 = 100;
int result6 = subarraySum(nums6, k6);
System.out.println("测试用例 6 [1,2,3] k=100: " + result6 + " (预期: 0)");

// 测试用例 7: 全 0 数组且 k=0
int[] nums7 = {0, 0, 0};
int k7 = 0;
int result7 = subarraySum(nums7, k7);
System.out.println("测试用例 7 [0,0,0] k=0: " + result7 + " (预期: 6)");
}

/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");
    int size = 100000; // 10 万元素
    int[] largeArray = new int[size];

    // 初始化大数组 (避免溢出)
    for (int i = 0; i < size; i++) {
        largeArray[i] = i % 100 - 50; // 包含正负数
    }

    long startTime = System.currentTimeMillis();
    int result = subarraySum(largeArray, 0); // 测试 k=0 的情况
    long endTime = System.currentTimeMillis();

    System.out.println("处理 " + size + " 个元素, 结果: " + result + ", 耗时: " + (endTime - startTime) + "ms");
}

/***

```

```
* 主函数 - 测试入口
*/
public static void main(String[] args) {
    // 运行单元测试
    testSubarraySum();

    // 运行性能测试（可选）
    // performanceTest();

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

=====

文件: Code02\_SubarraySumEqualsK.py

=====

```
"""
和为 K 的子数组 (Subarray Sum Equals K)
```

题目描述:

给你一个整数数组 `nums` 和一个整数 `k`, 请你统计并返回该数组中和为 `k` 的子数组的个数。  
子数组是数组中元素的连续非空序列。

示例:

输入: `nums = [1, 1, 1]`, `k = 2`

输出: 2

输入: `nums = [1, 2, 3]`, `k = 3`

输出: 2

提示:

`1 <= nums.length <= 2 * 10^4`

`-1000 <= nums[i] <= 1000`

`-10^7 <= k <= 10^7`

题目链接: <https://leetcode.com/problems/subarray-sum-equals-k/>

解题思路:

使用前缀和 + 哈希表的方法。

1. 遍历数组, 计算前缀和
2. 对于当前位置的前缀和 `sum`, 查找是否存在前缀和为  $(sum - k)$  的历史记录
3. 如果存在, 则说明存在子数组和为 `k`

#### 4. 使用哈希表记录每个前缀和出现的次数

时间复杂度:  $O(n)$  - 需要遍历数组一次

空间复杂度:  $O(n)$  - 哈希表最多存储  $n$  个不同的前缀和

工程化考量:

1. 边界条件处理: 空数组、 $k$  值极端情况
2. 哈希表选择: 字典提供  $O(1)$  的平均查找时间
3. 整数溢出: Python 自动处理大整数, 无需担心溢出
4. 负数处理:  $k$  可能为负数, 但算法本身支持负数

最优解分析:

这是最优解, 因为必须遍历所有元素才能统计所有子数组。

哈希表方法将时间复杂度从  $O(n^2)$  优化到  $O(n)$ 。

算法核心:

设  $\text{prefix}[i]$  为前  $i$  个元素的和, 则子数组  $[i, j]$  的和为  $\text{prefix}[j] - \text{prefix}[i-1] = k$   
即  $\text{prefix}[j] - k = \text{prefix}[i-1]$ , 因此统计  $\text{prefix}[j] - k$  出现的次数即可。

算法调试技巧:

1. 打印中间过程: 可以在循环中打印每个位置的前缀和和哈希表状态
2. 边界测试: 测试空数组、 $k=0$ 、负数等情况
3. 性能测试: 测试大规模数据下的性能表现

语言特性差异:

Python 是动态类型语言, 无需显式声明变量类型。

Python 字典自动处理哈希冲突, 语法简洁。

与 Java/C++ 相比, Python 有更简洁的语法和内置的数据结构操作。

"""

```
class Solution:
```

```
    def subarraySum(self, nums, k):
```

```
        """
```

```
        计算和为 k 的子数组个数
```

Args:

  nums (List[int]): 输入数组

  k (int): 目标和

Returns:

  int: 和为  $k$  的子数组个数

异常场景处理:

- 空数组：返回 0
- k 值极端：可能为极大值或极小值
- 数组元素包含负数：算法本身支持

边界条件：

- 数组长度为 0
- k=0 的情况（需要特殊处理空子数组）
- 数组元素全为 0 且 k=0

"""

# 边界情况处理

```
if not nums:
    return 0
```

# 使用字典记录前缀和及其出现次数

# 初始化：前缀和为 0 出现 1 次（表示空数组）

```
prefix_sum_count = {0: 1}
```

count = 0 # 结果计数

```
prefix_sum = 0  # 当前前缀和
```

# 遍历数组

```
for i, num in enumerate(nums):
```

# 更新前缀和

```
prefix_sum += num
```

# 调试打印：显示中间过程

```
# print(f"位置 {i}: 前缀和 = {prefix_sum}, 目标 = {prefix_sum - k}")
```

# 查找是否存在前缀和为 (prefix\_sum - k) 的历史记录

# 如果存在，说明存在子数组和为 k

```
if prefix_sum - k in prefix_sum_count:
```

```
    count += prefix_sum_count[prefix_sum - k]
```

# 调试打印：找到子数组

```
# print(f"找到子数组, 当前计数: {count}")
```

# 更新当前前缀和的出现次数

```
prefix_sum_count[prefix_sum] = prefix_sum_count.get(prefix_sum, 0) + 1
```

# 调试打印：字典状态

```
# print(f"字典更新: {prefix_sum} -> {prefix_sum_count[prefix_sum]}")
```

```
return count
```

```
def test_subarray_sum():
    """单元测试函数"""
    print("==== 和为 K 的子数组单元测试 ===")
    solution = Solution()

    # 测试用例 1: 经典情况
    nums1 = [1, 1, 1]
    k1 = 2
    result1 = solution.subarraySum(nums1, k1)
    print(f"测试用例 1 [1,1,1] k=2: {result1} (预期: 2)")

    # 测试用例 2: 多个子数组
    nums2 = [1, 2, 3]
    k2 = 3
    result2 = solution.subarraySum(nums2, k2)
    print(f"测试用例 2 [1,2,3] k=3: {result2} (预期: 2)")

    # 测试用例 3: 包含 0 和负数
    nums3 = [1, -1, 0]
    k3 = 0
    result3 = solution.subarraySum(nums3, k3)
    print(f"测试用例 3 [1,-1,0] k=0: {result3} (预期: 3)")

    # 测试用例 4: 单个元素
    nums4 = [5]
    k4 = 5
    result4 = solution.subarraySum(nums4, k4)
    print(f"测试用例 4 [5] k=5: {result4} (预期: 1)")

    # 测试用例 5: 空数组
    nums5 = []
    k5 = 1
    result5 = solution.subarraySum(nums5, k5)
    print(f"测试用例 5 [] k=1: {result5} (预期: 0)")

    # 测试用例 6: 大 k 值
    nums6 = [1, 2, 3]
    k6 = 100
    result6 = solution.subarraySum(nums6, k6)
    print(f"测试用例 6 [1,2,3] k=100: {result6} (预期: 0)")

    # 测试用例 7: 全 0 数组且 k=0
    nums7 = [0, 0, 0]
```

```

k7 = 0
result7 = solution.subarraySum(nums7, k7)
print(f"测试用例 7 [0,0,0] k=0: {result7} (预期: 6)")

def performance_test():
    """性能测试函数"""
    print("\n==== 性能测试 ===")
    solution = Solution()
    size = 20000 # 2 万元素 (题目上限)
    large_array = [i % 100 - 50 for i in range(size)] # 包含正负数

    import time
    start_time = time.time()
    result = solution.subarraySum(large_array, 0) # 测试 k=0 的情况
    end_time = time.time()

    print(f"处理 {size} 个元素, 结果: {result}, 耗时: {end_time - start_time:.4f} 秒")

if __name__ == "__main__":
    # 运行单元测试
    test_subarray_sum()

    # 运行性能测试 (可选)
    # performance_test()

    print("\n==== 测试完成 ===")

```

文件: Code03\_ContiguousArray.cpp

```

=====
/* 
 * 连续数组 (Contiguous Array)
 *
 * 题目描述:
 * 给定一个二进制数组 nums，找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。
 *
 * 示例:
 * 输入: nums = [0,1]
 * 输出: 2
 * 说明: [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。
 *
 * 输入: nums = [0,1,0]

```

```
* 输出: 2
* 说明: [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。
*
* 提示:
* 1 <= nums.length <= 10^5
* nums[i] 不是 0 就是 1
*
* 题目链接: https://leetcode.com/problems/contiguous-array/
*
* 解题思路:
* 1. 将 0 看作 -1, 问题转化为求和为 0 的最长子数组
* 2. 使用前缀和 + 哈希表的方法
* 3. 遍历数组, 计算前缀和
* 4. 如果某个前缀和之前出现过, 说明这两个位置之间的子数组和为 0
* 5. 使用哈希表记录每个前缀和第一次出现的位置
*
* 时间复杂度: O(n) - 需要遍历数组一次
* 空间复杂度: O(n) - 哈希表最多存储 n 个不同的前缀和
*
* 工程化考量:
* 1. 边界条件处理: 空数组、单元素数组
* 2. 哈希表初始化: 前缀和为 0 在位置 -1 出现, 便于计算长度
* 3. 映射技巧: 0 → -1, 1 → 1 的转换是关键
* 4. 性能优化: 使用 unordered_map 的 O(1) 查找时间
*
* 最优解分析:
* 这是最优解, 因为必须遍历所有元素才能找到最长子数组。
* 哈希表方法将时间复杂度从 O(n^2) 优化到 O(n)。
*
* 算法核心:
* 设 count 为前缀和 (0 → -1, 1 → 1), 当 count[i] = count[j] 时, 子数组 [i+1, j] 的和为 0。
* 即 0 和 1 的数量相等。
*/

```

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

```
using namespace std;
```

```
class Solution {
public:
```

```
/**  
 * 找到含有相同数量 0 和 1 的最长连续子数组的长度  
 *  
 * @param nums 输入的二进制数组  
 * @return 最长连续子数组的长度  
 *  
 * 异常场景处理：  
 * - 空数组：返回 0  
 * - 单元素数组：返回 0（不可能有相同数量的 0 和 1）  
 * - 全 0 或全 1 数组：返回 0  
 */  
  
int findMaxLength(vector<int>& nums) {  
    // 边界情况处理  
    if (nums.empty() || nums.size() <= 1) {  
        return 0;  
    }  
  
    // 使用 unordered_map 记录前缀和及其第一次出现的位置  
    unordered_map<int, int> map;  
    // 初始化：前缀和为 0 在位置-1 出现（便于计算长度）  
    map[0] = -1;  
  
    int maxLength = 0; // 最长子数组长度  
    int count = 0; // 当前前缀和（0 看作-1， 1 看作 1）  
  
    // 遍历数组  
    for (int i = 0; i < nums.size(); i++) {  
        // 更新前缀和：0 看作-1， 1 看作 1  
        count += (nums[i] == 0) ? -1 : 1;  
  
        // 如果当前前缀和之前出现过，更新最大长度  
        if (map.find(count) != map.end()) {  
            int length = i - map[count];  
            maxLength = max(maxLength, length);  
        } else {  
            // 记录当前前缀和第一次出现的位置  
            map[count] = i;  
        }  
    }  
  
    return maxLength;  
}  
};
```

```
/**  
 * 测试函数  
 */  
  
void testFindMaxLength() {  
    Solution solution;  
  
    // 测试用例 1: 基础情况  
    vector<int> nums1 = {0, 1};  
    int result1 = solution.findMaxLength(nums1);  
    cout << "测试用例 1 [0,1]: " << result1 << " (预期: 2)" << endl;  
  
    // 测试用例 2: 三个元素  
    vector<int> nums2 = {0, 1, 0};  
    int result2 = solution.findMaxLength(nums2);  
    cout << "测试用例 2 [0,1,0]: " << result2 << " (预期: 2)" << endl;  
  
    // 测试用例 3: 复杂情况  
    vector<int> nums3 = {0, 0, 1, 0, 0, 1, 1, 0};  
    int result3 = solution.findMaxLength(nums3);  
    cout << "测试用例 3 [0,0,1,0,0,1,1,0]: " << result3 << " (预期: 6)" << endl;  
  
    // 测试用例 4: 全 0 数组  
    vector<int> nums4 = {0, 0, 0};  
    int result4 = solution.findMaxLength(nums4);  
    cout << "测试用例 4 [0,0,0]: " << result4 << " (预期: 0)" << endl;  
  
    // 测试用例 5: 全 1 数组  
    vector<int> nums5 = {1, 1, 1};  
    int result5 = solution.findMaxLength(nums5);  
    cout << "测试用例 5 [1,1,1]: " << result5 << " (预期: 0)" << endl;  
  
    // 测试用例 6: 空数组  
    vector<int> nums6 = {};  
    int result6 = solution.findMaxLength(nums6);  
    cout << "测试用例 6 []: " << result6 << " (预期: 0)" << endl;  
  
    // 测试用例 7: 单元素数组  
    vector<int> nums7 = {0};  
    int result7 = solution.findMaxLength(nums7);  
    cout << "测试用例 7 [0]: " << result7 << " (预期: 0)" << endl;  
  
    // 测试用例 8: 交替数组
```

```
vector<int> nums8 = {0, 1, 0, 1, 0, 1};
int result8 = solution.findMaxLength(nums8);
cout << "测试用例 8 [0,1,0,1,0,1]: " << result8 << " (预期: 6)" << endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
    cout << "\n==== 性能测试 ===" << endl;
    Solution solution;
    int size = 100000; // 10 万元素
    vector<int> largeArray(size);

    // 初始化大数组 (交替 0 和 1)
    for (int i = 0; i < size; i++) {
        largeArray[i] = i % 2;
    }

    auto startTime = chrono::high_resolution_clock::now();
    int result = solution.findMaxLength(largeArray);
    auto endTime = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);
    cout << "处理 " << size << " 个元素, 最长子数组长度: " << result << ", 耗时: " <<
duration.count() << "ms" << endl;
}

/***
 * 主函数
 */
int main() {
    cout << "==== 连续数组测试 ===" << endl;
    testFindMaxLength();

    // 运行性能测试 (可选)
    // performanceTest();

    cout << "\n==== 测试完成 ===" << endl;
    return 0;
}
```

=====

文件: Code03\_ContiguousArray.java

```
=====
```

```
package class046;
```

```
import java.util.HashMap;
```

```
/**
```

```
* 连续数组 (Contiguous Array)
```

```
*
```

```
* 题目描述:
```

```
* 给定一个二进制数组 nums，找到含有相同数量的 0 和 1 的最长连续子数组，并返回该子数组的长度。
```

```
*
```

```
* 示例:
```

```
* 输入: nums = [0, 1]
```

```
* 输出: 2
```

```
* 说明: [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。
```

```
*
```

```
* 输入: nums = [0, 1, 0]
```

```
* 输出: 2
```

```
* 说明: [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。
```

```
*
```

```
* 提示:
```

```
* 1 <= nums.length <= 10^5
```

```
* nums[i] 不是 0 就是 1
```

```
*
```

```
* 题目链接: https://leetcode.com/problems/contiguous-array/
```

```
*
```

```
* 解题思路:
```

```
* 1. 将 0 看作 -1，问题转化为求和为 0 的最长子数组
```

```
* 2. 使用前缀和 + 哈希表的方法
```

```
* 3. 遍历数组，计算前缀和
```

```
* 4. 如果某个前缀和之前出现过，说明这两个位置之间的子数组和为 0
```

```
* 5. 使用哈希表记录每个前缀和第一次出现的位置
```

```
*
```

```
* 时间复杂度: O(n) - 需要遍历数组一次
```

```
* 空间复杂度: O(n) - 哈希表最多存储 n 个不同的前缀和
```

```
*
```

```
* 工程化考量:
```

```
* 1. 边界条件处理: 空数组、单元素数组
```

```
* 2. 哈希表初始化: 前缀和为 0 在位置 -1 出现，便于计算长度
```

```
* 3. 映射技巧: 0 → -1, 1 → 1 的转换是关键
```

```
* 4. 性能优化: 使用 HashMap 的 O(1) 查找时间
```

```
*  
* 最优解分析:  
* 这是最优解，因为必须遍历所有元素才能找到最长子数组。  
* 哈希表方法将时间复杂度从  $O(n^2)$  优化到  $O(n)$ 。  
*  
* 算法核心:  
* 设 count 为前缀和 ( $0 \rightarrow -1$ ,  $1 \rightarrow 1$ )，当  $count[i] = count[j]$  时，子数组  $[i+1, j]$  的和为 0。  
* 即 0 和 1 的数量相等。  
*  
* 算法调试技巧:  
* 1. 打印中间过程：可以在循环中打印每个位置的前缀和和哈希表状态  
* 2. 边界测试：测试全 0、全 1、交替等情况  
* 3. 性能测试：测试大规模数据下的性能表现  
*  
* 语言特性差异：  
* Java 的 HashMap 自动处理哈希冲突，但需要注意哈希函数的选择。  
* 与 C++ 相比，Java 有自动内存管理，无需手动释放哈希表内存。  
* 与 Python 相比，Java 是静态类型语言，需要显式声明类型。  
*/  
  
public class Code03_ContiguousArray {  
  
    /**  
     * 找到含有相同数量 0 和 1 的最长连续子数组的长度  
     *  
     * @param nums 输入的二进制数组  
     * @return 最长连续子数组的长度  
     *  
     * 异常场景处理：  
     * - 空数组：返回 0  
     * - 单元素数组：返回 0 (不可能有相同数量的 0 和 1)  
     * - 全 0 或全 1 数组：返回 0  
     *  
     * 边界条件：  
     * - 数组长度为 0 或 1  
     * - 数组元素全为 0 或全为 1  
     * - 数组元素交替出现  
     */  
  
    public static int findMaxLength(int[] nums) {  
        // 边界情况处理  
        if (nums == null || nums.length <= 1) {  
            return 0;  
        }  
    }  
}
```

```
// 哈希表记录前缀和及其第一次出现的位置
HashMap<Integer, Integer> map = new HashMap<>();
// 初始化: 前缀和为 0 在位置-1 出现 (便于计算长度)
// 这样当 count=0 时, 长度计算为 i - (-1) = i+1
map.put(0, -1);

int maxLength = 0; // 最长子数组长度
int count = 0; // 当前前缀和 (0 看作-1, 1 看作 1)

// 遍历数组
for (int i = 0; i < nums.length; i++) {
    // 更新前缀和: 0 看作-1, 1 看作 1
    count += (nums[i] == 0) ? -1 : 1;

    // 调试打印: 显示中间过程
    // System.out.println("位置 " + i + ": 值 = " + nums[i] + ", 前缀和 = " + count);

    // 如果当前前缀和之前出现过, 更新最大长度
    if (map.containsKey(count)) {
        int length = i - map.get(count);
        maxLength = Math.max(maxLength, length);
        // 调试打印: 找到符合条件的子数组
        // System.out.println("找到子数组: 位置 " + (map.get(count) + 1) + " 到 " + i +
    ", 长度 = " + length);
    } else {
        // 记录当前前缀和第一次出现的位置
        map.put(count, i);
        // 调试打印: 记录新前缀和
        // System.out.println("记录新前缀和: " + count + " -> " + i);
    }
}

return maxLength;
}

/**
 * 单元测试方法
 */
public static void testFindMaxLength() {
    System.out.println("==== 连续数组单元测试 ===");

    // 测试用例 1: 基础情况
    int[] nums1 = {0, 1};
```

```
int result1 = findMaxLength(nums1);
System.out.println("测试用例 1 [0, 1]: " + result1 + " (预期: 2)");

// 测试用例 2: 三个元素
int[] nums2 = {0, 1, 0};
int result2 = findMaxLength(nums2);
System.out.println("测试用例 2 [0, 1, 0]: " + result2 + " (预期: 2)");

// 测试用例 3: 复杂情况
int[] nums3 = {0, 0, 1, 0, 0, 1, 1, 0};
int result3 = findMaxLength(nums3);
System.out.println("测试用例 3 [0, 0, 1, 0, 0, 1, 1, 0]: " + result3 + " (预期: 6)");

// 测试用例 4: 全 0 数组
int[] nums4 = {0, 0, 0};
int result4 = findMaxLength(nums4);
System.out.println("测试用例 4 [0, 0, 0]: " + result4 + " (预期: 0)");

// 测试用例 5: 全 1 数组
int[] nums5 = {1, 1, 1};
int result5 = findMaxLength(nums5);
System.out.println("测试用例 5 [1, 1, 1]: " + result5 + " (预期: 0)");

// 测试用例 6: 空数组
int[] nums6 = {};
int result6 = findMaxLength(nums6);
System.out.println("测试用例 6 []: " + result6 + " (预期: 0)");

// 测试用例 7: 单元素数组
int[] nums7 = {0};
int result7 = findMaxLength(nums7);
System.out.println("测试用例 7 [0]: " + result7 + " (预期: 0)");

// 测试用例 8: 交替数组
int[] nums8 = {0, 1, 0, 1, 0, 1};
int result8 = findMaxLength(nums8);
System.out.println("测试用例 8 [0, 1, 0, 1, 0, 1]: " + result8 + " (预期: 6)");
}

/**
 * 性能测试方法
 */
public static void performanceTest() {
```

```

System.out.println("\n==== 性能测试 ===");
int size = 100000; // 10 万元素
int[] largeArray = new int[size];

// 初始化大数组（交替 0 和 1）
for (int i = 0; i < size; i++) {
    largeArray[i] = i % 2;
}

long startTime = System.currentTimeMillis();
int result = findMaxLength(largeArray);
long endTime = System.currentTimeMillis();

System.out.println("处理 " + size + " 个元素，最长子数组长度: " + result + ", 耗时: " +
(endTime - startTime) + "ms");
}

/**
 * 主函数 - 测试入口
 */
public static void main(String[] args) {
    // 运行单元测试
    testFindMaxLength();

    // 运行性能测试（可选）
    // performanceTest();

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

文件: Code03\_ContiguousArray.py

```
"""
连续数组 (Contiguous Array)
```

题目描述:

给定一个二进制数组 `nums`, 找到含有相同数量的 0 和 1 的最长连续子数组, 并返回该子数组的长度。

示例:

输入: `nums = [0, 1]`

输出: 2

说明: [0, 1] 是具有相同数量 0 和 1 的最长连续子数组。

输入: nums = [0, 1, 0]

输出: 2

说明: [0, 1] (或 [1, 0]) 是具有相同数量 0 和 1 的最长连续子数组。

提示:

$1 \leq \text{nums.length} \leq 10^5$

nums[i] 不是 0 就是 1

题目链接: <https://leetcode.com/problems/contiguous-array/>

解题思路:

1. 将 0 看作 -1，问题转化为求和为 0 的最长子数组
2. 使用前缀和 + 哈希表的方法
3. 遍历数组，计算前缀和
4. 如果某个前缀和之前出现过，说明这两个位置之间的子数组和为 0
5. 使用哈希表记录每个前缀和第一次出现的位置

时间复杂度:  $O(n)$  – 需要遍历数组一次

空间复杂度:  $O(n)$  – 哈希表最多存储  $n$  个不同的前缀和

工程化考量:

1. 边界条件处理: 空数组、单元素数组
2. 映射技巧:  $0 \rightarrow -1$ ,  $1 \rightarrow 1$  的数学变换
3. 哈希表初始化: 空前缀和为 0 出现在位置 -1
4. 性能优化: 一次遍历完成所有计算

最优解分析:

这是最优解，因为必须遍历所有元素才能找到最长子数组。

哈希表方法将时间复杂度从  $O(n^2)$  优化到  $O(n)$ 。

算法核心:

设 count 为前缀和 ( $0 \rightarrow -1$ ,  $1 \rightarrow 1$ )，则子数组  $[i, j]$  满足:

$\text{count}[j] - \text{count}[i-1] = 0 \rightarrow \text{count}[j] = \text{count}[i-1]$

因此记录每个 count 值第一次出现的位置，找到相同 count 值的最远距离。

算法调试技巧:

1. 打印中间过程: 可以在循环中打印每个位置的前缀和和哈希表状态
2. 边界测试: 测试空数组、全 0 数组、全 1 数组等特殊情况
3. 性能测试: 测试大规模数据下的性能表现

语言特性差异：

Python 是动态类型语言，无需显式声明变量类型。

Python 字典自动处理哈希冲突，语法简洁。

与 Java/C++ 相比，Python 有更简洁的语法和内置的数据结构操作。

"""

class Solution:

def findMaxLength(self, nums):

"""

找到含有相同数量 0 和 1 的最长连续子数组的长度

Args:

nums (List[int]): 输入的二进制数组

Returns:

int: 最长连续子数组的长度

异常场景处理：

- 空数组：返回 0
- 单元素数组：返回 0（不可能有相同数量的 0 和 1）
- 全 0 或全 1 数组：返回 0

边界条件：

- 数组长度为 0 或 1
- 数组元素全为 0 或全为 1
- 数组元素包含非 0 非 1 值（题目保证只有 0 和 1）

"""

# 边界情况处理

if not nums or len(nums) <= 1:

return 0

# 字典记录前缀和及其第一次出现的位置

# 初始化：前缀和为 0 在位置 -1 出现（便于计算长度）

# 数学原理： $\text{count}[j] - \text{count}[i-1] = 0 \rightarrow \text{count}[j] = \text{count}[i-1]$

prefix\_sum\_index = {0: -1}

max\_length = 0 # 最长子数组长度

sum\_val = 0 # 当前前缀和 (0 看作 -1)

# 遍历数组，时间复杂度 O(n)

for i, num in enumerate(nums):

# 更新前缀和：0 看作 -1, 1 看作 1

# 这样相同数量的 0 和 1 对应和为 0

```

    sum_val += -1 if num == 0 else 1

    # 调试打印: 显示中间过程 (调试时取消注释)
    # print(f"位置 {i}: 数字 = {num}, 前缀和 = {sum_val}")

    # 如果当前前缀和之前出现过, 更新最大长度
    if sum_val in prefix_sum_index:
        # 计算当前子数组长度: i - 第一次出现的位置
        length = i - prefix_sum_index[sum_val]
        max_length = max(max_length, length)

        # 调试打印: 找到子数组 (调试时取消注释)
        # print(f"找到子数组 [{prefix_sum_index[sum_val]+1}, {i}], 长度 = {length}")

    else:
        # 记录当前前缀和第一次出现的位置
        prefix_sum_index[sum_val] = i

        # 调试打印: 记录新前缀和 (调试时取消注释)
        # print(f"记录前缀和 {sum_val} 出现在位置 {i}")

    return max_length

```

```

def test_contiguous_array():
    """单元测试函数"""
    print("== 连续数组单元测试 ==")
    solution = Solution()

    # 测试用例 1: 基础情况
    nums1 = [0, 1]
    result1 = solution.findMaxLength(nums1)
    print(f"测试用例 1 [0,1]: {result1} (预期: 2)")

    # 测试用例 2: 多个元素
    nums2 = [0, 1, 0]
    result2 = solution.findMaxLength(nums2)
    print(f"测试用例 2 [0,1,0]: {result2} (预期: 2)")

    # 测试用例 3: 完整数组
    nums3 = [0, 1, 0, 1]
    result3 = solution.findMaxLength(nums3)
    print(f"测试用例 3 [0,1,0,1]: {result3} (预期: 4)")

    # 测试用例 4: 空数组
    nums4 = []

```

```
result4 = solution.findMaxLength(nums4)
print(f"测试用例 4 []: {result4} (预期: 0)")

# 测试用例 5: 单元素数组
nums5 = [0]
result5 = solution.findMaxLength(nums5)
print(f"测试用例 5 [0]: {result5} (预期: 0)")

# 测试用例 6: 全 0 数组
nums6 = [0, 0, 0]
result6 = solution.findMaxLength(nums6)
print(f"测试用例 6 [0,0,0]: {result6} (预期: 0)")

# 测试用例 7: 全 1 数组
nums7 = [1, 1, 1]
result7 = solution.findMaxLength(nums7)
print(f"测试用例 7 [1,1,1]: {result7} (预期: 0)")

# 测试用例 8: 复杂情况
nums8 = [0, 1, 0, 0, 1, 1, 0]
result8 = solution.findMaxLength(nums8)
print(f"测试用例 8 [0,1,0,0,1,1,0]: {result8} (预期: 6)")

def performance_test():
    """性能测试函数"""
    print("\n==== 性能测试 ====")
    solution = Solution()
    size = 100000 # 10 万元素
    import random
    large_array = [random.randint(0, 1) for _ in range(size)]

    import time
    start_time = time.time()
    result = solution.findMaxLength(large_array)
    end_time = time.time()

    print(f"处理 {size} 个元素, 最长子数组长度: {result}, 耗时: {end_time - start_time:.4f}秒")

if __name__ == "__main__":
    # 运行单元测试
    test_contiguous_array()

    # 运行性能测试 (可选)
```

```
# performance_test()  
  
print("\n==== 测试完成 ===")
```

=====

文件: Code03\_NumberOfSubarraySumEqualsAim.java

=====

```
package class046;  
  
import java.util.HashMap;  
  
/**  
 * 和为 K 的子数组个数 (Number of Subarray Sum Equals Aim)  
 *  
 * 题目描述:  
 * 给定一个整数数组和一个整数 k，你需要找到该数组中和为 k 的连续的子数组的个数。  
 *  
 * 示例:  
 * 输入:nums = [1,1,1], k = 2  
 * 输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。  
 *  
 * 输入:nums = [1,2,3], k = 3  
 * 输出: 2  
 *  
 * 提示:  
 * 1 <= nums.length <= 20000  
 * -1000 <= nums[i] <= 1000  
 * -10^7 <= k <= 10^7  
 *  
 * 题目链接: https://leetcode.cn/problems/subarray-sum-equals-k/  
 *  
 * 解题思路:  
 * 使用前缀和 + 哈希表的方法。  
 * 1. 遍历数组，计算前缀和  
 * 2. 对于当前位置的前缀和 sum，查找是否存在前缀和为 (sum - k) 的历史记录  
 * 3. 如果存在，则说明存在子数组和为 k  
 * 4. 使用哈希表记录每个前缀和出现的次数  
 *  
 * 时间复杂度: O(n) - 需要遍历数组一次  
 * 空间复杂度: O(n) - 哈希表最多存储 n 个不同的前缀和  
 *  
 * 工程化考量:
```

- \* 1. 边界条件处理：空数组、k 值极端情况
- \* 2. 哈希表选择：HashMap 提供 O(1) 的平均查找时间
- \* 3. 整数溢出：使用 long 避免大数溢出
- \* 4. 负数处理：k 可能为负数，但算法本身支持负数
- \*
- \* 最优解分析：
- \* 这是最优解，因为必须遍历所有元素才能统计所有子数组。
- \* 哈希表方法将时间复杂度从  $O(n^2)$  优化到  $O(n)$ 。
- \*
- \* 算法核心：
- \* 设 prefix[i] 为前 i 个元素的和，则子数组 [i, j] 的和为 prefix[j] - prefix[i-1] = k
- \* 即 prefix[j] - k = prefix[i-1]，因此统计 prefix[j] - k 出现的次数即可。
- \*
- \* 算法调试技巧：
- \* 1. 打印中间过程：可以在循环中打印每个位置的前缀和和哈希表状态
- \* 2. 边界测试：测试空数组、k=0、负数等情况
- \* 3. 性能测试：测试大规模数据下的性能表现
- \*
- \* 语言特性差异：
- \* Java 的 HashMap 自动处理哈希冲突，但需要注意哈希函数的选择。
- \* 与 C++ 相比，Java 有自动内存管理，无需手动释放哈希表内存。
- \* 与 Python 相比，Java 是静态类型语言，需要显式声明类型。

\*/

```
public class Code03_NumberOfSubarraySumEqualsAim {
```

```
    /**
     * 计算和为 aim 的子数组个数
     *
     * @param nums 输入数组
     * @param aim 目标和
     * @return 和为 aim 的子数组个数
     *
     * 异常场景处理：
     * - 空数组：返回 0
     * - aim 值极端：可能为极大值或极小值
     * - 数组元素包含负数：算法本身支持
     *
     * 边界条件：
     * - 数组长度为 0
     * - aim=0 的情况（需要特殊处理空子数组）
     * - 数组元素全为 0 且 aim=0
     */

```

```
    public static int subarraySum(int[] nums, int aim) {
```

```
// 边界情况处理
if (nums == null || nums.length == 0) {
    return 0;
}

// 使用 HashMap 记录前缀和及其出现次数
// 初始化：前缀和为 0 出现 1 次（表示空数组）
HashMap<Integer, Integer> map = new HashMap<>();
map.put(0, 1);

int ans = 0;           // 结果计数
int prefixSum = 0;     // 当前前缀和

// 遍历数组
for (int i = 0; i < nums.length; i++) {
    // 更新前缀和
    prefixSum += nums[i];

    // 调试打印：显示中间过程
    // System.out.println("位置 " + i + ": 前缀和 = " + prefixSum + ", 目标 = " +
(prefixSum - aim));

    // 查找是否存在前缀和为(prefixSum - aim)的历史记录
    // 如果存在，说明存在子数组和为aim
    if (map.containsKey(prefixSum - aim)) {
        ans += map.get(prefixSum - aim);
        // 调试打印：找到子数组
        // System.out.println("找到子数组, 当前计数: " + ans);
    }

    // 更新当前前缀和的出现次数
    map.put(prefixSum, map.getOrDefault(prefixSum, 0) + 1);

    // 调试打印：哈希表状态
    // System.out.println("哈希表更新: " + prefixSum + " -> " + map.get(prefixSum));
}

return ans;
}
```

=====

文件: Code04\_PositivesEqualsNegativesLongestSubarray.java

```
=====
package class046;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;
import java.util.HashMap;

/**
 * 正负数个数相等的最长子数组 (Positives Equals Negatives Longest Subarray)
 *
 * 题目描述:
 * 给定一个无序数组 arr，其中元素可正、可负、可 0
 * 求 arr 所有子数组中正数与负数个数相等的最长子数组的长度
 *
 * 示例:
 * 输入: arr = [1, -1, 0, 1, -1]
 * 输出: 4
 * 解释: 子数组 [1, -1, 0, 1] 或 [1, -1, 0, 1, -1] 中正数和负数个数相等。
 *
 * 输入: arr = [1, 1, -1, -1, 0]
 * 输出: 4
 * 解释: 子数组 [1, 1, -1, -1] 中正数和负数个数相等。
 *
 * 提示:
 * 1 <= arr.length <= 10^5
 * -10^4 <= arr[i] <= 10^4
 *
 * 题目链接: https://www.nowcoder.com/practice/545544c060804eceaed0bb84fc992fb
 *
 * 解题思路:
 * 1. 将正数看作 1，负数看作 -1，0 看作 0，问题转化为求和为 0 的最长子数组
 * 2. 使用前缀和 + 哈希表的方法
 * 3. 遍历数组，计算前缀和
 * 4. 如果某个前缀和之前出现过，说明这两个位置之间的子数组和为 0
 * 5. 使用哈希表记录每个前缀和第一次出现的位置
 *
 * 时间复杂度: O(n) - 需要遍历数组一次
```

- \* 空间复杂度:  $O(n)$  - 哈希表最多存储  $n$  个不同的前缀和
- \*
- \* 工程化考量:
  - \* 1. 边界条件处理: 空数组、单元素数组
  - \* 2. 哈希表初始化: 前缀和为 0 在位置-1 出现, 便于计算长度
  - \* 3. 映射技巧: 正数 $\rightarrow 1$ , 负数 $\rightarrow -1$ , 0 $\rightarrow 0$  的转换是关键
  - \* 4. 性能优化: 使用 HashMap 的  $O(1)$  查找时间
- \*
- \* 最优解分析:
  - \* 这是最优解, 因为必须遍历所有元素才能找到最长子数组。
  - \* 哈希表方法将时间复杂度从  $O(n^2)$  优化到  $O(n)$ 。
- \*
- \* 算法核心:
  - \* 设 count 为前缀和 (正数 $\rightarrow 1$ , 负数 $\rightarrow -1$ , 0 $\rightarrow 0$ ), 当  $count[i] = count[j]$  时, 子数组  $[i+1, j]$  的和为 0。
  - \* 即正数和负数的数量相等。
- \*
- \* 算法调试技巧:
  - \* 1. 打印中间过程: 可以在循环中打印每个位置的前缀和和哈希表状态
  - \* 2. 边界测试: 测试全正数、全负数、交替等情况
  - \* 3. 性能测试: 测试大规模数据下的性能表现
- \*
- \* 语言特性差异:
  - \* Java 的 HashMap 自动处理哈希冲突, 但需要注意哈希函数的选择。
  - \* 与 C++ 相比, Java 有自动内存管理, 无需手动释放哈希表内存。
  - \* 与 Python 相比, Java 是静态类型语言, 需要显式声明类型。

```
*/  
public class Code04_PositivesEqualsNegativesLongestSubarray {  
  
    public static int MAXN = 100001;  
  
    public static int[] arr = new int[MAXN];  
  
    public static int n;  
  
    public static HashMap<Integer, Integer> map = new HashMap<>();  
  
    public static void main(String[] args) throws IOException {  
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));  
        StreamTokenizer in = new StreamTokenizer(br);  
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));  
        while (in.nextToken() != StreamTokenizer.TT_EOF) {  
            n = (int) in.nval;  
            for (int i = 0, num; i < n; i++) {
```

```

        in.nextToken();
        num = (int) in.nval;
        arr[i] = num != 0 ? (num > 0 ? 1 : -1) : 0;
    }
    out.println(compute());
}
out.flush();
out.close();
br.close();
}

/***
 * 计算正数和负数个数相等的最长子数组长度
 *
 * 异常场景处理：
 * - 空数组：返回 0
 * - 单元素数组：返回 0（不可能有相同数量的正数和负数）
 * - 全正数或全负数数组：返回 0
 *
 * 边界条件：
 * - 数组长度为 0 或 1
 * - 数组元素全为正数或全为负数
 * - 数组元素交替出现
 */
public static int compute() {
    // 清空哈希表
    map.clear();

    // 初始化：前缀和为 0 在位置-1 出现（便于计算长度）
    // 这样当 sum=0 时，长度计算为 i - (-1) = i+1
    map.put(0, -1);

    int ans = 0; // 最长子数组长度
    int sum = 0; // 当前前缀和（正数看作 1， 负数看作-1， 0 看作 0）

    // 遍历数组
    for (int i = 0; i < n; i++) {
        // 更新前缀和：正数看作 1， 负数看作-1， 0 看作 0
        sum += arr[i];

        // 调试打印：显示中间过程
        // System.out.println("位置 " + i + ": 值 = " + arr[i] + ", 前缀和 = " + sum);
    }
}

```

```

        // 如果当前前缀和之前出现过，更新最大长度
        if (map.containsKey(sum)) {
            int length = i - map.get(sum);
            ans = Math.max(ans, length);
            // 调试打印：找到符合条件的子数组
            // System.out.println("找到子数组：位置 " + (map.get(sum) + 1) + " 到 " + i + "，
长度 = " + length);
        } else {
            // 记录当前前缀和第一次出现的位置
            map.put(sum, i);
            // 调试打印：记录新前缀和
            // System.out.println("记录新前缀和：" + sum + " -> " + i);
        }
    }

    return ans;
}

}

```

=====

文件：Code04\_SubarraySumsDivisibleByK.cpp

```

/*
 * 和可被 K 整除的子数组 (Subarray Sums Divisible by K)
 *
 * 题目描述：
 * 给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。
 *
 * 示例：
 * 输入：A = [4, 5, 0, -2, -3, 1]，K = 5
 * 输出：7
 * 解释：有 7 个子数组满足其元素之和可被 K = 5 整除。
 *
 * 输入：A = [5]，K = 9
 * 输出：0
 *
 * 提示：
 * 1 <= A.length <= 3 * 10^4
 * -10^4 <= A[i] <= 10^4
 * 2 <= K <= 10^4
 *
```

- \* 题目链接: <https://leetcode.com/problems/subarray-sums-divisible-by-k/>
- \*
- \* 解题思路:
  - \* 1. 利用前缀和的性质: 如果两个前缀和除以 K 的余数相同, 那么这两个位置之间的子数组和可被 K 整除
  - \* 2. 使用前缀和 + 哈希表的方法
  - \* 3. 遍历数组, 计算前缀和并对 K 取余
  - \* 4. 使用哈希表记录每个余数出现的次数
  - \* 5. 对于相同的余数, 任意两个位置之间的子数组都满足条件
- \*
- \* 时间复杂度:  $O(n)$  - 需要遍历数组一次
- \* 空间复杂度:  $O(\min(n, K))$  - 哈希表最多存储 K 个不同的余数或 n 个前缀和
- \*
- \* 工程化考量:
  - \* 1. 边界条件处理: 空数组、 $K=0$ 、 $K=1$  等特殊情况
  - \* 2. 负数取模处理: C++中负数取模结果为负, 需要转换为正数
  - \* 3. 哈希表选择: `unordered_map` 提供  $O(1)$  的平均查找时间
  - \* 4. 整数溢出: 使用 `long long` 避免大数溢出
- \*
- \* 最优解分析:
  - \* 这是最优解, 因为必须遍历所有元素才能统计所有子数组。
  - \* 哈希表方法将时间复杂度从  $O(n^2)$  优化到  $O(n)$ 。
- \*
- \* 算法核心:
  - \* 设 `prefix[i]` 为前  $i$  个元素的和, 则子数组  $[i, j]$  的和为 `prefix[j] - prefix[i-1]`。
  - \* 当  $(prefix[j] - prefix[i-1]) \% K = 0$  时, 即 `prefix[j] \% K = prefix[i-1] \% K`。
  - \* 因此统计相同余数的前缀和对数即可。
- \*/

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

using namespace std;

class Solution {
public:
    /**
     * 计算和可被 K 整除的子数组数目
     *
     * @param A 输入数组
     * @param K 除数
     * @return 和可被 K 整除的子数组数目
     */
}
```

```

*
* 异常场景处理:
* - 空数组: 返回 0
* - K=0: 返回 0 (除数不能为 0)
* - K=1: 所有子数组都满足条件
*/
int subarraysDivByK(vector<int>& A, int K) {
    // 边界情况处理
    if (A.empty() || K == 0) {
        return 0;
    }

    // 如果 K=1, 所有子数组都满足条件
    if (K == 1) {
        int n = A.size();
        return n * (n + 1) / 2;
    }

    // 使用 unordered_map 记录每个余数出现的次数
    unordered_map<int, int> map;
    // 初始化: 余数为 0 出现 1 次 (表示空前缀)
    map[0] = 1;

    int count = 0;           // 结果计数
    long long prefixSum = 0; // 当前前缀和, 使用 long long 避免溢出

    // 遍历数组
    for (int num : A) {
        // 更新前缀和
        prefixSum += num;

        // 计算前缀和对 K 的余数 (处理负数情况)
        // C++中负数取模结果为负, 需要转换为[0, K-1]范围内的正数
        int remainder = prefixSum % K;
        if (remainder < 0) {
            remainder += K;
        }

        // 如果该余数之前出现过, 说明存在满足条件的子数组
        if (map.find(remainder) != map.end()) {
            count += map[remainder];
        }
    }
}

```

```
// 更新该余数的出现次数
map[remainder]++;
}

return count;
}

};

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

    // 测试用例 1: 经典情况
    vector<int> A1 = {4, 5, 0, -2, -3, 1};
    int K1 = 5;
    int result1 = solution.subarraysDivByK(A1, K1);
    cout << "测试用例 1 [4,5,0,-2,-3,1] K=5: " << result1 << " (预期: 7)" << endl;

    // 测试用例 2: 单个元素
    vector<int> A2 = {5};
    int K2 = 9;
    int result2 = solution.subarraysDivByK(A2, K2);
    cout << "测试用例 2 [5] K=9: " << result2 << " (预期: 0)" << endl;

    // 测试用例 3: K=1 的情况
    vector<int> A3 = {1, 2, 3};
    int K3 = 1;
    int result3 = solution.subarraysDivByK(A3, K3);
    cout << "测试用例 3 [1,2,3] K=1: " << result3 << " (预期: 6)" << endl;

    // 测试用例 4: 空数组
    vector<int> A4 = {};
    int K4 = 5;
    int result4 = solution.subarraysDivByK(A4, K4);
    cout << "测试用例 4 [] K=5: " << result4 << " (预期: 0)" << endl;

    // 测试用例 5: K=0 的情况
    vector<int> A5 = {1, 2, 3};
    int K5 = 0;
    int result5 = solution.subarraysDivByK(A5, K5);
    cout << "测试用例 5 [1,2,3] K=0: " << result5 << " (预期: 0)" << endl;
```

```

// 测试用例 6: 全 0 数组
vector<int> A6 = {0, 0, 0};
int K6 = 2;
int result6 = solution.subarraysDivByK(A6, K6);
cout << "测试用例 6 [0,0,0] K=2: " << result6 << " (预期: 6)" << endl;

// 测试用例 7: 包含负数
vector<int> A7 = {-1, 2, 9};
int K7 = 2;
int result7 = solution.subarraysDivByK(A7, K7);
cout << "测试用例 7 [-1,2,9] K=2: " << result7 << " (预期: 2)" << endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
    cout << "\n==== 性能测试 ===" << endl;
    Solution solution;
    int size = 30000; // 3 万元素 (题目上限)
    int K = 1000; // K 的上限
    vector<int> largeArray(size);

    // 初始化大数组
    for (int i = 0; i < size; i++) {
        largeArray[i] = (i % 200) - 100; // 包含正负数
    }

    auto startTime = chrono::high_resolution_clock::now();
    int result = solution.subarraysDivByK(largeArray, K);
    auto endTime = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);
    cout << "处理 " << size << " 个元素, 结果: " << result << ", 耗时: " << duration.count() << "ms" << endl;
}

/***
 * 主函数
 */
int main() {
    cout << "==== 和可被 K 整除的子数组测试 ===" << endl;
}

```

```
    testSubarraysDivByK();

    // 运行性能测试（可选）
    // performanceTest();

    cout << "\n==== 测试完成 ===" << endl;
    return 0;
}
```

=====

文件: Code04\_SubarraySumsDivisibleByK.java

=====

```
package class046;

import java.util.HashMap;

/**
 * 和可被 K 整除的子数组 (Subarray Sums Divisible by K)
 *
 * 题目描述:
 * 给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。
 *
 * 示例:
 * 输入: A = [4, 5, 0, -2, -3, 1], K = 5
 * 输出: 7
 * 解释: 有 7 个子数组满足其元素之和可被 K = 5 整除。
 *
 * 输入: A = [5], K = 9
 * 输出: 0
 *
 * 提示:
 * 1 <= A.length <= 3 * 10^4
 * -10^4 <= A[i] <= 10^4
 * 2 <= K <= 10^4
 *
 * 题目链接: https://leetcode.com/problems/subarray-sums-divisible-by-k/
 *
 * 解题思路:
 * 1. 利用前缀和的性质: 如果两个前缀和除以 K 的余数相同, 那么这两个位置之间的子数组和可被 K 整除
 * 2. 使用前缀和 + 哈希表的方法
 * 3. 遍历数组, 计算前缀和并对 K 取余
 * 4. 使用哈希表记录每个余数出现的次数
```

- \* 5. 对于相同的余数，任意两个位置之间的子数组都满足条件
- \*
- \* 时间复杂度： $O(n)$  – 需要遍历数组一次
- \* 空间复杂度： $O(\min(n, K))$  – 哈希表最多存储  $K$  个不同的余数或  $n$  个前缀和
- \*
- \* 工程化考量：
  - \* 1. 边界条件处理：空数组、 $K=0$ 、 $K=1$  等特殊情况
  - \* 2. 负数取模处理：Java 中负数取模结果为负，需要转换为正数
  - \* 3. 哈希表选择：HashMap 提供  $O(1)$  的平均查找时间
  - \* 4. 整数溢出：使用 long 避免大数溢出
- \*
- \* 最优解分析：
  - \* 这是最优解，因为必须遍历所有元素才能统计所有子数组。
  - \* 哈希表方法将时间复杂度从  $O(n^2)$  优化到  $O(n)$ 。
- \*
- \* 算法核心：
  - \* 设  $\text{prefix}[i]$  为前  $i$  个元素的和，则子数组  $[i, j]$  的和为  $\text{prefix}[j] - \text{prefix}[i-1]$ 。
  - \* 当  $(\text{prefix}[j] - \text{prefix}[i-1]) \% K = 0$  时，即  $\text{prefix}[j] \% K = \text{prefix}[i-1] \% K$ 。
  - \* 因此统计相同余数的前缀和对数即可。
- \*
- \* 算法调试技巧：
  - \* 1. 打印中间过程：可以在循环中打印每个位置的前缀和和余数
  - \* 2. 边界测试：测试  $K=0$ 、 $K=1$ 、负数等情况
  - \* 3. 性能测试：测试大规模数据下的性能表现
- \*
- \* 语言特性差异：
  - \* Java 的负数取模需要特殊处理，而 Python 的负数取模结果为正。
  - \* 与 C++ 相比，Java 有自动内存管理，无需手动释放哈希表内存。
- \*/

```
public class Code04_SubarraySumsDivisibleByK {
```

```
    /**
     * 计算和可被 K 整除的子数组数目
     *
     * @param A 输入数组
     * @param K 除数
     * @return 和可被 K 整除的子数组数目
     *
     * 异常场景处理：
     * - 空数组：返回 0
     * - K=0：返回 0 (除数不能为 0)
     * - K=1：所有子数组都满足条件
     * - 数组元素包含负数：算法本身支持
    }
```

```

*
* 边界条件:
* - K=0 的情况
* - K=1 的情况 (所有子数组都满足)
* - 数组元素全为 0
*/
public static int subarraysDivByK(int[] A, int K) {
    // 边界情况处理
    if (A == null || A.length == 0 || K == 0) {
        return 0;
    }

    // 如果 K=1, 所有子数组都满足条件
    if (K == 1) {
        int n = A.length;
        return n * (n + 1) / 2;
    }

    // 使用 HashMap 记录每个余数出现的次数
    HashMap<Integer, Integer> map = new HashMap<>();
    // 初始化: 余数为 0 出现 1 次 (表示空前缀)
    map.put(0, 1);

    int count = 0;           // 结果计数
    long prefixSum = 0;      // 当前前缀和, 使用 long 避免溢出

    // 遍历数组
    for (int num : A) {
        // 更新前缀和
        prefixSum += num;

        // 计算前缀和对 K 的余数 (处理负数情况)
        // Java 中负数取模结果为负, 需要转换为[0, K-1]范围内的正数
        int remainder = (int) (prefixSum % K);
        if (remainder < 0) {
            remainder += K;
        }

        // 调试打印: 显示中间过程
        // System.out.println("前缀和 = " + prefixSum + ", 余数 = " + remainder);

        // 如果该余数之前出现过, 说明存在满足条件的子数组
        if (map.containsKey(remainder)) {

```

```
        count += map.get(remainder);
        // 调试打印: 找到符合条件的子数组
        // System.out.println("找到子数组, 当前计数: " + count);
    }

    // 更新该余数的出现次数
    map.put(remainder, map.getOrDefault(remainder, 0) + 1);

    // 调试打印: 哈希表状态
    // System.out.println("哈希表更新: " + remainder + " -> " + map.get(remainder));
}

return count;
}

/**
 * 单元测试方法
 */
public static void testSubarraysDivByK() {
    System.out.println("== 和可被 K 整除的子数组单元测试 ==");

    // 测试用例 1: 经典情况
    int[] A1 = {4, 5, 0, -2, -3, 1};
    int K1 = 5;
    int result1 = subarraysDivByK(A1, K1);
    System.out.println("测试用例 1 [4,5,0,-2,-3,1] K=5: " + result1 + " (预期: 7)");

    // 测试用例 2: 单个元素
    int[] A2 = {5};
    int K2 = 9;
    int result2 = subarraysDivByK(A2, K2);
    System.out.println("测试用例 2 [5] K=9: " + result2 + " (预期: 0)");

    // 测试用例 3: K=1 的情况
    int[] A3 = {1, 2, 3};
    int K3 = 1;
    int result3 = subarraysDivByK(A3, K3);
    System.out.println("测试用例 3 [1,2,3] K=1: " + result3 + " (预期: 6)");

    // 测试用例 4: 空数组
    int[] A4 = {};
    int K4 = 5;
    int result4 = subarraysDivByK(A4, K4);
```

```

System.out.println("测试用例 4 [] K=5: " + result4 + " (预期: 0);

// 测试用例 5: K=0 的情况
int[] A5 = {1, 2, 3};
int K5 = 0;
int result5 = subarraysDivByK(A5, K5);
System.out.println("测试用例 5 [1,2,3] K=0: " + result5 + " (预期: 0)");

// 测试用例 6: 全 0 数组
int[] A6 = {0, 0, 0};
int K6 = 2;
int result6 = subarraysDivByK(A6, K6);
System.out.println("测试用例 6 [0,0,0] K=2: " + result6 + " (预期: 6)");

// 测试用例 7: 包含负数
int[] A7 = {-1, 2, 9};
int K7 = 2;
int result7 = subarraysDivByK(A7, K7);
System.out.println("测试用例 7 [-1,2,9] K=2: " + result7 + " (预期: 2)");
}

/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n==== 性能测试 ====");
    int size = 30000; // 3 万元素 (题目上限)
    int[] largeArray = new int[size];
    int K = 1000; // K 的上限

    // 初始化大数组
    for (int i = 0; i < size; i++) {
        largeArray[i] = (i % 200) - 100; // 包含正负数
    }

    long startTime = System.currentTimeMillis();
    int result = subarraysDivByK(largeArray, K);
    long endTime = System.currentTimeMillis();

    System.out.println("处理 " + size + " 个元素, 结果: " + result + ", 耗时: " + (endTime - startTime) + "ms");
}

```

```
/**  
 * 主函数 - 测试入口  
 */  
public static void main(String[] args) {  
    // 运行单元测试  
    testSubarraysDivByK();  
  
    // 运行性能测试（可选）  
    // performanceTest();  
  
    System.out.println("\n==== 测试完成 ===");  
}  
}
```

=====

文件: Code04\_SubarraySumsDivisibleByK.py

=====

"""

和可被 K 整除的子数组 (Subarray Sums Divisible by K)

题目描述:

给定一个整数数组 A，返回其中元素之和可被 K 整除的（连续、非空）子数组的数目。

示例:

输入: A = [4, 5, 0, -2, -3, 1], K = 5

输出: 7

解释: 有 7 个子数组满足其元素之和可被 K = 5 整除。

输入: A = [5], K = 9

输出: 0

提示:

$1 \leq A.length \leq 3 * 10^4$

$-10^4 \leq A[i] \leq 10^4$

$2 \leq K \leq 10^4$

题目链接: <https://leetcode.com/problems/subarray-sums-divisible-by-k/>

解题思路:

1. 利用前缀和的性质: 如果两个前缀和除以 K 的余数相同, 那么这两个位置之间的子数组和可被 K 整除
2. 使用前缀和 + 哈希表的方法
3. 遍历数组, 计算前缀和并对 K 取余

4. 使用哈希表记录每个余数出现的次数
5. 对于相同的余数，任意两个位置之间的子数组都满足条件

时间复杂度： $O(n)$  – 需要遍历数组一次

空间复杂度： $O(\min(n, K))$  – 哈希表最多存储  $K$  个不同的余数或  $n$  个前缀和

工程化考量：

1. 负数余数处理：Python 的模运算与数学定义不同，需要特殊处理
2. 边界条件：空数组、 $K=0$  等特殊情况
3. 性能优化：一次遍历完成所有计算
4. 哈希表初始化：空前缀和的余数为 0 出现 1 次

最优解分析：

这是最优解，时间复杂度  $O(n)$ ，空间复杂度  $O(\min(n, K))$ 。

必须遍历所有元素才能统计所有满足条件的子数组。

数学原理：

设  $\text{prefix}[i]$  为前  $i$  个元素的和，则子数组  $[i, j]$  的和为  $\text{prefix}[j] - \text{prefix}[i-1]$

要求  $(\text{prefix}[j] - \text{prefix}[i-1]) \% K == 0 \rightarrow \text{prefix}[j] \% K == \text{prefix}[i-1] \% K$

算法调试技巧：

1. 打印中间过程：显示每个位置的前缀和和余数
2. 边界测试：测试  $K=0$ 、负数数组等特殊情况
3. 性能测试：测试大规模数据下的性能表现

语言特性差异：

Python 的模运算对于负数结果与数学定义不同，需要特殊处理。

Java/C++ 的模运算对于负数结果与数学定义一致，不需要额外处理。

"""

```
class Solution:  
    def subarraysDivByK(self, A, K):  
        """  
        计算和可被 K 整除的子数组数目  
        """
```

Args:

A (List[int]): 输入数组  
K (int): 除数

Returns:

int: 和可被 K 整除的子数组数目

异常场景处理：

- 空数组：返回 0
- K=0：根据题目约束  $K \geq 2$ ，无需处理
- 负数处理：Python 模运算需要特殊处理负数

边界条件：

- 数组为空
- $K=1$ （但题目约束  $K \geq 2$ ）
- 数组元素全为 0
- 数组元素全为负数

"""

# 边界情况处理

```
if not A or K == 0:
    return 0
```

# 字典记录每个余数出现的次数

# 初始化：余数为 0 出现 1 次（表示空前缀）

# 数学原理： $\text{prefix}[j] \% K == \text{prefix}[i-1] \% K$

```
remainder_count = {0: 1}
```

count = 0 # 结果计数

sum\_val = 0 # 当前前缀和

# 遍历数组，时间复杂度  $O(n)$

```
for i, num in enumerate(A):
```

# 更新前缀和

```
sum_val += num
```

# 计算前缀和对 K 的余数

# Python 的模运算：对于负数，结果与数学定义不同

# 例如： $-1 \% 5 = 4$  (Python)，数学上应该是 -1

```
remainder = sum_val % K
```

# 处理负数余数：如果余数为负，加上 K 使其为正

# 这是 Python 特有的处理，确保余数在  $[0, K-1]$  范围内

```
if remainder < 0:
```

```
    remainder += K
```

# 调试打印：显示中间过程

```
# print(f"位置 {i}: 数字 = {num}, 前缀和 = {sum_val}, 余数 = {remainder}")
```

# 如果该余数之前出现过，说明存在满足条件的子数组

```
if remainder in remainder_count:
```

# 当前余数出现的次数就是新发现的子数组数量

```
        count += remainder_count[remainder]
        # 调试打印: 找到新子数组
        # print(f"找到 {remainder_count[remainder]} 个新子数组, 总计数 = {count}")

        # 更新该余数的出现次数
        remainder_count[remainder] = remainder_count.get(remainder, 0) + 1

    return count

def test_subarrays_divisible_by_k():
    """单元测试函数"""
    print("== 和可被 K 整除的子数组单元测试 ==")
    solution = Solution()

    # 测试用例 1: 题目示例
    A1 = [4, 5, 0, -2, -3, 1]
    K1 = 5
    result1 = solution.subarraysDivByK(A1, K1)
    print(f"测试用例 1 [4,5,0,-2,-3,1], K=5: {result1} (预期: 7)")

    # 测试用例 2: 单个元素
    A2 = [5]
    K2 = 9
    result2 = solution.subarraysDivByK(A2, K2)
    print(f"测试用例 2 [5], K=9: {result2} (预期: 0)")

    # 测试用例 3: 空数组
    A3 = []
    K3 = 5
    result3 = solution.subarraysDivByK(A3, K3)
    print(f"测试用例 3 [], K=5: {result3} (预期: 0)")

    # 测试用例 4: 全 0 数组
    A4 = [0, 0, 0]
    K4 = 3
    result4 = solution.subarraysDivByK(A4, K4)
    print(f"测试用例 4 [0,0,0], K=3: {result4} (预期: 6)")

    # 测试用例 5: 负数数组
    A5 = [-1, -2, -3]
    K5 = 3
    result5 = solution.subarraysDivByK(A5, K5)
```

```

print(f"测试用例 5 [-1,-2,-3], K=3: {result5} (预期: 2)")

# 测试用例 6: 边界情况
A6 = [2, 3, 7]
K6 = 1
result6 = solution.subarraysDivByK(A6, K6)
print(f"测试用例 6 [2,3,7], K=1: {result6} (预期: 6)")

def performance_test():
    """性能测试函数"""
    print("\n==== 性能测试 ===")
    solution = Solution()
    size = 30000 # 3 万元素
    import random
    large_array = [random.randint(-10000, 10000) for _ in range(size)]
    K = 100

    import time
    start_time = time.time()
    result = solution.subarraysDivByK(large_array, K)
    end_time = time.time()

    print(f"处理 {size} 个元素, 子数组数量: {result}, 耗时: {end_time - start_time:.4f} 秒")

if __name__ == "__main__":
    # 运行单元测试
    test_subarrays_divisible_by_k()

    # 运行性能测试 (可选)
    # performance_test()

    print("\n==== 测试完成 ===")

```

=====

文件: Code05\_LongestWellPerformingInterval.java

=====

```

package class046;

import java.util.HashMap;

// 表现良好的最长时间段
// 给你一份工作时间表 hours, 上面记录着某一位员工每天的工作小时数

```

```

// 我们认为当员工一天中的工作小时数大于 8 小时的时候，那么这一天就是 劳累的一天
// 所谓 表现良好的时间段，意味在这段时间内，「劳累的天数」是严格 大于 不劳累的天数
// 请你返回 表现良好时间段 的最大长度
// 测试链接 : https://leetcode.cn/problems/longest-well-performing-interval/
public class Code05_LongestWellPerformingInterval {

    public static int longestWPI(int[] hours) {
        // 某个前缀和，最早出现的位置
        HashMap<Integer, Integer> map = new HashMap<>();
        // 0 这个前缀和，最早出现在-1，一个数也没有的时候
        map.put(0, -1);
        int ans = 0;
        for (int i = 0, sum = 0; i < hours.length; i++) {
            sum += hours[i] > 8 ? 1 : -1;
            if (sum > 0) {
                ans = i + 1;
            } else {
                // sum <= 0
                if (map.containsKey(sum - 1)) {
                    ans = Math.max(ans, i - map.get(sum - 1));
                }
            }
            if (!map.containsKey(sum)) {
                map.put(sum, i);
            }
        }
        return ans;
    }
}

```

}

=====

文件: Code05\_ProductOfArrayExceptSelf.cpp

=====

```

/***
 * 除自身以外数组的乘积 (Product of Array Except Self)
 *
 * 题目描述:
 * 给你一个长度为 n 的整数数组 nums，其中 n > 1，返回输出数组 output，
 * 其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。
 *
 * 示例:

```

- \* 输入: [1, 2, 3, 4]
- \* 输出: [24, 12, 8, 6]
- \*
- \* 提示:
- \* 题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。
- \*
- \* 说明:
- \* 请不要使用除法，且在  $O(n)$  时间复杂度内完成此题。
- \*
- \* 进阶:
- \* 你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）
- \*
- \* 题目链接: <https://leetcode.com/problems/product-of-array-except-self/>
- \*
- \* 解题思路:
- \* 1. 使用两个数组分别存储左侧所有元素的乘积和右侧所有元素的乘积
- \* 2. 对于位置  $i$ ，结果为左侧乘积乘以右侧乘积
- \* 3. 进阶：使用输出数组存储左侧乘积，然后从右向左遍历计算右侧乘积并直接更新结果
- \*
- \* 时间复杂度:  $O(n)$  – 需要遍历数组两次
- \* 空间复杂度:  $O(1)$  – 不考虑输出数组，只使用常数额外空间
- \*
- \* 工程化考量:
- \* 1. 边界条件处理: 空数组、单元素数组
- \* 2. 空间优化: 使用输出数组存储中间结果，避免额外空间
- \* 3. 整数溢出: 虽然题目保证在 32 位整数范围内，但实际工程中需要考虑
- \* 4. 代码可读性: 清晰的变量命名和注释
- \*
- \* 最优解分析:
- \* 这是最优解，因为必须遍历所有元素才能计算乘积。
- \* 空间复杂度  $O(1)$  也是最优的（不考虑输出数组）。
- \*
- \* 算法核心:
- \* 对于每个位置  $i$ ，结果 = 左侧所有元素的乘积  $\times$  右侧所有元素的乘积
- \* 通过左右两次遍历，分别计算左侧乘积和右侧乘积。

\*/

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

using namespace std;
```

```
class Solution {
public:
    /**
     * 计算除自身以外数组的乘积
     *
     * @param nums 输入数组
     * @return 除自身以外数组的乘积
     *
     * 异常场景处理：
     * - 空数组：直接返回原数组
     * - 单元素数组：直接返回原数组
     * - 包含 0 的数组：需要特殊处理（但算法本身支持）
     */
    vector<int> productExceptSelf(vector<int>& nums) {
        // 边界情况处理：空数组或单元素数组直接返回
        if (nums.empty() || nums.size() <= 1) {
            return nums;
        }

        int n = nums.size();
        vector<int> result(n);

        // 第一遍遍历：从左到右，计算每个位置左侧所有元素的乘积
        result[0] = 1; // 第一个元素左侧没有元素，乘积为 1
        for (int i = 1; i < n; i++) {
            result[i] = result[i - 1] * nums[i - 1];
        }

        // 第二遍遍历：从右到左，计算每个位置右侧所有元素的乘积，并与左侧乘积相乘
        int rightProduct = 1;
        for (int i = n - 1; i >= 0; i--) {
            // 当前位置的结果 = 左侧乘积 × 右侧乘积
            result[i] *= rightProduct;
            // 更新右侧乘积，为下一个位置（左边）准备
            rightProduct *= nums[i];
        }

        return result;
    }
};

/**
```

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

    // 测试用例 1: 正常情况
    vector<int> nums1 = {1, 2, 3, 4};
    vector<int> result1 = solution.productExceptSelf(nums1);
    cout << "测试用例 1 [1,2,3,4]: ";
    for (int num : result1) {
        cout << num << " ";
    }
    cout << " (预期: 24 12 8 6)" << endl;

    // 测试用例 2: 包含负数
    vector<int> nums2 = {2, 3, 4, 5};
    vector<int> result2 = solution.productExceptSelf(nums2);
    cout << "测试用例 2 [2,3,4,5]: ";
    for (int num : result2) {
        cout << num << " ";
    }
    cout << " (预期: 60 40 30 24)" << endl;

    // 测试用例 3: 包含 0
    vector<int> nums3 = {1, 0, 3, 4};
    vector<int> result3 = solution.productExceptSelf(nums3);
    cout << "测试用例 3 [1,0,3,4]: ";
    for (int num : result3) {
        cout << num << " ";
    }
    cout << " (预期: 0 12 0 0)" << endl;

    // 测试用例 4: 包含负数和 0
    vector<int> nums4 = {-1, 2, 0, -3};
    vector<int> result4 = solution.productExceptSelf(nums4);
    cout << "测试用例 4 [-1,2,0,-3]: ";
    for (int num : result4) {
        cout << num << " ";
    }
    cout << " (预期: 0 0 6 0)" << endl;

    // 测试用例 5: 空数组
    vector<int> nums5 = {};
```

```

vector<int> result5 = solution.productExceptSelf(nums5);
cout << "测试用例 5 []: ";
for (int num : result5) {
    cout << num << " ";
}
cout << "(预期: 空数组)" << endl;

// 测试用例 6: 单元素数组
vector<int> nums6 = {5};
vector<int> result6 = solution.productExceptSelf(nums6);
cout << "测试用例 6 [5]: ";
for (int num : result6) {
    cout << num << " ";
}
cout << "(预期: 5)" << endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
    cout << "\n==== 性能测试 ====" << endl;
    Solution solution;
    int size = 1000000; // 100 万元素
    vector<int> largeArray(size);

    // 初始化大数组 (避免溢出)
    for (int i = 0; i < size; i++) {
        largeArray[i] = (i % 10) + 1; // 数值范围 1-10
    }

    auto startTime = chrono::high_resolution_clock::now();
    vector<int> result = solution.productExceptSelf(largeArray);
    auto endTime = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);
    cout << "处理 " << size << " 个元素耗时: " << duration.count() << "ms" << endl;
}

/***
 * 主函数
 */
int main() {

```

```
cout << "==== 除自身以外数组的乘积测试 ===" << endl;
testProductExceptSelf();

// 运行性能测试（可选）
// performanceTest();

cout << "\n==== 测试完成 ===" << endl;
return 0;
}
```

---

文件: Code05\_ProductOfArrayExceptSelf.java

---

```
package class046;

/**
 * 除自身以外数组的乘积 (Product of Array Except Self)
 *
 * 题目描述:
 * 给你一个长度为 n 的整数数组 nums，其中 n > 1，返回输出数组 output，其中 output[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积。
 *
 * 示例:
 * 输入: [1, 2, 3, 4]
 * 输出: [24, 12, 8, 6]
 *
 * 提示:
 * 题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。
 *
 * 说明:
 * 请不要使用除法，且在 O(n) 时间复杂度内完成此题。
 *
 * 进阶:
 * 你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）
 *
 * 题目链接: https://leetcode.com/problems/product-of-array-except-self/
 *
 * 解题思路:
 * 1. 使用两个数组分别存储左侧所有元素的乘积和右侧所有元素的乘积
 * 2. 对于位置 i，结果为左侧乘积乘以右侧乘积
 * 3. 进阶：使用输出数组存储左侧乘积，然后从右向左遍历计算右侧乘积并直接更新结果
```

```
*  
* 时间复杂度: O(n) - 需要遍历数组两次  
* 空间复杂度: O(1) - 不考虑输出数组, 只使用常数额外空间  
*  
* 工程化考量:  
* 1. 边界条件处理: 空数组、单元素数组  
* 2. 空间优化: 使用输出数组存储中间结果, 避免额外空间  
* 3. 整数溢出: 虽然题目保证在 32 位整数范围内, 但实际工程中需要考虑  
* 4. 代码可读性: 清晰的变量命名和注释  
*  
* 最优解分析:  
* 这是最优解, 因为必须遍历所有元素才能计算乘积。  
* 空间复杂度 O(1) 也是最优的 (不考虑输出数组)。  
*  
* 算法核心:  
* 对于每个位置 i, 结果 = 左侧所有元素的乘积 × 右侧所有元素的乘积  
* 通过左右两次遍历, 分别计算左侧乘积和右侧乘积。  
*  
* 算法调试技巧:  
* 1. 打印中间过程: 可以在循环中打印每个位置的左侧乘积和右侧乘积  
* 2. 边界测试: 测试包含 0、负数、大数等情况  
* 3. 性能测试: 测试大规模数据下的性能表现  
*  
* 语言特性差异:  
* Java 中数组是对象, 可以直接修改输出数组。  
* 与 C++ 相比, Java 有自动内存管理, 无需手动释放数组内存。  
* 与 Python 相比, Java 是静态类型语言, 需要显式声明数组类型。  
*/  
  
public class Code05_ProductOfArrayExceptSelf {  
  
    /**  
     * 计算除自身以外数组的乘积  
     *  
     * @param nums 输入数组  
     * @return 除自身以外数组的乘积  
     *  
     * 异常场景处理:  
     * - 空数组: 直接返回原数组  
     * - 单元素数组: 直接返回原数组  
     * - 包含 0 的数组: 需要特殊处理 (但算法本身支持)  
     * - 包含负数的数组: 算法本身支持  
     *  
     * 边界条件:  
    */
```

```

* - 数组长度为 0 或 1
* - 数组元素包含 0
* - 数组元素包含负数
* - 数组元素包含大数（可能溢出）
*/
public static int[] productExceptSelf(int[] nums) {
    // 边界情况处理：空数组或单元素数组直接返回
    if (nums == null || nums.length <= 1) {
        return nums;
    }

    int n = nums.length;
    int[] result = new int[n];

    // 第一遍遍历：从左到右，计算每个位置左侧所有元素的乘积
    // result[i] = nums[0] * nums[1] * ... * nums[i-1]
    result[0] = 1; // 第一个元素左侧没有元素，乘积为 1
    for (int i = 1; i < n; i++) {
        result[i] = result[i - 1] * nums[i - 1];
        // 调试打印：显示左侧乘积
        // System.out.println("位置 " + i + " 左侧乘积: " + result[i]);
    }

    // 第二遍遍历：从右到左，计算每个位置右侧所有元素的乘积，并与左侧乘积相乘
    // 同时计算右侧乘积：rightProduct = nums[i+1] * nums[i+2] * ... * nums[n-1]
    int rightProduct = 1;
    for (int i = n - 1; i >= 0; i--) {
        // 当前位置的结果 = 左侧乘积 × 右侧乘积
        result[i] *= rightProduct;
        // 更新右侧乘积，为下一个位置（左边）准备
        rightProduct *= nums[i];
        // 调试打印：显示最终结果和右侧乘积
        // System.out.println("位置 " + i + " 结果: " + result[i] + ", 右侧乘积: " +
rightProduct);
    }

    return result;
}

/**
 * 单元测试方法
 */
public static void testProductExceptSelf() {

```

```
System.out.println("==> 除自身以外数组的乘积单元测试 ==>");
```

```
// 测试用例 1: 正常情况
```

```
int[] nums1 = {1, 2, 3, 4};  
int[] result1 = productExceptSelf(nums1);  
System.out.print("测试用例 1 [1,2,3,4]: ");  
for (int num : result1) {  
    System.out.print(num + " ");  
}  
System.out.println(" (预期: 24 12 8 6)");
```

```
// 测试用例 2: 包含负数
```

```
int[] nums2 = {2, 3, 4, 5};  
int[] result2 = productExceptSelf(nums2);  
System.out.print("测试用例 2 [2,3,4,5]: ");  
for (int num : result2) {  
    System.out.print(num + " ");  
}  
System.out.println(" (预期: 60 40 30 24)");
```

```
// 测试用例 3: 包含 0
```

```
int[] nums3 = {1, 0, 3, 4};  
int[] result3 = productExceptSelf(nums3);  
System.out.print("测试用例 3 [1,0,3,4]: ");  
for (int num : result3) {  
    System.out.print(num + " ");  
}  
System.out.println(" (预期: 0 12 0 0)");
```

```
// 测试用例 4: 包含负数和 0
```

```
int[] nums4 = {-1, 2, 0, -3};  
int[] result4 = productExceptSelf(nums4);  
System.out.print("测试用例 4 [-1,2,0,-3]: ");  
for (int num : result4) {  
    System.out.print(num + " ");  
}  
System.out.println(" (预期: 0 0 6 0)");
```

```
// 测试用例 5: 空数组
```

```
int[] nums5 = {};  
int[] result5 = productExceptSelf(nums5);  
System.out.print("测试用例 5 []: ");  
for (int num : result5) {
```

```

        System.out.print(num + " ");
    }

    System.out.println(" (预期: 空数组)");

    // 测试用例 6: 单元素数组
    int[] nums6 = {5};
    int[] result6 = productExceptSelf(nums6);
    System.out.print("测试用例 6 [5]: ");
    for (int num : result6) {
        System.out.print(num + " ");
    }
    System.out.println(" (预期: 5)");
}

/**
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println("\n== 性能测试 ==");
    int size = 1000000; // 100 万元素
    int[] largeArray = new int[size];

    // 初始化大数组 (避免溢出)
    for (int i = 0; i < size; i++) {
        largeArray[i] = (i % 10) + 1; // 数值范围 1-10
    }

    long startTime = System.currentTimeMillis();
    productExceptSelf(largeArray);
    long endTime = System.currentTimeMillis();

    System.out.println("处理 " + size + " 个元素耗时: " + (endTime - startTime) + "ms");
}

/**
 * 主函数 - 测试入口
 */
public static void main(String[] args) {
    // 运行单元测试
    testProductExceptSelf();

    // 运行性能测试 (可选)
    // performanceTest();
}

```

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

---

文件: Code05\_ProductOfArrayExceptSelf.py

---

"""

除自身以外数组的乘积 (Product of Array Except Self)

题目描述:

给你一个长度为  $n$  的整数数组  $\text{nums}$ , 其中  $n > 1$ , 返回输出数组  $\text{output}$ , 其中  $\text{output}[i]$  等于  $\text{nums}$  中除  $\text{nums}[i]$  之外其余各元素的乘积。

示例:

输入: [1, 2, 3, 4]

输出: [24, 12, 8, 6]

提示:

题目数据保证数组之中任意元素的全部前缀元素和后缀（甚至是整个数组）的乘积都在 32 位整数范围内。

说明:

请不要使用除法，且在  $O(n)$  时间复杂度内完成此题。

进阶:

你可以在常数空间复杂度内完成这个题目吗？（出于对空间复杂度分析的目的，输出数组不被视为额外空间。）

题目链接: <https://leetcode.com/problems/product-of-array-except-self/>

解题思路:

1. 使用两个数组分别存储左侧所有元素的乘积和右侧所有元素的乘积
2. 对于位置  $i$ , 结果为左侧乘积乘以右侧乘积
3. 进阶: 使用输出数组存储左侧乘积, 然后从右向左遍历计算右侧乘积并直接更新结果

时间复杂度:  $O(n)$  – 需要遍历数组两次

空间复杂度:  $O(1)$  – 不考虑输出数组, 只使用常数额外空间

工程化考量:

1. 边界条件处理: 空数组、单元素数组
2. 零元素处理: 当数组中有 0 时, 乘积结果的特殊处理
3. 性能优化: 两次遍历完成计算, 避免使用除法

#### 4. 空间优化：使用输出数组存储中间结果

最优解分析：

这是最优解，时间复杂度  $O(n)$ ，空间复杂度  $O(1)$ （不算输出数组）。

必须遍历所有元素才能计算每个位置的乘积。

数学原理：

对于位置  $i$ ，结果 =  $(\text{nums}[0] * \dots * \text{nums}[i-1]) * (\text{nums}[i+1] * \dots * \text{nums}[n-1])$

算法调试技巧：

1. 打印中间过程：显示前缀乘积和后缀乘积的计算过程
2. 边界测试：测试包含 0 的数组、负数数组等特殊情况
3. 性能测试：测试大规模数据下的性能表现

语言特性差异：

Python 的整数范围较大，无需担心溢出问题。

Java/C++需要考虑整数溢出，可能需要使用 long 类型。

"""

```
class Solution:
```

```
    def productExceptSelf(self, nums):
```

```
        """
```

```
        计算除自身以外数组的乘积
```

Args:

    nums (List[int]): 输入数组

Returns:

    List[int]: 除自身以外数组的乘积

异常场景处理：

- 空数组：返回空数组
- 单元素数组：返回 [1]
- 包含 0 的数组：需要特殊处理 0 的影响

边界条件：

- 数组长度为 0 或 1
- 数组包含 0 元素
- 数组包含负数元素

```
    """
```

```
# 边界情况处理
```

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

```

n = len(nums)
result = [0] * n

# 第一遍遍历：计算每个位置左侧所有元素的乘积
# result[i] 存储 nums[0] * nums[1] * ... * nums[i-1]
result[0] = 1
for i in range(1, n):
    result[i] = result[i - 1] * nums[i - 1]
    # 调试打印：左侧乘积计算
    # print(f"左侧计算 位置 {i}: result[{i}] = {result[i]}")

# 第二遍遍历：从右向左，计算每个位置右侧所有元素的乘积，并与左侧乘积相乘
# result[i] *= (nums[i+1] * nums[i+2] * ... * nums[n-1])
right_product = 1
for i in range(n - 1, -1, -1):
    result[i] *= right_product
    right_product *= nums[i]
    # 调试打印：右侧乘积计算
    # print(f"右侧计算 位置 {i}: result[{i}] = {result[i]}, right_product = {right_product}")

return result

```

```

def test_product_except_self():
    """单元测试函数"""
    print("== 除自身以外数组的乘积单元测试 ==")
    solution = Solution()

    # 测试用例 1：基础情况
    nums1 = [1, 2, 3, 4]
    result1 = solution.productExceptSelf(nums1)
    print(f"测试用例 1 [1, 2, 3, 4]: {result1} (预期: [24, 12, 8, 6])")

    # 测试用例 2：包含 0 的数组
    nums2 = [-1, 1, 0, -3, 3]
    result2 = solution.productExceptSelf(nums2)
    print(f"测试用例 2 [-1, 1, 0, -3, 3]: {result2} (预期: [0, 0, 9, 0, 0])")

    # 测试用例 3：空数组
    nums3 = []
    result3 = solution.productExceptSelf(nums3)

```

```

print(f"测试用例 3 []: {result3} (预期: [])")

# 测试用例 4: 单元素数组
nums4 = [5]
result4 = solution.productExceptSelf(nums4)
print(f"测试用例 4 [5]: {result4} (预期: [1])")

# 测试用例 5: 负数数组
nums5 = [-2, -3, 4]
result5 = solution.productExceptSelf(nums5)
print(f"测试用例 5 [-2,-3,4]: {result5} (预期: [-12,-8,6])")

def performance_test():
    """性能测试函数"""
    print("\n==== 性能测试 ===")
    solution = Solution()
    size = 100000 # 10 万元素
    import random
    large_array = [random.randint(-30, 30) for _ in range(size)]

    import time
    start_time = time.time()
    result = solution.productExceptSelf(large_array)
    end_time = time.time()

    print(f"处理 {size} 个元素, 耗时: {end_time - start_time:.4f} 秒")

if __name__ == "__main__":
    # 运行单元测试
    test_product_except_self()

    # 运行性能测试 (可选)
    # performance_test()

    print("\n==== 测试完成 ===")

```

=====

文件: Code06\_MakeSumDivisibleByP.java

=====

```

package class046;

import java.util.HashMap;

```

```

// 使数组和能被 P 整除
// 给你一个正整数数组 nums, 请你移除 最短 子数组 (可以为空)
// 使得剩余元素的 和 能被 p 整除。 不允许 将整个数组都移除。
// 请你返回你需要移除的最短子数组的长度, 如果无法满足题目要求, 返回 -1。
// 子数组 定义为原数组中连续的一组元素。
// 测试链接 : https://leetcode.cn/problems/make-sum-divisible-by-p/
public class Code06_MakeSumDivisibleByP {

    public static int minSubarray(int[] nums, int p) {
        // 整体余数
        int mod = 0;
        for (int num : nums) {
            mod = (mod + num) % p;
        }
        if (mod == 0) {
            return 0;
        }
        // key : 前缀和%p 的余数
        // value : 最晚出现的位置
        HashMap<Integer, Integer> map = new HashMap<>();
        map.put(0, -1);
        int ans = Integer.MAX_VALUE;
        for (int i = 0, cur = 0, find; i < nums.length; i++) {
            // 0...i 这部分的余数
            cur = (cur + nums[i]) % p;
            find = cur >= mod ? (cur - mod) : (cur + p - mod);
            // find = (cur + p - mod) % p;
            if (map.containsKey(find)) {
                ans = Math.min(ans, i - map.get(find));
            }
            map.put(cur, i);
        }
        return ans == nums.length ? -1 : ans;
    }
}

```

=====

文件: Code07\_EvenCountsLongestSubarray.java

=====

```
package class046;
```

```
import java.util.Arrays;

// 每个元音包含偶数次的最长子字符串
// 给你一个字符串 s ，请你返回满足以下条件的最长子字符串的长度
// 每个元音字母，即 'a' , 'e' , 'i' , 'o' , 'u'
// 在子字符串中都恰好出现了偶数次。
// 测试链接 : https://leetcode.cn/problems/find-the-longest-substring-containing-vowels-in-even-counts/
public class Code07_EvenCountsLongestSubarray {

    public static int findTheLongestSubstring(String s) {
        int n = s.length();
        // 只有 5 个元音字符，状态就 5 位
        int[] map = new int[32];
        // map[0...31] = -2
        // map[01100] = -2, 这个状态之前没出现过
        Arrays.fill(map, -2);
        map[0] = -1;
        int ans = 0;
        for (int i = 0, status = 0, m; i < n; i++) {
            // status : 0....i-1 字符串上, aeiou 的奇偶性
            // s[i] = 当前字符
            // 情况 1 : 当前字符不是元音, status 不变
            // 情况 2 : 当前字符是元音, a^u(0^4), 修改相应状态
            m = move(s.charAt(i));
            if (m != -1) {
                status ^= 1 << m;
            }
            // status: 0....i 字符串上, aeiou 的奇偶性
            // 同样的状态, 之前最早出现在哪
            if (map[status] != -2) {
                ans = Math.max(ans, i - map[status]);
            } else {
                map[status] = i;
            }
        }
        return ans;
    }

    public static int move(char cha) {
        switch (cha) {
            case 'a': return 0;
```

```
        case 'e': return 1;
        case 'i': return 2;
        case 'o': return 3;
        case 'u': return 4;
        default: return -1;
    }
}

}

=====
```

文件: Code08\_ArrayManipulation.cpp

```
=====
/***
 * 数组操作 (Array Manipulation)
 *
 * 题目描述:
 * 给定一个长度为 n 的数组，初始时所有元素都为 0。然后进行 m 次操作，每次操作给定三个整数 a, b, k,
 * 表示将数组中从索引 a 到索引 b (包含 a 和 b) 的所有元素都增加 k。求执行完所有操作后数组中的最大值。
 *
 * 示例:
 * 输入: n = 5, queries = [[1, 2, 100], [2, 5, 100], [3, 4, 100]]
 * 输出: 200
 * 解释:
 * 初始数组: [0, 0, 0, 0, 0]
 * 操作 1: [100, 100, 0, 0, 0]
 * 操作 2: [100, 200, 100, 100, 100]
 * 操作 3: [100, 200, 200, 200, 100]
 * 最大值: 200
 *
 * 提示:
 * 3 <= n <= 10^7
 * 1 <= m <= 2*10^5
 * 1 <= a <= b <= n
 * 0 <= k <= 10^9
 *
 * 题目链接: https://www.hackerrank.com/challenges/crush/problem
 *
 * 解题思路:
 * 使用差分数组技巧结合前缀和来优化区间更新操作。
 * 1. 创建一个差分数组 diff，大小为 n+1
 * 2. 对于每个操作 [a, b, k]，执行 diff[a-1] += k 和 diff[b] -= k
```

- \* 3. 对差分数组计算前缀和，得到最终数组
- \* 4. 在计算前缀和的过程中记录最大值
- \*
- \* 时间复杂度： $O(n + m)$  – 需要遍历所有操作和数组一次
- \* 空间复杂度： $O(n)$  – 需要额外的差分数组空间
- \*
- \* 工程化考量：
  - \* 1. 边界条件处理：n=0、空操作数组等特殊情况
  - \* 2. 索引转换：题目使用 1-based 索引，需要转换为 0-based
  - \* 3. 整数溢出：使用 long long 避免大数溢出
  - \* 4. 性能优化：差分数组方法将  $O(n*m)$  优化到  $O(n+m)$
- \*
- \* 最优解分析：
  - \* 这是最优解，因为需要处理所有操作，而且数组大小可能很大，不能使用暴力方法。
  - \* 差分数组方法将时间复杂度从  $O(n*m)$  优化到  $O(n+m)$ 。
- \*
- \* 算法核心：
  - \* 差分数组的思想：对于区间 [a, b] 增加 k，只需要在 a 位置+k，在 b+1 位置-k。
  - \* 然后通过前缀和还原整个数组，同时记录最大值。
- \*/

```
#include <vector>
#include <iostream>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;

class Solution {
public:
    /**
     * 计算数组操作后的最大值
     *
     * @param n 数组长度
     * @param queries 操作数组，每个操作包含[起始索引, 结束索引, 增加值]
     * @return 操作后数组的最大值
     *
     * 异常场景处理：
     * - n <= 0: 返回 0
     * - queries 为空: 返回 0
     * - 索引越界: 题目保证 1 <= a <= b <= n
    */
}
```

```

* - 大数溢出：使用 long long 避免
*/
long long arrayManipulation(int n, vector<vector<int>>& queries) {
    // 边界情况处理
    if (n <= 0 || queries.empty()) {
        return 0;
    }

    // 创建差分数组，大小为 n+1 以便处理边界情况
    vector<long long> diff(n + 1, 0);

    // 处理每个操作
    for (auto& query : queries) {
        int a = query[0];          // 起始索引 (1-based)
        int b = query[1];          // 结束索引 (1-based)
        int k = query[2];          // 增加值

        // 在差分数组中标记区间更新
        // 在 a-1 位置增加 k (0-based 索引)
        diff[a - 1] += k;
        // 在 b 位置减少 k (如果 b < n, 避免数组越界)
        if (b < n) {
            diff[b] -= k;
        }
    }

    // 通过计算差分数组的前缀和得到最终数组，并记录最大值
    long long maxVal = LLONG_MIN;
    long long currentSum = 0;

    for (int i = 0; i < n; i++) {
        currentSum += diff[i];
        if (currentSum > maxVal) {
            maxVal = currentSum;
        }
    }

    return maxVal;
}

};

/***
 * 测试函数

```

```
/*
void testArrayManipulation() {
    Solution solution;

    // 测试用例 1: 经典情况
    int n1 = 5;
    vector<vector<int>> queries1 = {{1, 2, 100}, {2, 5, 100}, {3, 4, 100}};
    long long result1 = solution.arrayManipulation(n1, queries1);
    cout << "测试用例 1 n=5, queries=[[1,2,100], [2,5,100], [3,4,100]]: " << result1 << " (预期: 200)" << endl;

    // 测试用例 2: 复杂操作
    int n2 = 10;
    vector<vector<int>> queries2 = {{2, 6, 8}, {3, 5, 7}, {1, 8, 1}, {5, 9, 15}};
    long long result2 = solution.arrayManipulation(n2, queries2);
    cout << "测试用例 2 n=10, queries=[[2,6,8], [3,5,7], [1,8,1], [5,9,15]]: " << result2 << " (预期: 31)" << endl;

    // 测试用例 3: 边界情况
    int n3 = 4;
    vector<vector<int>> queries3 = {{1, 2, 5}, {2, 4, 10}, {1, 3, 3}};
    long long result3 = solution.arrayManipulation(n3, queries3);
    cout << "测试用例 3 n=4, queries=[[1,2,5], [2,4,10], [1,3,3]]: " << result3 << " (预期: 18)" << endl;

    // 测试用例 4: n=1
    int n4 = 1;
    vector<vector<int>> queries4 = {{1, 1, 5}};
    long long result4 = solution.arrayManipulation(n4, queries4);
    cout << "测试用例 4 n=1, queries=[[1,1,5]]: " << result4 << " (预期: 5)" << endl;

    // 测试用例 5: 空操作
    int n5 = 5;
    vector<vector<int>> queries5 = {};
    long long result5 = solution.arrayManipulation(n5, queries5);
    cout << "测试用例 5 n=5, queries=[]: " << result5 << " (预期: 0)" << endl;

    // 测试用例 6: n=0
    int n6 = 0;
    vector<vector<int>> queries6 = {{1, 1, 5}};
    long long result6 = solution.arrayManipulation(n6, queries6);
    cout << "测试用例 6 n=0, queries=[[1,1,5]]: " << result6 << " (预期: 0)" << endl;
```

```

// 测试用例 7: 大数测试
int n7 = 3;
vector<vector<int>> queries7 = {{1, 3, 1000000000}};
long long result7 = solution.arrayManipulation(n7, queries7);
cout << "测试用例 7 n=3, queries=[[1,3,1000000000]]: " << result7 << " (预期: 1000000000)" <<
endl;
}

/***
 * 性能测试函数
 */
void performanceTest() {
    cout << "\n== 性能测试 ==" << endl;
    Solution solution;
    int n = 100000; // 10 万元素 (减少规模避免超时)
    int m = 20000; // 2 万次操作

    // 生成随机操作
    vector<vector<int>> queries(m, vector<int>(3));
    srand(time(0));

    for (int i = 0; i < m; i++) {
        int a = rand() % n + 1; // 1-based 索引
        int b = a + rand() % (n - a + 1); // b >= a
        int k = rand() % 1000000000; // k 在 0 到 10^9 之间

        queries[i][0] = a;
        queries[i][1] = b;
        queries[i][2] = k;
    }

    auto startTime = chrono::high_resolution_clock::now();
    long long result = solution.arrayManipulation(n, queries);
    auto endTime = chrono::high_resolution_clock::now();

    auto duration = chrono::duration_cast<chrono::milliseconds>(endTime - startTime);
    cout << "处理 n=" << n << ", m=" << m << " 耗时: " << duration.count() << "ms" << endl;
    cout << "最大值: " << result << endl;
}

/***
 * 主函数
 */

```

```
int main() {
    cout << "==== 数组操作测试 ===" << endl;
    testArrayManipulation();

    // 运行性能测试（可选）
    // performanceTest();

    cout << "\n==== 测试完成 ===" << endl;
    return 0;
}
```

=====

文件: Code08\_ArrayManipulation.java

=====

```
package class046;

/**
 * 数组操作 (Array Manipulation)
 *
 * 题目描述:
 * 给定一个长度为 n 的数组，初始时所有元素都为 0。然后进行 m 次操作，每次操作给定三个整数 a, b, k,
 * 表示将数组中从索引 a 到索引 b (包含 a 和 b) 的所有元素都增加 k。求执行完所有操作后数组中的最大值。
 *
 * 示例:
 * 输入: n = 5, queries = [[1, 2, 100], [2, 5, 100], [3, 4, 100]]
 * 输出: 200
 *
 * 解释:
 * 初始数组: [0, 0, 0, 0, 0]
 * 操作 1: [100, 100, 0, 0, 0]
 * 操作 2: [100, 200, 100, 100, 100]
 * 操作 3: [100, 200, 200, 200, 100]
 * 最大值: 200
 *
 * 提示:
 * 3 <= n <= 10^7
 * 1 <= m <= 2*10^5
 * 1 <= a <= b <= n
 * 0 <= k <= 10^9
 *
 * 题目链接: https://www.hackerrank.com/challenges/crush/problem
 *
 * 解题思路:

```

- \* 使用差分数组技巧结合前缀和来优化区间更新操作。
  - \* 1. 创建一个差分数组 diff，大小为 n+1
  - \* 2. 对于每个操作 [a, b, k]，执行  $diff[a-1] += k$  和  $diff[b] -= k$
  - \* 3. 对差分数组计算前缀和，得到最终数组
  - \* 4. 在计算前缀和的过程中记录最大值
- \*
- \* 时间复杂度： $O(n + m)$  – 需要遍历所有操作和数组一次
- \* 空间复杂度： $O(n)$  – 需要额外的差分数组空间
- \*
- \* 工程化考量：
  - \* 1. 边界条件处理：n=0、空操作数组等特殊情况
  - \* 2. 索引转换：题目使用 1-based 索引，需要转换为 0-based
  - \* 3. 整数溢出：使用 long 避免大数溢出
  - \* 4. 性能优化：差分数组方法将  $O(n*m)$  优化到  $O(n+m)$
- \*
- \* 最优解分析：
  - \* 这是最优解，因为需要处理所有操作，而且数组大小可能很大，不能使用暴力方法。
  - \* 差分数组方法将时间复杂度从  $O(n*m)$  优化到  $O(n+m)$ 。
- \*
- \* 算法核心：
  - \* 差分数组的思想：对于区间 [a, b] 增加 k，只需要在 a 位置+k，在 b+1 位置-k。
  - \* 然后通过前缀和还原整个数组，同时记录最大值。
- \*
- \* 算法调试技巧：
  - \* 1. 打印中间过程：可以在处理每个操作后打印差分数组状态
  - \* 2. 边界测试：测试 n=1、操作重叠、边界索引等情况
  - \* 3. 性能测试：测试大规模数据下的性能表现
- \*
- \* 语言特性差异：
  - \* Java 中数组索引是 0-based，需要处理 1-based 到 0-based 的转换。
  - \* 与 C++ 相比，Java 有自动内存管理，无需手动释放数组内存。
  - \* 与 Python 相比，Java 是静态类型语言，需要显式声明数组类型。
- \*/

```
public class Code08_ArrayManipulation {
```

/\*\*  
 \* 计算数组操作后的最大值  
 \*  
 \* @param n 数组长度  
 \* @param queries 操作数组，每个操作包含[起始索引, 结束索引, 增加值]  
 \* @return 操作后数组的最大值  
 \*  
 \* 异常场景处理：

```

* - n <= 0: 返回 0
* - queries 为空: 返回 0
* - 索引越界: 题目保证 1 <= a <= b <= n
* - 大数溢出: 使用 long 避免
*
* 边界条件:
* - n=1 的情况
* - 操作重叠的情况
* - 边界索引 (a=1, b=n)
* - k=0 的情况
*/
public static long arrayManipulation(int n, int[][] queries) {
    // 边界情况处理
    if (n <= 0 || queries == null || queries.length == 0) {
        return 0;
    }

    // 创建差分数组, 大小为 n+1 以便处理边界情况
    long[] diff = new long[n + 1];

    // 处理每个操作
    for (int[] query : queries) {
        int a = query[0];      // 起始索引 (1-based)
        int b = query[1];      // 结束索引 (1-based)
        int k = query[2];      // 增加值

        // 在差分数组中标记区间更新
        // 在 a-1 位置增加 k (0-based 索引)
        diff[a - 1] += k;
        // 在 b 位置减少 k (如果 b < n, 避免数组越界)
        if (b < n) {
            diff[b] -= k;
        }

        // 调试打印: 显示每个操作后的差分数组状态
        // System.out.println("操作: [" + a + ", " + b + ", " + k + "]");
        // System.out.println("差分数组: " + java.util.Arrays.toString(diff));
    }

    // 通过计算差分数组的前缀和得到最终数组, 并记录最大值
    long maxVal = Long.MIN_VALUE;
    long currentSum = 0;

```

```
for (int i = 0; i < n; i++) {
    currentSum += diff[i];
    if (currentSum > maxVal) {
        maxVal = currentSum;
    }
}

// 调试打印：显示前缀和计算过程
// System.out.println("位置 " + i + ": 当前和 = " + currentSum + ", 最大值 = " +
maxVal);
}

return maxVal;
}

/**
 * 单元测试方法
 */
public static void testArrayManipulation() {
    System.out.println("==> 数组操作单元测试 ==>");

    // 测试用例 1: 经典情况
    int n1 = 5;
    int[][] queries1 = {{1, 2, 100}, {2, 5, 100}, {3, 4, 100}};
    long result1 = arrayManipulation(n1, queries1);
    System.out.println("测试用例 1 n=5, queries=[[1,2,100], [2,5,100], [3,4,100]]: " + result1 +
" (预期: 200)");

    // 测试用例 2: 复杂操作
    int n2 = 10;
    int[][] queries2 = {{2, 6, 8}, {3, 5, 7}, {1, 8, 1}, {5, 9, 15}};
    long result2 = arrayManipulation(n2, queries2);
    System.out.println("测试用例 2 n=10, queries=[[2,6,8], [3,5,7], [1,8,1], [5,9,15]]: " +
result2 + " (预期: 31)");

    // 测试用例 3: 边界情况
    int n3 = 4;
    int[][] queries3 = {{1, 2, 5}, {2, 4, 10}, {1, 3, 3}};
    long result3 = arrayManipulation(n3, queries3);
    System.out.println("测试用例 3 n=4, queries=[[1,2,5], [2,4,10], [1,3,3]]: " + result3 + "
(预期: 18)");

    // 测试用例 4: n=1
    int n4 = 1;
```

```

int[][] queries4 = {{1, 1, 5}};
long result4 = arrayManipulation(n4, queries4);
System.out.println("测试用例 4 n=1, queries=[[1,1,5]]: " + result4 + " (预期: 5)");

// 测试用例 5: 空操作
int n5 = 5;
int[][] queries5 = {};
long result5 = arrayManipulation(n5, queries5);
System.out.println("测试用例 5 n=5, queries=[]: " + result5 + " (预期: 0)");

// 测试用例 6: n=0
int n6 = 0;
int[][] queries6 = {{1, 1, 5}};
long result6 = arrayManipulation(n6, queries6);
System.out.println("测试用例 6 n=0, queries=[[1,1,5]]: " + result6 + " (预期: 0)");

// 测试用例 7: 大数测试
int n7 = 3;
int[][] queries7 = {{1, 3, 1000000000}};
long result7 = arrayManipulation(n7, queries7);
System.out.println("测试用例 7 n=3, queries=[[1,3,1000000000]]: " + result7 + " (预期: 1000000000)");
}

/***
 * 性能测试方法
 */
public static void performanceTest() {
    System.out.println(
"== 性能测试 ==");
    int n = 10000000; // 1000 万元素
    int m = 200000; // 20 万次操作

    // 生成随机操作
    int[][] queries = new int[m][3];
    java.util.Random random = new java.util.Random();

    for (int i = 0; i < m; i++) {
        int a = random.nextInt(n) + 1; // 1-based 索引
        int b = a + random.nextInt(n - a + 1); // b >= a
        int k = random.nextInt(1000000000); // k 在 0 到 10^9 之间

        queries[i][0] = a;

```

```

        queries[i][1] = b;
        queries[i][2] = k;
    }

    long startTime = System.currentTimeMillis();
    long result = arrayManipulation(n, queries);
    long endTime = System.currentTimeMillis();

    System.out.println("处理 n=" + n + ", m=" + m + " 耗时: " + (endTime - startTime) +
"ms");
    System.out.println("最大值: " + result);
}

/**
 * 主函数 - 测试入口
 */
public static void main(String[] args) {
    // 运行单元测试
    testArrayManipulation();

    // 运行性能测试 (可选)
    // performanceTest();

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

```

文件: Code08\_ArrayManipulation.py

```
=====
"""

```

数组操作 (Array Manipulation)

题目描述:

给定一个长度为  $n$  的数组，初始时所有元素都为 0。然后进行  $m$  次操作，每次操作给定三个整数  $a$ ,  $b$ ,  $k$ ，表示将数组中从索引  $a$  到索引  $b$  (包含  $a$  和  $b$ ) 的所有元素都增加  $k$ 。求执行完所有操作后数组中的最大值。

示例:

输入:  $n = 5$ ,  $\text{queries} = [[1, 2, 100], [2, 5, 100], [3, 4, 100]]$

输出: 200

解释:

初始数组: [0, 0, 0, 0, 0]  
操作 1: [100, 100, 0, 0, 0]  
操作 2: [100, 200, 100, 100, 100]  
操作 3: [100, 200, 200, 200, 100]  
最大值: 200

提示:

$3 \leq n \leq 10^7$   
 $1 \leq m \leq 2 \times 10^5$   
 $1 \leq a \leq b \leq n$   
 $0 \leq k \leq 10^9$

题目链接: <https://www.hackerrank.com/challenges/crush/problem>

解题思路:

使用差分数组技巧结合前缀和来优化区间更新操作。

1. 创建一个差分数组 diff，大小为  $n+1$
2. 对于每个操作  $[a, b, k]$ ，执行  $diff[a-1] += k$  和  $diff[b] -= k$
3. 对差分数组计算前缀和，得到最终数组
4. 在计算前缀和的过程中记录最大值

时间复杂度:  $O(n + m)$  – 需要遍历所有操作和数组一次

空间复杂度:  $O(n)$  – 需要额外的差分数组空间

工程化考量:

1. 边界条件处理:  $n=0$ 、空操作列表等特殊情况
2. 性能优化: 使用差分数组避免  $O(n*m)$  的时间复杂度
3. 空间优化: 只存储差分数组，不存储整个数组
4. 大数处理:  $k$  可能达到  $10^9$ ，需要确保整数范围

最优解分析:

这是最优解，时间复杂度  $O(n+m)$ ，空间复杂度  $O(n)$ 。

对于大规模数据，直接操作数组会超时，必须使用差分数组技巧。

数学原理:

差分数组  $d[i] = arr[i] - arr[i-1]$

区间  $[a, b]$  加  $k$  等价于:  $d[a] += k, d[b+1] -= k$

然后通过前缀和还原原数组:  $arr[i] = arr[i-1] + d[i]$

算法调试技巧:

1. 打印中间过程: 显示差分数组和前缀和的计算过程
2. 边界测试: 测试  $n=0$ 、单个操作等特殊情况
3. 性能测试: 测试大规模数据下的性能表现

语言特性差异:

Python 的整数范围较大，无需担心溢出问题。

Java/C++需要考虑整数溢出，可能需要使用 long 类型。

"""

```
def array_manipulation(n, queries):
```

"""

计算数组操作后的最大值

:param n: 数组长度

:param queries: 操作数组，每个操作包含[起始索引, 结束索引, 增加值]

:return: 操作后数组的最大值

异常场景处理:

- n=0: 返回 0
- 空操作列表: 返回 0
- 边界索引处理: 确保索引在有效范围内

边界条件:

- n=0 或 n=1
- 操作列表为空
- 操作索引超出范围
- k=0 的操作 (无影响)

"""

# 边界情况处理

```
if n <= 0 or not queries:  
    return 0
```

# 创建差分数组，大小为 n+1 以便处理边界情况

# 使用 n+1 可以避免边界检查，索引从 0 到 n-1

```
diff = [0] * (n + 1)
```

# 处理每个操作，时间复杂度 O(m)

```
for query in queries:
```

a = query[0] # 起始索引 (1-based)

b = query[1] # 结束索引 (1-based)

k = query[2] # 增加值

# 在差分数组中标记区间更新

# 在 a-1 位置加上 k (因为数组是 0-based 索引)

```
    diff[a - 1] += k
```

```

# 在 b 位置减去 k (如果 b 在范围内)
if b < n:
    diff[b] -= k

# 调试打印: 显示操作效果
# print(f"操作 [{a}, {b}], {k}]: diff[{a-1}] += {k}, diff[{b}] -= {k}")

# 通过计算差分数组的前缀和得到最终数组，并记录最大值，时间复杂度 O(n)
max_val = float('-inf')
current_sum = 0

for i in range(n):
    current_sum += diff[i]
    max_val = max(max_val, current_sum)
    # 调试打印: 显示前缀和计算过程
    # print(f"位置 {i}: 当前值 = {current_sum}, 最大值 = {max_val}")

return max_val

def test_array_manipulation():
    """单元测试函数"""
    print("== 数组操作单元测试 ==")

    # 测试用例 1: 题目示例
    n1 = 5
    queries1 = [[1, 2, 100], [2, 5, 100], [3, 4, 100]]
    result1 = array_manipulation(n1, queries1)
    print(f"测试用例 1 n=5, queries=[[1,2,100],[2,5,100],[3,4,100]]: {result1} (预期: 200)")

    # 测试用例 2: 多个操作重叠
    n2 = 10
    queries2 = [[2, 6, 8], [3, 5, 7], [1, 8, 1], [5, 9, 15]]
    result2 = array_manipulation(n2, queries2)
    print(f"测试用例 2 n=10, queries=[[2,6,8],[3,5,7],[1,8,1],[5,9,15]]: {result2} (预期: 31)")

    # 测试用例 3: 空操作
    n3 = 5
    queries3 = []
    result3 = array_manipulation(n3, queries3)
    print(f"测试用例 3 n=5, queries=[]: {result3} (预期: 0)")

    # 测试用例 4: 单个操作

```

```

n4 = 3
queries4 = [[1, 3, 5]]
result4 = array_manipulation(n4, queries4)
print(f"测试用例 4 n=3, queries=[[1,3,5]]: {result4} (预期: 5)")

# 测试用例 5: 边界情况 n=0
n5 = 0
queries5 = [[1, 1, 10]]
result5 = array_manipulation(n5, queries5)
print(f"测试用例 5 n=0, queries=[[1,1,10]]: {result5} (预期: 0)")

def performance_test():
    """性能测试函数"""
    print(
        """
==== 性能测试 ====
    """
    )
    n = 10000000 # 1000 万元素
    m = 200000 # 20 万次操作
    import random
    queries = []
    for _ in range(m):
        a = random.randint(1, n)
        b = random.randint(a, n)
        k = random.randint(0, 1000000000)
        queries.append([a, b, k])

    import time
    start_time = time.time()
    result = array_manipulation(n, queries)
    end_time = time.time()

    print(f"处理 n={n}, m={m}, 最大值: {result}, 耗时: {end_time - start_time:.4f} 秒")

if __name__ == "__main__":
    # 运行单元测试
    test_array_manipulation()

    # 运行性能测试 (可选)
    # performance_test()

    print(
        """
==== 测试完成 ====
"""
    )

```

文件: Code09\_RangeSumQuery2D.cpp

```
=====
/**  
 * 二维区域和检索 - 矩阵不可变 (Range Sum Query 2D - Immutable)  
 *  
 * 题目描述:  
 * 给定一个二维矩阵 matrix，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1)，右下角为 (row2, col2)。  
 *  
 * 实现 NumMatrix 类:  
 * - NumMatrix(int[][] matrix) 用整数矩阵 matrix 初始化对象  
 * - int sumRegion(int row1, int col1, int row2, int col2) 返回左上角 (row1, col1)、右下角 (row2, col2) 的子矩阵的元素总和。  
 *  
 * 示例:  
 * 输入:  
 * ["NumMatrix","sumRegion","sumRegion","sumRegion"]  
 *  
 * [[[3,0,1,4,2],[5,6,3,2,1],[1,2,0,1,5],[4,1,0,1,7],[1,0,3,0,5]], [2,1,4,3], [1,1,2,2], [1,2,2,4]]  
 * 输出:  
 * [null, 8, 11, 12]  
 *  
 * 提示:  
 * m == matrix.length  
 * n == matrix[i].length  
 * 1 <= m, n <= 200  
 * -10^5 <= matrix[i][j] <= 10^5  
 * 0 <= row1 <= row2 < m  
 * 0 <= col1 <= col2 < n  
 * 最多调用 10^4 次 sumRegion 方法  
 *  
 * 题目链接: https://leetcode.com/problems/range-sum-query-2d-immutable/  
 *  
 * 解题思路:  
 * 使用二维前缀和技巧:  
 * 1. 创建一个二维前缀和数组 dp，其中 dp[i][j] 表示从 (0,0) 到 (i-1, j-1) 的矩形区域和  
 * 2. 初始化时计算整个矩阵的前缀和  
 * 3. 查询时使用公式: sum = dp[row2+1][col2+1] - dp[row1][col2+1] - dp[row2+1][col1] +  
 dp[row1][col1]  
 *  
 * 时间复杂度:  
 * - 初始化: O(m*n) - 需要遍历整个矩阵构建前缀和数组
```

- \* - 查询: O(1) - 每次查询只需要常数时间
- \* 空间复杂度: O(m\*n) - 需要额外的前缀和数组空间
- \*
- \* 工程化考量:
  - \* 1. 边界条件处理: 空矩阵、单元素矩阵
  - \* 2. 性能优化: 预处理前缀和, 查询时 O(1) 时间
  - \* 3. 空间优化: 必须存储前缀和数组, 无法避免
  - \* 4. 大数处理: 元素值可能很大, 需要确保整数范围
- \*
- \* 最优解分析:
  - \* 这是最优解, 时间复杂度 O(m\*n) 初始化, O(1) 查询。
  - \* 对于频繁查询的场景, 预处理是必要的。
- \*
- \* 数学原理:
  - \* 二维前缀和公式:
$$dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + matrix[i-1][j-1]$$
- \*
- \* 区域和公式:
$$sum = dp[r2+1][c2+1] - dp[r1][c2+1] - dp[r2+1][c1] + dp[r1][c1]$$
- \*
- \* 算法调试技巧:
  - \* 1. 打印中间过程: 显示前缀和数组的计算过程
  - \* 2. 边界测试: 测试空矩阵、单元素矩阵等特殊情况
  - \* 3. 性能测试: 测试大规模矩阵下的性能表现
- \*
- \* 语言特性差异:
  - \* C++需要手动管理内存, 使用 vector 容器可以简化内存管理。
  - \* 与 Python/Java 相比, C++有更好的性能但语法更复杂。

```
#include <iostream>
#include <vector>
#include <cassert>
#include <chrono>
#include <cstdlib>
#include <algorithm>

using namespace std;

class NumMatrix {
private:
    vector<vector<int>> dp; // 二维前缀和数组
```

```

public:
    /**
     * 初始化二维前缀和数组
     *
     * @param matrix 输入矩阵
     *
     * 异常场景处理:
     * - 空矩阵: 创建空的前缀和数组
     * - 单元素矩阵: 正常处理
     * - 非矩形矩阵: 题目保证是矩形矩阵
     *
     * 边界条件:
     * - 矩阵为空
     * - 矩阵只有一行或一列
     * - 查询范围超出矩阵边界
    */
    NumMatrix(vector<vector<int>>& matrix) {
        // 边界情况处理
        if (matrix.empty() || matrix[0].empty()) {
            return;
        }

        int m = matrix.size();
        int n = matrix[0].size();

        // 创建二维前缀和数组, 大小为(m+1) x (n+1)
        // 使用 m+1 和 n+1 可以避免边界检查
        dp.resize(m + 1, vector<int>(n + 1, 0));

        // 计算前缀和, 时间复杂度 O(m*n)
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                // 前缀和公式: 上 + 左 - 左上 + 当前元素
                dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + matrix[i-1][j-1];
                // 调试打印: 显示前缀和计算过程
                // cout << "位置 (" << i-1 << ", " << j-1 << "): dp[" << i << "][" << j << "] = "
                << dp[i][j] << endl;
            }
        }
    }

    /**
     * 计算子矩阵区域和
    */

```

```

*
* @param row1, col1: 左上角坐标
* @param row2, col2: 右下角坐标
* @return 子矩阵元素和
*
* 边界条件:
* - 坐标超出范围 (题目保证有效)
* - 单元素查询
* - 整个矩阵查询
*/
int sumRegion(int row1, int col1, int row2, int col2) {
    // 使用前缀和公式计算区域和, 时间复杂度 O(1)
    // 公式: 右下 - 右上 - 左下 + 左上
    int result = dp[row2+1][col2+1] - dp[row1][col2+1] - dp[row2+1][col1] + dp[row1][col1];

    // 调试打印: 显示查询过程
    // cout << "查询区域 [" << row1 << ", " << col1 << "] 到 [" << row2 << ", " << col2 << "]: 结果 = " << result << endl;

    return result;
}

// 单元测试函数
void testNumMatrix() {
    cout << "==== 二维区域和检索单元测试 ===" << endl;

    // 测试用例 1: 题目示例
    vector<vector<int>> matrix1 = {
        {3, 0, 1, 4, 2},
        {5, 6, 3, 2, 1},
        {1, 2, 0, 1, 5},
        {4, 1, 0, 1, 7},
        {1, 0, 3, 0, 5}
    };

    NumMatrix numMatrix1(matrix1);

    // 测试查询
    int result1 = numMatrix1.sumRegion(2, 1, 4, 3); // 预期: 8
    int result2 = numMatrix1.sumRegion(1, 1, 2, 2); // 预期: 11
    int result3 = numMatrix1.sumRegion(1, 2, 2, 4); // 预期: 12
}

```

```

cout << "测试用例 1: " << result1 << ", " << result2 << ", " << result3 << " (预期: 8, 11,
12)" << endl;
assert(result1 == 8);
assert(result2 == 11);
assert(result3 == 12);

// 测试用例 2: 空矩阵
vector<vector<int>> matrix2;
NumMatrix numMatrix2(matrix2);
// 空矩阵应该能正常构造, 但查询可能有问题, 这里不测试查询

cout << "测试用例 2 空矩阵: 构造成功" << endl;

// 测试用例 3: 单元素矩阵
vector<vector<int>> matrix3 = {{5}};
NumMatrix numMatrix3(matrix3);
int result5 = numMatrix3.sumRegion(0, 0, 0, 0); // 预期: 5

cout << "测试用例 3 单元素: " << result5 << " (预期: 5)" << endl;
assert(result5 == 5);

// 测试用例 4: 单行矩阵
vector<vector<int>> matrix4 = {{1, 2, 3, 4, 5}};
NumMatrix numMatrix4(matrix4);
int result6 = numMatrix4.sumRegion(0, 1, 0, 3); // 预期: 9 (2+3+4)

cout << "测试用例 4 单行矩阵: " << result6 << " (预期: 9)" << endl;
assert(result6 == 9);

cout << "==== 单元测试通过 ===" << endl;
}

// 性能测试函数
void performanceTest() {
    cout << "\n==== 性能测试 ===" << endl;

    // 创建大规模矩阵
    int m = 200, n = 200; // 最大规模
    vector<vector<int>> matrix(m, vector<int>(n, 1)); // 全 1 矩阵

    // 测试初始化性能
    auto start = chrono::high_resolution_clock::now();
    NumMatrix numMatrix(matrix);

```

```

auto end = chrono::high_resolution_clock::now();
auto initTime = chrono::duration_cast<chrono::microseconds>(end - start).count();

// 测试多次查询性能
int queryCount = 10000;
start = chrono::high_resolution_clock::now();
for (int i = 0; i < queryCount; i++) {
    int row1 = rand() % m;
    int col1 = rand() % n;
    int row2 = min(row1 + rand() % 10, m - 1);
    int col2 = min(col1 + rand() % 10, n - 1);
    numMatrix.sumRegion(row1, col1, row2, col2);
}
end = chrono::high_resolution_clock::now();
auto queryTime = chrono::duration_cast<chrono::microseconds>(end - start).count();

cout << "初始化 " << m << "x" << n << " 矩阵耗时: " << initTime << " 微秒" << endl;
cout << queryCount << " 次查询耗时: " << queryTime << " 微秒" << endl;
cout << " 平均每次查询耗时: " << static_cast<double>(queryTime) / queryCount << " 微秒" << endl;
}

int main() {
    // 运行单元测试
    testNumMatrix();

    // 运行性能测试（可选）
    // performanceTest();

    cout << "\n==== 测试完成 ===" << endl;
    return 0;
}
=====
```

文件: Code09\_RangeSumQuery2D.java

```

package class046;

/**
 * 二维区域和检索 - 矩阵不可变 (Range Sum Query 2D - Immutable)
 *
 * 题目描述:
```

\* 给定一个二维矩阵 matrix，处理以下类型的多个查询：计算其子矩形范围内元素的总和，  
 \* 该子矩阵的左上角为 (row1, col1)，右下角为 (row2, col2)。  
 \*  
 \* 实现 NumMatrix 类：  
 \* - NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化  
 \* - int sumRegion(int row1, int col1, int row2, int col2) 返回子矩阵元素总和  
 \*  
 \* 示例：  
 \* 输入：  
 \* ["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]  
 \* [[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]  
 \* 输出：  
 \* [null, 8, 11, 12]  
 \*  
 \* 解释：  
 \* NumMatrix numMatrix = new  
 NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]);  
 \* numMatrix.sumRegion(2, 1, 4, 3); // return 8 (红色矩形框的元素总和)  
 \* numMatrix.sumRegion(1, 1, 2, 2); // return 11 (绿色矩形框的元素总和)  
 \* numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩形框的元素总和)  
 \*  
 \* 提示：  
 \* m == matrix.length  
 \* n == matrix[i].length  
 \* 1 <= m, n <= 200  
 \* -10^5 <= matrix[i][j] <= 10^5  
 \* 0 <= row1 <= row2 < m  
 \* 0 <= col1 <= col2 < n  
 \* 最多调用 10^4 次 sumRegion 方法  
 \*  
 \* 题目链接：<https://leetcode.com/problems/range-sum-query-2d-immutable/>  
 \*  
 \* 解题思路：  
 \* 使用二维前缀和预处理技术。  
 \* 1. 构建二维前缀和数组 preSum，其中 preSum[i][j] 表示从 (0, 0) 到 (i-1, j-1) 的矩形区域内所有元素的和  
 \* 2. 利用容斥原理计算任意子矩阵的和：  
 \*     sumRegion(row1, col1, row2, col2) =  
 \*     preSum[row2+1][col2+1] - preSum[row1][col2+1] - preSum[row2+1][col1] + preSum[row1][col1]  
 \*  
 \* 时间复杂度：  
 \* - 初始化：O(m\*n) - 需要遍历整个矩阵构建前缀和数组  
 \* - 查询：O(1) - 每次查询只需要常数时间

```

* 空间复杂度: O(m*n) - 需要额外的前缀和数组空间
*
* 这是最优解，因为查询次数可能很多，预处理后可以实现 O(1) 查询时间。
*/
public class Code09_RangeSumQuery2D {
    private int[][] preSum;

    /**
     * 构造函数，初始化二维前缀和数组
     *
     * @param matrix 输入的二维矩阵
     */
    public Code09_RangeSumQuery2D(int[][] matrix) {
        // 边界情况处理
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return;
        }

        int m = matrix.length;
        int n = matrix[0].length;

        // 创建前缀和数组，多一行一列便于处理边界情况
        preSum = new int[m + 1][n + 1];

        // 构建二维前缀和数组
        // preSum[i][j] 表示从(0,0)到(i-1, j-1)的矩形区域内所有元素的和
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                preSum[i][j] = preSum[i - 1][j] + preSum[i][j - 1] - preSum[i - 1][j - 1] +
matrix[i - 1][j - 1];
            }
        }
    }

    /**
     * 计算指定子矩阵的元素总和
     *
     * @param row1 子矩阵左上角行索引
     * @param col1 子矩阵左上角列索引
     * @param row2 子矩阵右下角行索引
     * @param col2 子矩阵右下角列索引
     * @return 子矩阵元素总和
     */

```

```

public int sumRegion(int row1, int col1, int row2, int col2) {
    // 使用容斥原理计算子矩阵和
    return preSum[row2 + 1][col2 + 1] - preSum[row1][col2 + 1] - preSum[row2 + 1][col1] +
preSum[row1][col1];
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例
    int[][] matrix = {
        {3, 0, 1, 4, 2},
        {5, 6, 3, 2, 1},
        {1, 2, 0, 1, 5},
        {4, 1, 0, 1, 7},
        {1, 0, 3, 0, 5}
    };
    Code09_RangeSumQuery2D numMatrix = new Code09_RangeSumQuery2D(matrix);

    // 测试查询 1: (2, 1, 4, 3) -> 应该返回 8
    int result1 = numMatrix.sumRegion(2, 1, 4, 3);
    System.out.println("测试查询 1 (2,1,4,3): " + result1); // 预期输出: 8

    // 测试查询 2: (1, 1, 2, 2) -> 应该返回 11
    int result2 = numMatrix.sumRegion(1, 1, 2, 2);
    System.out.println("测试查询 2 (1,1,2,2): " + result2); // 预期输出: 11

    // 测试查询 3: (1, 2, 2, 4) -> 应该返回 12
    int result3 = numMatrix.sumRegion(1, 2, 2, 4);
    System.out.println("测试查询 3 (1,2,2,4): " + result3); // 预期输出: 12
}
}

```

文件: Code09\_RangeSumQuery2D.py

"""
二维区域和检索 - 矩阵不可变 (Range Sum Query 2D - Immutable)

题目描述:

给定一个二维矩阵 matrix，处理以下类型的多个查询：计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1)，右下角为 (row2, col2)。

实现 NumMatrix 类：

- NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化
- int sumRegion(int row1, int col1, int row2, int col2) 返回子矩阵元素总和

示例：

输入：

```
["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
[[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]]
```

输出：

```
[null, 8, 11, 12]
```

解释：

```
NumMatrix numMatrix = new
NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]);
numMatrix.sumRegion(2, 1, 4, 3); // return 8 (红色矩形框的元素总和)
numMatrix.sumRegion(1, 1, 2, 2); // return 11 (绿色矩形框的元素总和)
numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩形框的元素总和)
```

提示：

```
m == matrix.length
n == matrix[i].length
1 <= m, n <= 200
-10^5 <= matrix[i][j] <= 10^5
0 <= row1 <= row2 < m
0 <= col1 <= col2 < n
最多调用 10^4 次 sumRegion 方法
```

题目链接：<https://leetcode.com/problems/range-sum-query-2d-immutable/>

解题思路：

使用二维前缀和预处理技术。

1. 构建二维前缀和数组 preSum，其中 preSum[i][j] 表示从 (0, 0) 到 (i-1, j-1) 的矩形区域内所有元素的和
2. 利用容斥原理计算任意子矩阵的和：

```
sumRegion(row1, col1, row2, col2) =
preSum[row2+1][col2+1] - preSum[row1][col2+1] - preSum[row2+1][col1] + preSum[row1][col1]
```

时间复杂度：

- 初始化：O(m\*n) - 需要遍历整个矩阵构建前缀和数组
- 查询：O(1) - 每次查询只需要常数时间

空间复杂度：O(m\*n) - 需要额外的前缀和数组空间

这是最优解，因为查询次数可能很多，预处理后可以实现 O(1) 查询时间。

"""

```
class NumMatrix:
```

```
    def __init__(self, matrix):
```

"""

构造函数，初始化二维前缀和数组

:param matrix: 输入的二维矩阵

"""

# 边界情况处理

```
        if not matrix or not matrix[0]:
```

```
            return
```

m, n = len(matrix), len(matrix[0])

# 创建前缀和数组，多一行一列便于处理边界情况

```
        self.preSum = [[0] * (n + 1) for _ in range(m + 1)]
```

# 构建二维前缀和数组

# preSum[i][j] 表示从(0, 0)到(i-1, j-1)的矩形区域内所有元素的和

```
        for i in range(1, m + 1):
```

```
            for j in range(1, n + 1):
```

```
                self.preSum[i][j] = self.preSum[i - 1][j] + self.preSum[i][j - 1] - self.preSum[i
```

```
- 1][j - 1] + \
```

```
                matrix[i - 1][j - 1]
```

```
    def sumRegion(self, row1, col1, row2, col2):
```

"""

计算指定子矩阵的元素总和

:param row1: 子矩阵左上角行索引

:param col1: 子矩阵左上角列索引

:param row2: 子矩阵右下角行索引

:param col2: 子矩阵右下角列索引

:return: 子矩阵元素总和

"""

# 使用容斥原理计算子矩阵和

```
        return self.preSum[row2 + 1][col2 + 1] - self.preSum[row1][col2 + 1] - self.preSum[row2 +
```

```
1][col1] + \
```

```
        self.preSum[row1][col1]
```

```

def main():
    """测试用例"""
    # 测试用例
    matrix = [
        [3, 0, 1, 4, 2],
        [5, 6, 3, 2, 1],
        [1, 2, 0, 1, 5],
        [4, 1, 0, 1, 7],
        [1, 0, 3, 0, 5]
    ]

    numMatrix = NumMatrix(matrix)

    # 测试查询 1: (2, 1, 4, 3) -> 应该返回 8
    result1 = numMatrix.sumRegion(2, 1, 4, 3)
    print("测试查询 1 (2,1,4,3):", result1)  # 预期输出: 8

    # 测试查询 2: (1, 1, 2, 2) -> 应该返回 11
    result2 = numMatrix.sumRegion(1, 1, 2, 2)
    print("测试查询 2 (1,1,2,2):", result2)  # 预期输出: 11

    # 测试查询 3: (1, 2, 2, 4) -> 应该返回 12
    result3 = numMatrix.sumRegion(1, 2, 2, 4)
    print("测试查询 3 (1,2,2,4):", result3)  # 预期输出: 12

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

文件: Code10\_RangeSumQuery2D.cpp

```
=====
/***
 * 二维区域和检索 - 矩阵不可变 (Range Sum Query 2D - Immutable)
 *
 * 题目描述:
 * 给定一个二维矩阵 matrix，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1)，右下角为 (row2, col2)。
 *
 * 实现 NumMatrix 类:
```

```
* - NumMatrix(int[][] matrix) 用整数矩阵 matrix 初始化对象
* - int sumRegion(int row1, int col1, int row2, int col2) 返回左上角 (row1, col1)、右下角 (row2, col2) 的子矩阵的元素总和。
*
* 示例：
* 输入：
* ["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
*
[[[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]
*
* 输出：
* [null, 8, 11, 12]
*
* 提示：
* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 200
* -10^5 <= matrix[i][j] <= 10^5
* 0 <= row1 <= row2 < m
* 0 <= col1 <= col2 < n
* 最多调用 10^4 次 sumRegion 方法
*
* 题目链接: https://leetcode.com/problems/range-sum-query-2d-immutable/
*
* 解题思路：
* 使用二维前缀和技巧：
* 1. 创建一个二维前缀和数组 dp，其中 dp[i][j] 表示从 (0,0) 到 (i-1, j-1) 的矩形区域和
* 2. 初始化时计算整个矩阵的前缀和
* 3. 查询时使用公式：sum = dp[row2+1][col2+1] - dp[row1][col2+1] - dp[row2+1][col1] + dp[row1][col1]
*
* 时间复杂度：
* - 初始化: O(m*n) - 需要遍历整个矩阵构建前缀和数组
* - 查询: O(1) - 每次查询只需要常数时间
* 空间复杂度: O(m*n) - 需要额外的前缀和数组空间
*
* 工程化考量：
* 1. 边界条件处理: 空矩阵、单元素矩阵
* 2. 性能优化: 预处理前缀和, 查询时 O(1) 时间
* 3. 空间优化: 必须存储前缀和数组, 无法避免
* 4. 大数处理: 元素值可能很大, 需要确保整数范围
*
* 最优解分析：
* 这是最优解, 时间复杂度 O(m*n) 初始化, O(1) 查询。
```

- \* 对于频繁查询的场景，预处理是必要的。
- \*
- \* 数学原理：
- \* 二维前缀和公式：
- \*  $dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + matrix[i-1][j-1]$
- \*
- \* 区域和公式：
- \*  $sum = dp[r2+1][c2+1] - dp[r1][c2+1] - dp[r2+1][c1] + dp[r1][c1]$
- \*
- \* 算法调试技巧：
- \* 1. 打印中间过程：显示前缀和数组的计算过程
- \* 2. 边界测试：测试空矩阵、单元素矩阵等特殊情况
- \* 3. 性能测试：测试大规模矩阵下的性能表现
- \*
- \* 语言特性差异：
- \* C++需要手动管理内存，使用 vector 容器可以简化内存管理。
- \* 与 Python/Java 相比，C++有更好的性能但语法更复杂。
- \*/

```
#include <iostream>
#include <vector>
#include <cassert>
#include <chrono>
#include <cstdlib>
#include <algorithm>

using namespace std;

class NumMatrix {
private:
    vector<vector<int>> dp; // 二维前缀和数组

public:
    /**
     * 初始化二维前缀和数组
     *
     * @param matrix 输入矩阵
     *
     * 异常场景处理：
     * - 空矩阵：创建空的前缀和数组
     * - 单元素矩阵：正常处理
     * - 非矩形矩阵：题目保证是矩形矩阵
     */
}
```

```

* 边界条件:
* - 矩阵为空
* - 矩阵只有一行或一列
* - 查询范围超出矩阵边界
*/
NumMatrix(vector<vector<int>>& matrix) {
    // 边界情况处理
    if (matrix.empty() || matrix[0].empty()) {
        return;
    }

    int m = matrix.size();
    int n = matrix[0].size();

    // 创建二维前缀和数组, 大小为(m+1) x (n+1)
    // 使用 m+1 和 n+1 可以避免边界检查
    dp.resize(m + 1, vector<int>(n + 1, 0));

    // 计算前缀和, 时间复杂度 O(m*n)
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            // 前缀和公式: 上 + 左 - 左上 + 当前元素
            dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i-1][j-1] + matrix[i-1][j-1];
            // 调试打印: 显示前缀和计算过程
            // cout << "位置 (" << i-1 << ", " << j-1 << ")": dp[" << i << "][" << j << "] = "
            << dp[i][j] << endl;
        }
    }
}

/**
 * 计算子矩阵区域和
 *
 * @param row1, col1: 左上角坐标
 * @param row2, col2: 右下角坐标
 * @return 子矩阵元素和
 *
 * 边界条件:
 * - 坐标超出范围 (题目保证有效)
 * - 单元素查询
 * - 整个矩阵查询
 */
int sumRegion(int row1, int col1, int row2, int col2) {

```

```

// 使用前缀和公式计算区域和，时间复杂度 O(1)
// 公式：右下 - 右上 - 左下 + 左上
int result = dp[row2+1][col2+1] - dp[row1][col2+1] - dp[row2+1][col1] + dp[row1][col1];

// 调试打印：显示查询过程
// cout << "查询区域 [" << row1 << ", " << col1 << "] 到 [" << row2 << ", " << col2 << "]: 结果 = " << result << endl;

return result;
}

};

// 单元测试函数
void testNumMatrix() {
    cout << "==== 二维区域和检索单元测试 ===" << endl;

    // 测试用例 1：题目示例
    vector<vector<int>> matrix1 = {
        {3, 0, 1, 4, 2},
        {5, 6, 3, 2, 1},
        {1, 2, 0, 1, 5},
        {4, 1, 0, 1, 7},
        {1, 0, 3, 0, 5}
    };

    NumMatrix numMatrix1(matrix1);

    // 测试查询
    int result1 = numMatrix1.sumRegion(2, 1, 4, 3); // 预期: 8
    int result2 = numMatrix1.sumRegion(1, 1, 2, 2); // 预期: 11
    int result3 = numMatrix1.sumRegion(1, 2, 2, 4); // 预期: 12

    cout << "测试用例 1: " << result1 << ", " << result2 << ", " << result3 << " (预期: 8, 11, 12)" << endl;
    assert(result1 == 8);
    assert(result2 == 11);
    assert(result3 == 12);

    // 测试用例 2：空矩阵
    vector<vector<int>> matrix2;
    NumMatrix numMatrix2(matrix2);
    // 空矩阵应该能正常构造，但查询可能有问题，这里不测试查询

```

```

cout << "测试用例 2 空矩阵: 构造成功" << endl;

// 测试用例 3: 单元素矩阵
vector<vector<int>> matrix3 = {{5}};
NumMatrix numMatrix3(matrix3);
int result5 = numMatrix3.sumRegion(0, 0, 0, 0); // 预期: 5

cout << "测试用例 3 单元素: " << result5 << "(预期: 5)" << endl;
assert(result5 == 5);

// 测试用例 4: 单行矩阵
vector<vector<int>> matrix4 = {{1, 2, 3, 4, 5}};
NumMatrix numMatrix4(matrix4);
int result6 = numMatrix4.sumRegion(0, 1, 0, 3); // 预期: 9 (2+3+4)

cout << "测试用例 4 单行矩阵: " << result6 << "(预期: 9)" << endl;
assert(result6 == 9);

cout << "==== 单元测试通过 ===" << endl;
}

// 性能测试函数
void performanceTest() {
    cout << "\n==== 性能测试 ===" << endl;

    // 创建大规模矩阵
    int m = 200, n = 200; // 最大规模
    vector<vector<int>> matrix(m, vector<int>(n, 1)); // 全 1 矩阵

    // 测试初始化性能
    auto start = chrono::high_resolution_clock::now();
    NumMatrix numMatrix(matrix);
    auto end = chrono::high_resolution_clock::now();
    auto initTime = chrono::duration_cast<chrono::microseconds>(end - start).count();

    // 测试多次查询性能
    int queryCount = 10000;
    start = chrono::high_resolution_clock::now();
    for (int i = 0; i < queryCount; i++) {
        int row1 = rand() % m;
        int col1 = rand() % n;
        int row2 = min(row1 + rand() % 10, m - 1);
        int col2 = min(col1 + rand() % 10, n - 1);
    }
}

```

```

    numMatrix.sumRegion(row1, col1, row2, col2);
}

end = chrono::high_resolution_clock::now();
auto queryTime = chrono::duration_cast<chrono::microseconds>(end - start).count();

cout << "初始化 " << m << "x" << n << " 矩阵耗时: " << initTime << " 微秒" << endl;
cout << queryCount << " 次查询耗时: " << queryTime << " 微秒" << endl;
cout << "平均每次查询耗时: " << static_cast<double>(queryTime) / queryCount << " 微秒" <<
endl;
}

int main() {
    // 运行单元测试
    testNumMatrix();

    // 运行性能测试（可选）
    // performanceTest();

    cout << "\n==== 测试完成 ===" << endl;
    return 0;
}
=====
```

文件: Code10\_RikkaWithPrefixSum.java

```

package class046;

/**
 * Rikka with Prefix Sum (牛客网题目)
 *
 * 题目描述:
 * 给定一个长度为 n 初始全为 0 的数列 A。m 次操作，要求支持以下三种操作：
 * 1. 区间加一个数 v
 * 2. 全局修改，对于每一个 i，把 Ai 改成原序列前 i 项的和（前缀和）
 * 3. 区间求和
 *
 * 示例:
 * 输入:
 * 1
 * 100000 7
 * 1 1 3 1
 * 2
```

```

* 3 2333 6666
* 2
* 3 2333 6666
* 2
* 3 2333 6666
*
* 输出:
* 13002
* 58489497
* 12043005
*
* 提示:
*  $1 \leq n, m \leq 10^5$ 
*  $1 \leq L \leq R \leq n$ 
*  $0 \leq v \leq 10^9$ 
* 查询操作不超过 500 次
*
* 解题思路:
* 使用差分数组和组合数学来优化操作。
* 1. 对于操作 1 (区间加), 使用差分数组记录
* 2. 对于操作 2 (前缀和), 记录前缀和操作的次数
* 3. 对于操作 3 (区间求和), 使用组合数学公式计算结果
*
* 核心思想: 如果在  $(i, j)$  点 +w, 那么  $(x, y)$  点的系数为  $C(x-i+y-j-1, x-i-1)$ 
*
* 时间复杂度:  $O(m * \text{查询次数})$ 
* 空间复杂度:  $O(m)$ 
*
* 这是一个高级的前缀和应用题目, 结合了差分数组和组合数学。
*/
public class Code10_RikkaWithPrefixSum {
    private static final int MOD = 998244353;
    private static final int N = 200010;

    // 预计算阶乘和逆元
    private static long[] fac = new long[N];
    private static long[] inv = new long[N];

    static {
        // 预处理阶乘
        fac[0] = 1;
        for (int i = 1; i < N; i++) {
            fac[i] = fac[i - 1] * i % MOD;
        }
    }
}

```

```

}

// 预处理逆元
inv[N - 1] = qmod(fac[N - 1], MOD - 2);
for (int i = N - 2; i >= 0; i--) {
    inv[i] = inv[i + 1] * (i + 1) % MOD;
}
}

/***
* 快速幂运算
*
* @param x 底数
* @param y 指数
* @return (x^y) % MOD
*/
private static long qmod(long x, long y) {
    x %= MOD;
    long ans = 1;
    while (y > 0) {
        if ((y & 1) == 1) {
            ans = ans * x % MOD;
        }
        x = x * x % MOD;
        y >>= 1;
    }
    return ans;
}

/***
* 计算组合数 C(a, b)
*
* @param a 组合数上标
* @param b 组合数下标
* @return C(a, b) % MOD
*/
private static long C(int a, int b) {
    if (b > a || b < 0) {
        return 0;
    }
    return fac[a] * inv[b] % MOD * inv[a - b] % MOD;
}

```

```

/**
 * 主要解法函数
 *
 * @param n 数组长度
 * @param operations 操作数组
 * @return 查询结果数组
 */
public static long[] solve(int n, int[][] operations) {
    // 记录操作 1 的信息
    // 每个操作存储: [操作次数, 位置, 值]
    int[][] ops = new int[operations.length][3];
    int opCount = 0;

    // 当前前缀和操作次数
    int prefixSumCount = 1;

    // 结果数组
    long[] results = new long[operations.length];
    int resultCount = 0;

    for (int[] op : operations) {
        if (op[0] == 1) {
            // 操作 1: 区间加一个数
            int l = op[1];
            int r = op[2];
            int v = op[3];

            // 使用差分数组思想记录操作
            ops[opCount][0] = prefixSumCount - 1; // 记录当前前缀和操作次数
            ops[opCount][1] = l; // 起始位置
            ops[opCount][2] = v % MOD; // 值
            opCount++;
        }

        ops[opCount][0] = prefixSumCount - 1; // 记录当前前缀和操作次数
        ops[opCount][1] = r + 1; // 结束位置+1
        ops[opCount][2] = -(v % MOD); // 负值
        opCount++;

    } else if (op[0] == 2) {
        // 操作 2: 全局前缀和
        prefixSumCount++;
    } else {
        // 操作 3: 区间求和
        int l = op[1];

```

```

        int r = op[2];

        // 计算区间和
        long ans = (fun(prefixSumCount + 1, r, ops, opCount) -
                    fun(prefixSumCount + 1, l - 1, ops, opCount)) % MOD;
        ans = (ans + MOD) % MOD; // 确保结果为正

        results[resultCount++] = ans;
    }
}

// 返回实际结果数组
long[] finalResults = new long[resultCount];
System.arraycopy(results, 0, finalResults, 0, resultCount);
return finalResults;
}

/***
 * 辅助函数，计算贡献值
 *
 * @param x x 坐标
 * @param y y 坐标
 * @param ops 操作数组
 * @param opCount 操作数量
 * @return 贡献值
 */
private static long fun(int x, int y, int[][] ops, int opCount) {
    long ans = 0;
    for (int i = 0; i < opCount; i++) {
        if (ops[i][0] < x && ops[i][1] <= y) {
            ans = (ans + C(x - ops[i][0] + y - ops[i][1] - 1, x - ops[i][0] - 1) *
                   (long) ops[i][2]) % MOD;
        }
    }
    return ans;
}

/***
 * 测试用例
 */
public static void main(String[] args) {
    // 简化测试用例
    int n = 5;
}

```

```

int[][] operations = {
    {1, 1, 3, 1}, // 区间[1,3]加1
    {2},           // 前缀和操作
    {3, 2, 4}      // 查询区间[2,4]的和
};

long[] results = solve(n, operations);

System.out.println("测试结果:");
for (long result : results) {
    System.out.println(result);
}
}
}
=====
```

文件: Code10\_RikkaWithPrefixSum.py

```
=====
"""
Rikka with Prefix Sum (牛客网题目)
```

题目描述:

给定一个长度为 n 初始全为 0 的数列 A。m 次操作，要求支持以下三种操作：

1. 区间加一个数 v
2. 全局修改，对于每一个 i，把  $A_i$  改成原序列前 i 项的和（前缀和）
3. 区间求和

示例:

输入:

```

1
100000 7
1 1 3 1
2
3 2333 6666
2
3 2333 6666
2
3 2333 6666
```

输出:

```

13002
58489497
```

12043005

提示:

$1 \leq n, m \leq 10^5$

$1 \leq L \leq R \leq n$

$0 \leq v \leq 10^9$

查询操作不超过 500 次

解题思路:

使用差分数组和组合数学来优化操作。

1. 对于操作 1 (区间加), 使用差分数组记录
2. 对于操作 2 (前缀和), 记录前缀和操作的次数
3. 对于操作 3 (区间求和), 使用组合数学公式计算结果

核心思想: 如果在  $(i, j)$  点 +w, 那么  $(x, y)$  点的系数为  $C(x-i+y-j-1, x-i-1)$

时间复杂度:  $O(m * \text{查询次数})$

空间复杂度:  $O(m)$

这是一个高级的前缀和应用题目, 结合了差分数组和组合数学。

"""

```
class RikkaWithPrefixSum:  
    MOD = 998244353  
    N = 200010  
  
    def __init__(self):  
        # 预计算阶乘和逆元  
        self.fac = [0] * self.N  
        self.inv = [0] * self.N  
  
        # 预处理阶乘  
        self.fac[0] = 1  
        for i in range(1, self.N):  
            self.fac[i] = self.fac[i - 1] * i % self.MOD  
  
        # 预处理逆元  
        self.inv[self.N - 1] = self.qmod(self.fac[self.N - 1], self.MOD - 2)  
        for i in range(self.N - 2, -1, -1):  
            self.inv[i] = self.inv[i + 1] * (i + 1) % self.MOD  
  
    def qmod(self, x, y):
```

```
"""
```

## 快速幂运算

```
:param x: 底数
:param y: 指数
:return: (x ^ y) % MOD
"""
x %= self.MOD
ans = 1
while y > 0:
    if y & 1:
        ans = ans * x % self.MOD
    x = x * x % self.MOD
    y >>= 1
return ans
```

```
def C(self, a, b):
```

```
"""
计算组合数 C(a, b)
```

```
:param a: 组合数上标
:param b: 组合数下标
:return: C(a, b) % MOD
"""
if b > a or b < 0:
    return 0
return self.fac[a] * self.inv[b] % self.MOD * self.inv[a - b] % self.MOD
```

```
def fun(self, x, y, ops, op_count):
```

```
"""
辅助函数，计算贡献值
```

```
:param x: x 坐标
:param y: y 坐标
:param ops: 操作数组
:param op_count: 操作数量
:return: 贡献值
"""
ans = 0
for i in range(op_count):
    if ops[i][0] < x and ops[i][1] <= y:
        ans = (ans + self.C(x - ops[i][0] + y - ops[i][1] - 1, x - ops[i][0] - 1) *
               ops[i][2]) % self.MOD
```

```

return ans

def solve(self, n, operations):
    """
    主要解法函数

    :param n: 数组长度
    :param operations: 操作数组
    :return: 查询结果数组
    """

    # 记录操作 1 的信息
    # 每个操作存储: [操作次数, 位置, 值]
    ops = [[0, 0, 0] for _ in range(len(operations) * 2)]
    op_count = 0

    # 当前前缀和操作次数
    prefix_sum_count = 1

    # 结果数组
    results = [0] * len(operations)
    result_count = 0

    for op in operations:
        if op[0] == 1:
            # 操作 1: 区间加一个数
            l, r, v = op[1], op[2], op[3]

            # 使用差分数组思想记录操作
            ops[op_count][0] = prefix_sum_count - 1 # 记录当前前缀和操作次数
            ops[op_count][1] = l # 起始位置
            ops[op_count][2] = v % self.MOD # 值
            op_count += 1

            ops[op_count][0] = prefix_sum_count - 1 # 记录当前前缀和操作次数
            ops[op_count][1] = r + 1 # 结束位置+1
            ops[op_count][2] = -(v % self.MOD) # 负值
            op_count += 1

        elif op[0] == 2:
            # 操作 2: 全局前缀和
            prefix_sum_count += 1

        else:
            # 操作 3: 区间求和
            l, r = op[1], op[2]

```

```

# 计算区间和
ans = (self.fun(prefix_sum_count + 1, r, ops, op_count) -
        self.fun(prefix_sum_count + 1, l - 1, ops, op_count)) % self.MOD
ans = (ans + self.MOD) % self.MOD # 确保结果为正

results[result_count] = ans
result_count += 1

# 返回实际结果数组
return results[:result_count]

def main():
    """测试用例"""
    # 创建解法实例
    solver = RikkaWithPrefixSum()

    # 简化测试用例
    n = 5
    operations = [
        [1, 1, 3, 1], # 区间[1,3]加1
        [2], # 前缀和操作
        [3, 2, 4] # 查询区间[2,4]的和
    ]

    results = solver.solve(n, operations)

    print("测试结果:")
    for result in results:
        print(result)

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

文件: Code11\_GoodSubarrays.java

```

=====
package class046;

import java.util.HashMap;
```

```
import java.util.Map;

/**
 * Good Subarrays (Codeforces 1398C)
 *
 * 题目描述:
 * 给定一个由数字字符组成的字符串 s, 定义“好数组”为: 数组中所有元素的和等于元素个数。
 * 求字符串 s 的所有连续子串中, 有多少个“好数组”。
 *
 * 示例:
 * 输入: s = "111"
 * 输出: 3
 * 解释: 子串"1"(位置 0), "1"(位置 1), "1"(位置 2)都是好数组
 *
 * 输入: s = "123"
 * 输出: 4
 * 解释: 子串"1", "2", "3", "123"都是好数组
 *
 * 输入: s = "101010"
 * 输出: 15
 *
 * 提示:
 * 1 <= t <= 1000 (测试用例数)
 * 1 <= |s| <= 10^5 (字符串长度)
 * 字符串只包含数字字符'0'-'9'
 * 所有测试用例的字符串长度总和不超过 10^5
 *
 * 题目链接: https://codeforces.com/contest/1398/problem/C
 *
 * 解题思路:
 * 将问题转换为前缀和问题。
 * 1. 对于一个子数组 s[1..r], 它是好数组当且仅当  $\sum(s[1..r]) = r - l + 1$ 
 * 2. 通过移项得到:  $\sum(s[1..r]) - (r - l + 1) = 0$ 
 * 3. 定义  $b[i] = a[i] - 1$ , 其中  $a[i]$  是字符串中第  $i$  个字符的数值
 * 4. 那么条件变为:  $\sum(b[1..r]) = 0$ 
 * 5. 进一步转换为前缀和:  $\text{prefix}[r] - \text{prefix}[l-1] = 0$ , 即  $\text{prefix}[r] = \text{prefix}[l-1]$ 
 * 6. 使用哈希表统计每个前缀和出现的次数, 对于每个前缀和值, 如果有  $k$  次出现, 则可以形成  $k*(k-1)/2$  个好数组
 *
 * 时间复杂度: O(n) - 需要遍历字符串一次
 * 空间复杂度: O(n) - 哈希表最多存储 n 个不同的前缀和
 *
 * 这是最优解, 因为需要处理所有字符。
```

```
*/
```

```
public class Code11_GoodSubarrays {
```

```
/**
```

```
* 计算好数组的个数
```

```
*
```

```
* @param s 输入字符串
```

```
* @return 好数组的个数
```

```
*/
```

```
public static long goodSubarrays(String s) {
```

```
// 边界情况处理
```

```
if (s == null || s.length() == 0) {
```

```
    return 0;
```

```
}
```

```
int n = s.length();
```

```
// 使用哈希表记录前缀和及其出现次数
```

```
Map<Long, Integer> map = new HashMap<>();
```

```
// 初始化: 前缀和为 0 出现 1 次 (表示空前缀)
```

```
map.put(0L, 1);
```

```
long count = 0; // 结果计数
```

```
long prefix = 0; // 当前前缀和
```

```
// 遍历字符串
```

```
for (int i = 0; i < n; i++) {
```

```
// 将字符转换为数字并减 1
```

```
int digit = s.charAt(i) - '0';
```

```
int b = digit - 1;
```

```
// 更新前缀和
```

```
prefix += b;
```

```
// 如果当前前缀和之前出现过, 说明存在好数组
```

```
if (map.containsKey(prefix)) {
```

```
    count += map.get(prefix);
```

```
}
```

```
// 更新当前前缀和的出现次数
```

```
map.put(prefix, map.getOrDefault(prefix, 0) + 1);
```

```
}
```

```

        return count;
    }

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    String s1 = "111";
    long result1 = goodSubarrays(s1);
    // 预期输出: 3
    System.out.println("测试用例 1 \" + s1 + "\": " + result1);

    // 测试用例 2
    String s2 = "123";
    long result2 = goodSubarrays(s2);
    // 预期输出: 4
    System.out.println("测试用例 2 \" + s2 + "\": " + result2);

    // 测试用例 3
    String s3 = "101010";
    long result3 = goodSubarrays(s3);
    // 预期输出: 15
    System.out.println("测试用例 3 \" + s3 + "\": " + result3);

    // 测试用例 4
    String s4 = "000";
    long result4 = goodSubarrays(s4);
    // 预期输出: 6
    System.out.println("测试用例 4 \" + s4 + "\": " + result4);
}
}

```

=====

文件: Code11\_GoodSubarrays.py

=====

"""

Good Subarrays (Codeforces 1398C)

题目描述:

给定一个由数字字符组成的字符串  $s$ , 定义“好数组”为: 数组中所有元素的和等于元素个数。  
求字符串  $s$  的所有连续子串中, 有多少个“好数组”。

示例：

输入： s = "111"

输出： 3

解释： 子串"1"(位置 0), "1"(位置 1), "1"(位置 2)都是好数组

输入： s = "123"

输出： 4

解释： 子串"1", "2", "3", "123"都是好数组

输入： s = "101010"

输出： 15

提示：

$1 \leq t \leq 1000$  (测试用例数)

$1 \leq |s| \leq 10^5$  (字符串长度)

字符串只包含数字字符'0'-'9'

所有测试用例的字符串长度总和不超过  $10^5$

题目链接：<https://codeforces.com/contest/1398/problem/C>

解题思路：

将问题转换为前缀和问题。

1. 对于一个子数组  $s[1..r]$ , 它是好数组当且仅当  $\text{sum}(s[1..r]) = r-1+1$
2. 通过移项得到：  $\text{sum}(s[1..r]) - (r-1+1) = 0$
3. 定义  $b[i] = a[i] - 1$ , 其中  $a[i]$  是字符串中第  $i$  个字符的数值
4. 那么条件变为：  $\text{sum}(b[1..r]) = 0$
5. 进一步转换为前缀和：  $\text{prefix}[r] - \text{prefix}[1-1] = 0$ , 即  $\text{prefix}[r] = \text{prefix}[1-1]$
6. 使用哈希表统计每个前缀和出现的次数，对于每个前缀和值，如果有  $k$  次出现，则可以形成  $k*(k-1)/2$  个好数组

时间复杂度：  $O(n)$  – 需要遍历字符串一次

空间复杂度：  $O(n)$  – 哈希表最多存储  $n$  个不同的前缀和

这是最优解，因为需要处理所有字符。

"""

```
def good_subarrays(s):
```

```
    """
```

```
        计算好数组的个数
```

```
:param s: 输入字符串
```

```
:return: 好数组的个数
"""
# 边界情况处理
if not s:
    return 0

n = len(s)

# 使用字典记录前缀和及其出现次数
prefix_count = {0: 1} # 初始化: 前缀和为 0 出现 1 次 (表示空前缀)

count = 0 # 结果计数
prefix = 0 # 当前前缀和

# 遍历字符串
for i in range(n):
    # 将字符转换为数字并减 1
    digit = int(s[i])
    b = digit - 1

    # 更新前缀和
    prefix += b

    # 如果当前前缀和之前出现过, 说明存在好数组
    if prefix in prefix_count:
        count += prefix_count[prefix]

    # 更新当前前缀和的出现次数
    prefix_count[prefix] = prefix_count.get(prefix, 0) + 1

return count

def main():
    """测试用例"""
    # 测试用例 1
    s1 = "111"
    result1 = good_subarrays(s1)
    # 预期输出: 3
    print(f'测试用例 1 "{s1}": {result1}')

    # 测试用例 2
    s2 = "123"
```

```

result2 = good_subarrays(s2)
# 预期输出: 4
print(f' 测试用例 2 "{s2}": {result2}')

# 测试用例 3
s3 = "101010"
result3 = good_subarrays(s3)
# 预期输出: 15
print(f' 测试用例 3 "{s3}": {result3}')

# 测试用例 4
s4 = "000"
result4 = good_subarrays(s4)
# 预期输出: 6
print(f' 测试用例 4 "{s4}": {result4}')

```

```

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

文件: Code12\_SubarrayDivisibility.java

```

=====
```

```

package class046;

import java.util.HashMap;
import java.util.Map;

/**
 * Subarray Divisibility (CSES 1662)
 *
 * 题目描述:
 * 给定一个包含 n 个整数的数组和一个正整数 m，计算有多少个连续子数组的元素和可以被 m 整除。
 *
 * 示例:
 * 输入: n = 5, m = 3, array = [3, 2, 5, 1, 7]
 * 输出: 4
 * 解释: 子数组[3], [2,5,1], [3,2,5,1], [7]的和分别为 3, 8, 11, 7, 都可以被 3 整除
 *
 * 提示:
 * 1 <= n <= 2*10^5
 * 1 <= m <= 10^3

```

```

* -10^9 <= array[i] <= 10^9
*
* 题目链接: https://cses.fi/problemset/task/1662
*
* 解题思路:
* 使用前缀和与模运算的性质。
* 1. 对于子数组[i, j], 其和可被 m 整除当且仅当(prefix[j] - prefix[i-1]) % m = 0
* 2. 即 prefix[j] % m = prefix[i-1] % m
* 3. 因此, 我们只需要统计具有相同模 m 值的前缀和的个数
* 4. 对于每个模值 k, 如果有 c 个前缀和模 m 等于 k, 那么可以形成 c*(c-1)/2 个满足条件的子数组
* 5. 特别地, 模值为 0 的前缀和本身就可以构成满足条件的子数组 (从开头到当前位置的子数组)
*
* 时间复杂度: O(n) - 需要遍历数组一次
* 空间复杂度: O(m) - 哈希表最多存储 m 个不同的模值
*
* 这是最优解, 因为需要处理所有元素。
*/
public class Code12_SubarrayDivisibility {

    /**
     * 计算和可被 m 整除的子数组个数
     *
     * @param array 输入数组
     * @param m 除数
     * @return 和可被 m 整除的子数组个数
     */
    public static long subarrayDivisibility(int[] array, int m) {
        // 边界情况处理
        if (array == null || array.length == 0 || m <= 0) {
            return 0;
        }

        int n = array.length;

        // 使用哈希表记录每个模值出现的次数
        Map<Integer, Integer> map = new HashMap<>();
        // 初始化: 模值为 0 出现 1 次 (表示空前缀)
        map.put(0, 1);

        long count = 0;      // 结果计数
        long prefix = 0;     // 当前前缀和

        // 遍历数组

```

```
for (int i = 0; i < n; i++) {
    // 更新前缀和
    prefix += array[i];

    // 计算前缀和对 m 的模值（处理负数情况）
    int mod = (int) (prefix % m);
    if (mod < 0) {
        mod += m;
    }

    // 如果该模值之前出现过，说明存在满足条件的子数组
    if (map.containsKey(mod)) {
        count += map.get(mod);
    }

    // 更新该模值的出现次数
    map.put(mod, map.getOrDefault(mod, 0) + 1);
}

return count;
}

/***
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] array1 = {3, 2, 5, 1, 7};
    int m1 = 3;
    long result1 = subarrayDivisibility(array1, m1);
    // 预期输出: 4
    System.out.println("测试用例 1: " + result1);

    // 测试用例 2
    int[] array2 = {1, 2, 3, 4, 5};
    int m2 = 2;
    long result2 = subarrayDivisibility(array2, m2);
    // 预期输出: 6
    System.out.println("测试用例 2: " + result2);

    // 测试用例 3
    int[] array3 = {-1, 5, 0, -2};
    int m3 = 3;
```

```
long result3 = subarrayDivisibility(array3, m3);
// 预期输出: 3
System.out.println("测试用例 3: " + result3);
}
=====
```

文件: Code12\_SubarrayDivisibility.py

```
=====
"""
Subarray Divisibility (CSES 1662)
```

题目描述:

给定一个包含  $n$  个整数的数组和一个正整数  $m$ , 计算有多少个连续子数组的元素和可以被  $m$  整除。

示例:

输入:  $n = 5, m = 3, \text{array} = [3, 2, 5, 1, 7]$

输出: 4

解释: 子数组  $[3], [2, 5, 1], [3, 2, 5, 1], [7]$  的和分别为  $3, 8, 11, 7$ , 都可以被 3 整除

提示:

$1 \leq n \leq 2 \times 10^5$

$1 \leq m \leq 10^3$

$-10^9 \leq \text{array}[i] \leq 10^9$

题目链接: <https://cses.fi/problemset/task/1662>

解题思路:

使用前缀和与模运算的性质。

1. 对于子数组  $[i, j]$ , 其和可被  $m$  整除当且仅当  $(\text{prefix}[j] - \text{prefix}[i-1]) \% m = 0$
2. 即  $\text{prefix}[j] \% m = \text{prefix}[i-1] \% m$
3. 因此, 我们只需要统计具有相同模  $m$  值的前缀和的个数
4. 对于每个模值  $k$ , 如果有  $c$  个前缀和模  $m$  等于  $k$ , 那么可以形成  $c*(c-1)/2$  个满足条件的子数组
5. 特别地, 模值为 0 的前缀和本身就可以构成满足条件的子数组 (从开头到当前位置的子数组)

时间复杂度:  $O(n)$  – 需要遍历数组一次

空间复杂度:  $O(m)$  – 哈希表最多存储  $m$  个不同的模值

这是最优解, 因为需要处理所有元素。

```
"""
```

```
def subarray_divisibility(array, m):
    """
    计算和可被 m 整除的子数组个数

    :param array: 输入数组
    :param m: 除数
    :return: 和可被 m 整除的子数组个数
    """

    # 边界情况处理
    if not array or m <= 0:
        return 0

    n = len(array)

    # 使用字典记录每个模值出现的次数
    mod_count = {0: 1}  # 初始化: 模值为 0 出现 1 次 (表示空前缀)

    count = 0  # 结果计数
    prefix = 0  # 当前前缀和

    # 遍历数组
    for i in range(n):
        # 更新前缀和
        prefix += array[i]

        # 计算前缀和对 m 的模值 (处理负数情况)
        mod = prefix % m
        if mod < 0:
            mod += m

        # 如果该模值之前出现过, 说明存在满足条件的子数组
        if mod in mod_count:
            count += mod_count[mod]

        # 更新该模值的出现次数
        mod_count[mod] = mod_count.get(mod, 0) + 1

    return count

def main():
    """测试用例"""
    # 测试用例 1
```

```

array1 = [3, 2, 5, 1, 7]
m1 = 3
result1 = subarray_divisibility(array1, m1)
# 预期输出: 4
print("测试用例 1:", result1)

# 测试用例 2
array2 = [1, 2, 3, 4, 5]
m2 = 2
result2 = subarray_divisibility(array2, m2)
# 预期输出: 6
print("测试用例 2:", result2)

# 测试用例 3
array3 = [-1, 5, 0, -2]
m3 = 3
result3 = subarray_divisibility(array3, m3)
# 预期输出: 3
print("测试用例 3:", result3)

```

```

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

文件: Code13\_FindPivotIndex.cpp

```

=====
```

```

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

/**
 * 寻找数组中心索引 (Find Pivot Index)
 *
 * 题目描述:
 * 给你一个整数数组 nums，请编写一个能够返回数组 “中心索引”的方法。
 * 中心索引是数组的一个索引，其左侧所有元素相加的和等于右侧所有元素相加的和。
 * 如果数组不存在中心索引，返回 -1。如果数组有多个中心索引，应该返回最靠近左边的那一个。
 *
 * 注意：中心索引可能出现在数组的两端。
 *
 * 示例：

```

```
* 输入: nums = [1, 7, 3, 6, 5, 6]
* 输出: 3
* 解释:
* 索引 3 (nums[3] = 6) 的左侧数之和 (1+7+3 = 11)，右侧数之和 (5+6 = 11)，二者相等。
*
* 输入: nums = [1, 2, 3]
* 输出: -1
* 解释: 数组中不存在满足此条件的中心索引。
*
* 输入: nums = [2, 1, -1]
* 输出: 0
* 解释:
* 索引 0 的左侧不存在元素，视作和为 0；右侧数之和为 1 + (-1) = 0，二者相等。
*
* 提示:
* nums 的长度范围为 [0, 10000]。
* 任何一个 nums[i] 将会是一个范围在 [-1000, 1000] 的整数。
*
* 题目链接: https://leetcode.com/problems/find-pivot-index/
*
* 解题思路:
* 使用前缀和的思想，计算整个数组的总和，然后遍历数组，维护左侧元素的和，
* 当左侧和等于总和减去左侧和减去当前元素时，找到中心索引。
*
* 时间复杂度: O(n) - 需要遍历数组两次
* 空间复杂度: O(1) - 只使用常数额外空间
*/

```

```
class Solution {
public:
    /**
     * 寻找数组的中心索引
     *
     * @param nums 输入数组
     * @return 中心索引，如果不存在返回-1
     */
    int pivotIndex(vector<int>& nums) {
        // 边界情况处理
        if (nums.empty()) {
            return -1;
        }

        // 计算数组总和

```

```
int totalSum = 0;
for (int num : nums) {
    totalSum += num;
}

// 维护左侧元素的和
int leftSum = 0;

// 遍历数组寻找中心索引
for (int i = 0; i < nums.size(); i++) {
    // 右侧和 = 总和 - 左侧和 - 当前元素
    int rightSum = totalSum - leftSum - nums[i];

    // 如果左侧和等于右侧和, 返回当前索引
    if (leftSum == rightSum) {
        return i;
    }

    // 更新左侧和, 包括当前元素
    leftSum += nums[i];
}

// 没有找到中心索引
return -1;
};

int main() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {1, 7, 3, 6, 5, 6};
    int result1 = solution.pivotIndex(nums1);
    // 预期输出: 3
    cout << "测试用例 1: " << result1 << endl;

    // 测试用例 2
    vector<int> nums2 = {1, 2, 3};
    int result2 = solution.pivotIndex(nums2);
    // 预期输出: -1
    cout << "测试用例 2: " << result2 << endl;

    // 测试用例 3
}
```

```
vector<int> nums3 = {2, 1, -1};  
int result3 = solution.pivotIndex(nums3);  
// 预期输出: 0  
cout << "测试用例 3: " << result3 << endl;  
  
// 测试用例 4  
vector<int> nums4 = {};  
int result4 = solution.pivotIndex(nums4);  
// 预期输出: -1  
cout << "测试用例 4: " << result4 << endl;  
  
return 0;  
}
```

=====

文件: Code13\_FindPivotIndex.java

```
=====  
package class046;  
  
/**  
 * 寻找数组中心索引 (Find Pivot Index)  
 *  
 * 题目描述:  
 * 给你一个整数数组 nums，请编写一个能够返回数组“中心索引”的方法。  
 * 中心索引是数组的一个索引，其左侧所有元素相加的和等于右侧所有元素相加的和。  
 * 如果数组不存在中心索引，返回 -1。如果数组有多个中心索引，应该返回最靠近左边的那一个。  
 *  
 * 注意：中心索引可能出现在数组的两端。  
 *  
 * 示例:  
 * 输入: nums = [1, 7, 3, 6, 5, 6]  
 * 输出: 3  
 * 解释:  
 * 索引 3 (nums[3] = 6) 的左侧数之和 (1+7+3 = 11)，右侧数之和 (5+6 = 11)，二者相等。  
 *  
 * 输入: nums = [1, 2, 3]  
 * 输出: -1  
 * 解释: 数组中不存在满足此条件的中心索引。  
 *  
 * 输入: nums = [2, 1, -1]  
 * 输出: 0  
 * 解释:
```

```
* 索引 0 的左侧不存在元素，视作和为 0；右侧数之和为  $1 + (-1) = 0$ ，二者相等。  
*  
* 提示：  
* nums 的长度范围为 [0, 10000]。  
* 任何一个 nums[i] 将会是一个范围在 [-1000, 1000] 的整数。  
*  
* 题目链接：https://leetcode.com/problems/find-pivot-index/  
*  
* 解题思路：  
* 使用前缀和的思想，计算整个数组的总和，然后遍历数组，维护左侧元素的和，  
* 当左侧和等于总和减去左侧和减去当前元素时，找到中心索引。  
*  
* 时间复杂度：O(n) - 需要遍历数组两次  
* 空间复杂度：O(1) - 只使用常数额外空间  
*/
```

```
public class Code13_FindPivotIndex {  
  
    /**  
     * 寻找数组的中心索引  
     *  
     * @param nums 输入数组  
     * @return 中心索引，如果不存在返回-1  
     */  
  
    public static int pivotIndex(int[] nums) {  
        // 边界情况处理  
        if (nums == null || nums.length == 0) {  
            return -1;  
        }  
  
        // 计算数组总和  
        int totalSum = 0;  
        for (int num : nums) {  
            totalSum += num;  
        }  
  
        // 维护左侧元素的和  
        int leftSum = 0;  
  
        // 遍历数组寻找中心索引  
        for (int i = 0; i < nums.length; i++) {  
            // 右侧和 = 总和 - 左侧和 - 当前元素  
            int rightSum = totalSum - leftSum - nums[i];  
            if (leftSum == rightSum) {  
                return i;  
            }  
            leftSum += nums[i];  
        }  
        return -1;  
    }  
}
```

```
// 如果左侧和等于右侧和，返回当前索引
if (leftSum == rightSum) {
    return i;
}

// 更新左侧和，包括当前元素
leftSum += nums[i];
}

// 没有找到中心索引
return -1;
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {1, 7, 3, 6, 5, 6};
    int result1 = pivotIndex(nums1);
    // 预期输出: 3
    System.out.println("测试用例 1: " + result1);

    // 测试用例 2
    int[] nums2 = {1, 2, 3};
    int result2 = pivotIndex(nums2);
    // 预期输出: -1
    System.out.println("测试用例 2: " + result2);

    // 测试用例 3
    int[] nums3 = {2, 1, -1};
    int result3 = pivotIndex(nums3);
    // 预期输出: 0
    System.out.println("测试用例 3: " + result3);

    // 测试用例 4
    int[] nums4 = {};
    int result4 = pivotIndex(nums4);
    // 预期输出: -1
    System.out.println("测试用例 4: " + result4);
}
```

文件: Code13\_FindPivotIndex.py

"""

寻找数组中心索引 (Find Pivot Index)

题目描述:

给你一个整数数组 `nums`, 请编写一个能够返回数组 “中心索引” 的方法。

中心索引是数组的一个索引, 其左侧所有元素相加的和等于右侧所有元素相加的和。

如果数组不存在中心索引, 返回 `-1`。如果数组有多个中心索引, 应该返回最靠近左边的那一个。

注意: 中心索引可能出现在数组的两端。

示例:

输入: `nums = [1, 7, 3, 6, 5, 6]`

输出: 3

解释:

索引 3 (`nums[3] = 6`) 的左侧数之和 ( $1+7+3 = 11$ ), 右侧数之和 ( $5+6 = 11$ ), 二者相等。

输入: `nums = [1, 2, 3]`

输出: -1

解释: 数组中不存在满足此条件的中心索引。

输入: `nums = [2, 1, -1]`

输出: 0

解释:

索引 0 的左侧不存在元素, 视作和为 0; 右侧数之和为  $1 + (-1) = 0$ , 二者相等。

提示:

`nums` 的长度范围为  $[0, 10000]$ 。

任何一个 `nums[i]` 将会是一个范围在  $[-1000, 1000]$  的整数。

题目链接: <https://leetcode.com/problems/find-pivot-index/>

解题思路:

使用前缀和的思想, 计算整个数组的总和, 然后遍历数组, 维护左侧元素的和,

当左侧和等于总和减去左侧和减去当前元素时, 找到中心索引。

时间复杂度:  $O(n)$  – 需要遍历数组两次

空间复杂度:  $O(1)$  – 只使用常数额外空间

"""

```
class Solution:
    def pivotIndex(self, nums):
        """
        寻找数组的中心索引
        """

    Args:
        nums (List[int]): 输入数组

    Returns:
        int: 中心索引，如果不存在返回-1
        """
        # 边界情况处理
        if not nums:
            return -1

        # 计算数组总和
        total_sum = sum(nums)

        # 维护左侧元素的和
        left_sum = 0

        # 遍历数组寻找中心索引
        for i in range(len(nums)):
            # 右侧和 = 总和 - 左侧和 - 当前元素
            right_sum = total_sum - left_sum - nums[i]

            # 如果左侧和等于右侧和，返回当前索引
            if left_sum == right_sum:
                return i

            # 更新左侧和，包括当前元素
            left_sum += nums[i]

        # 没有找到中心索引
        return -1

    # 测试用例
if __name__ == "__main__":
    solution = Solution()

    # 测试用例 1
    nums1 = [1, 7, 3, 6, 5, 6]
```

```
result1 = solution.pivotIndex(nums1)
# 预期输出: 3
print("测试用例 1:", result1)
```

```
# 测试用例 2
nums2 = [1, 2, 3]
result2 = solution.pivotIndex(nums2)
# 预期输出: -1
print("测试用例 2:", result2)
```

```
# 测试用例 3
nums3 = [2, 1, -1]
result3 = solution.pivotIndex(nums3)
# 预期输出: 0
print("测试用例 3:", result3)
```

```
# 测试用例 4
nums4 = []
result4 = solution.pivotIndex(nums4)
# 预期输出: -1
print("测试用例 4:", result4)
```

=====

文件: Code14\_ContinuousSubarraySum.cpp

=====

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

```
/**
 * 连续的子数组和 (Continuous Subarray Sum)
 *
 * 题目描述:
 * 给你一个整数数组 nums 和一个整数 k , 编写一个函数来判断该数组是否含有同时满足下述条件的连续子数组:
 * 1. 子数组大小至少为 2
 * 2. 子数组元素总和为 k 的倍数
 * 如果存在, 返回 true ; 否则, 返回 false .
 *
 * 示例:
 * 输入: nums = [23, 2, 4, 6, 7], k = 6
```

```

* 输出: true
* 解释: [2, 4] 是一个大小为 2 的子数组, 总和为 6 。
*
* 输入: nums = [23, 2, 6, 4, 7], k = 6
* 输出: true
* 解释: [23, 2, 6, 4, 7] 是大小为 5 的子数组, 总和为 42 。42 是 6 的倍数, 因为  $42 = 7 * 6$  且 7 是整数。
*
* 输入: nums = [23, 2, 6, 4, 7], k = 13
* 输出: false
*
* 提示:
*  $1 \leq \text{nums.length} \leq 10^5$ 
*  $0 \leq \text{nums}[i] \leq 10^9$ 
*  $0 \leq \text{sum}(\text{nums}[i]) \leq 2^{31} - 1$ 
*  $1 \leq k \leq 2^{31} - 1$ 
*
* 题目链接: https://leetcode.com/problems/continuous-subarray-sum/
*
* 解题思路:
* 使用前缀和 + 哈希表的方法。
* 1. 遍历数组, 计算前缀和
* 2. 对于当前位置的前缀和, 计算其模 k 的余数
* 3. 如果两个不同位置的前缀和模 k 的余数相同, 说明这两个位置之间的子数组和是 k 的倍数
* 4. 同时需要保证两个位置之间的距离至少为 2
* 5. 特殊情况: k 为 0 的情况需要单独处理
*
* 时间复杂度: O(n) - 需要遍历数组一次
* 空间复杂度: O(n) - 哈希表最多存储 n 个不同的前缀和余数
*/

```

```

class Solution {
public:
    /**
     * 判断数组是否含有满足条件的连续子数组
     *
     * @param nums 输入数组
     * @param k 目标整数
     * @return 是否存在满足条件的子数组
     */
    bool checkSubarraySum(vector<int>& nums, int k) {
        // 边界情况处理
        if (nums.size() < 2) {

```

```
        return false;
    }

// 哈希表记录前缀和余数及其第一次出现的位置
unordered_map<int, int> remainderMap;
// 初始化: 前缀和为 0 在位置-1 出现
remainderMap[0] = -1;

long long prefixSum = 0; // 使用 long long 避免整数溢出

// 遍历数组
for (int i = 0; i < nums.size(); i++) {
    // 更新前缀和
    prefixSum += nums[i];

    // 计算前缀和模 k 的余数
    // 注意: 当 k 为 0 时, 不能进行模运算
    int remainder = 0;
    if (k != 0) {
        remainder = prefixSum % k;
        // 处理负数余数的情况
        if (remainder < 0) {
            remainder += k;
        }
    } else {
        remainder = prefixSum;
    }

    // 如果当前余数之前出现过, 并且两个位置之间的距离至少为 2
    if (remainderMap.find(remainder) != remainderMap.end()) {
        if (i - remainderMap[remainder] >= 2) {
            return true;
        }
    } else {
        // 记录当前余数第一次出现的位置
        remainderMap[remainder] = i;
    }
}

// 没有找到满足条件的子数组
return false;
}
};
```

```

int main() {
    Solution solution;

    // 测试用例 1
    vector<int> nums1 = {23, 2, 4, 6, 7};
    int k1 = 6;
    bool result1 = solution.checkSubarraySum(nums1, k1);
    // 预期输出: true
    cout << "测试用例 1: " << (result1 ? "true" : "false") << endl;

    // 测试用例 2
    vector<int> nums2 = {23, 2, 6, 4, 7};
    int k2 = 6;
    bool result2 = solution.checkSubarraySum(nums2, k2);
    // 预期输出: true
    cout << "测试用例 2: " << (result2 ? "true" : "false") << endl;

    // 测试用例 3
    vector<int> nums3 = {23, 2, 6, 4, 7};
    int k3 = 13;
    bool result3 = solution.checkSubarraySum(nums3, k3);
    // 预期输出: false
    cout << "测试用例 3: " << (result3 ? "true" : "false") << endl;

    // 测试用例 4 - k=0 的情况
    vector<int> nums4 = {0, 0};
    int k4 = 0;
    bool result4 = solution.checkSubarraySum(nums4, k4);
    // 预期输出: true
    cout << "测试用例 4: " << (result4 ? "true" : "false") << endl;

    return 0;
}

```

=====

文件: Code14\_ContinuousSubarraySum.java

=====

```

package class046;

import java.util.HashMap;

```

```
/**  
 * 连续的子数组和 (Continuous Subarray Sum)  
 *  
 * 题目描述:  
 * 给你一个整数数组 nums 和一个整数 k , 编写一个函数来判断该数组是否含有同时满足下述条件的连续子  
 * 数组:  
 * 1. 子数组大小至少为 2  
 * 2. 子数组元素总和为 k 的倍数  
 * 如果存在, 返回 true ; 否则, 返回 false 。  
 *  
 *  
 * 示例:  
 * 输入: nums = [23, 2, 4, 6, 7], k = 6  
 * 输出: true  
 * 解释: [2, 4] 是一个大小为 2 的子数组, 总和为 6 。  
 *  
 * 输入: nums = [23, 2, 6, 4, 7], k = 6  
 * 输出: true  
 * 解释: [23, 2, 6, 4, 7] 是大小为 5 的子数组, 总和为 42 。42 是 6 的倍数, 因为  $42 = 7 * 6$  且 7  
 * 是整数。  
 *  
 * 输入: nums = [23, 2, 6, 4, 7], k = 13  
 * 输出: false  
 *  
 * 提示:  
 *  $1 \leq \text{nums.length} \leq 10^5$   
 *  $0 \leq \text{nums}[i] \leq 10^9$   
 *  $0 \leq \text{sum}(\text{nums}[i]) \leq 2^{31} - 1$   
 *  $1 \leq k \leq 2^{31} - 1$   
 *  
 * 题目链接: https://leetcode.com/problems/continuous-subarray-sum/  
 *  
 * 解题思路:  
 * 使用前缀和 + 哈希表的方法。  
 * 1. 遍历数组, 计算前缀和  
 * 2. 对于当前位置的前缀和, 计算其模 k 的余数  
 * 3. 如果两个不同位置的前缀和模 k 的余数相同, 说明这两个位置之间的子数组和是 k 的倍数  
 * 4. 同时需要保证两个位置之间的距离至少为 2  
 * 5. 特殊情况: k 为 0 的情况需要单独处理  
 *  
 * 时间复杂度: O(n) - 需要遍历数组一次  
 * 空间复杂度: O(n) - 哈希表最多存储 n 个不同的前缀和余数  
 */  
  
public class Code14_ContinuousSubarraySum {
```

```
/**  
 * 判断数组是否含有满足条件的连续子数组  
 *  
 * @param nums 输入数组  
 * @param k 目标整数  
 * @return 是否存在满足条件的子数组  
 */  
public static boolean checkSubarraySum(int[] nums, int k) {  
    // 边界情况处理  
    if (nums == null || nums.length < 2) {  
        return false;  
    }  
  
    // 哈希表记录前缀和余数及其第一次出现的位置  
    HashMap<Integer, Integer> map = new HashMap<>();  
    // 初始化: 前缀和为 0 在位置-1 出现  
    map.put(0, -1);  
  
    int sum = 0; // 当前前缀和  
  
    // 遍历数组  
    for (int i = 0; i < nums.length; i++) {  
        // 更新前缀和  
        sum += nums[i];  
  
        // 计算前缀和模 k 的余数  
        // 注意: 当 k 为 0 时, 不能进行模运算  
        int remainder = k != 0 ? sum % k : sum;  
  
        // 处理负数余数的情况  
        if (remainder < 0 && k != 0) {  
            remainder += k;  
        }  
  
        // 如果当前余数之前出现过, 并且两个位置之间的距离至少为 2  
        if (map.containsKey(remainder)) {  
            if (i - map.get(remainder) >= 2) {  
                return true;  
            }  
        } else {  
            // 记录当前余数第一次出现的位置  
            map.put(remainder, i);  
        }  
    }  
}
```

```
        }
    }

    // 没有找到满足条件的子数组
    return false;
}

/***
 * 测试用例
 */
public static void main(String[] args) {
    // 测试用例 1
    int[] nums1 = {23, 2, 4, 6, 7};
    int k1 = 6;
    boolean result1 = checkSubarraySum(nums1, k1);
    // 预期输出: true
    System.out.println("测试用例 1: " + result1);

    // 测试用例 2
    int[] nums2 = {23, 2, 6, 4, 7};
    int k2 = 6;
    boolean result2 = checkSubarraySum(nums2, k2);
    // 预期输出: true
    System.out.println("测试用例 2: " + result2);

    // 测试用例 3
    int[] nums3 = {23, 2, 6, 4, 7};
    int k3 = 13;
    boolean result3 = checkSubarraySum(nums3, k3);
    // 预期输出: false
    System.out.println("测试用例 3: " + result3);

    // 测试用例 4 - k=0 的情况
    int[] nums4 = {0, 0};
    int k4 = 0;
    boolean result4 = checkSubarraySum(nums4, k4);
    // 预期输出: true
    System.out.println("测试用例 4: " + result4);
}
```

---

文件: Code14\_ContinuousSubarraySum.py

=====

连续的子数组和 (Continuous Subarray Sum)

题目描述:

给你一个整数数组 `nums` 和一个整数 `k`，编写一个函数来判断该数组是否含有同时满足下述条件的连续子数组：

1. 子数组大小至少为 2
2. 子数组元素总和为 `k` 的倍数

如果存在，返回 `true`；否则，返回 `false`。

示例:

输入: `nums = [23, 2, 4, 6, 7]`, `k = 6`

输出: `true`

解释: `[2, 4]` 是一个大小为 2 的子数组，总和为 6。

输入: `nums = [23, 2, 6, 4, 7]`, `k = 6`

输出: `true`

解释: `[23, 2, 6, 4, 7]` 是大小为 5 的子数组，总和为 42。42 是 6 的倍数，因为  $42 = 7 * 6$  且 7 是整数。

输入: `nums = [23, 2, 6, 4, 7]`, `k = 13`

输出: `false`

提示:

`1 <= nums.length <= 10^5`

`0 <= nums[i] <= 10^9`

`0 <= sum(nums[i]) <= 2^31 - 1`

`1 <= k <= 2^31 - 1`

题目链接: <https://leetcode.com/problems/continuous-subarray-sum/>

解题思路:

使用前缀和 + 哈希表的方法。

1. 遍历数组，计算前缀和
2. 对于当前位置的前缀和，计算其模 `k` 的余数
3. 如果两个不同位置的前缀和模 `k` 的余数相同，说明这两个位置之间的子数组和是 `k` 的倍数
4. 同时需要保证两个位置之间的距离至少为 2
5. 特殊情况: `k` 为 0 的情况需要单独处理

时间复杂度:  $O(n)$  – 需要遍历数组一次

空间复杂度:  $O(n)$  – 哈希表最多存储  $n$  个不同的前缀和余数

```
"""
```

```
class Solution:
```

```
    def checkSubarraySum(self, nums, k):
```

```
        """
```

```
        判断数组是否含有满足条件的连续子数组
```

```
    Args:
```

```
        nums (List[int]): 输入数组
```

```
        k (int): 目标整数
```

```
    Returns:
```

```
        bool: 是否存在满足条件的子数组
```

```
        """
```

```
# 边界情况处理
```

```
if not nums or len(nums) < 2:
```

```
    return False
```

```
# 哈希表记录前缀和余数及其第一次出现的位置
```

```
remainder_map = {0: -1}
```

```
# 当前前缀和
```

```
prefix_sum = 0
```

```
# 遍历数组
```

```
for i, num in enumerate(nums):
```

```
    # 更新前缀和
```

```
    prefix_sum += num
```

```
# 计算前缀和模 k 的余数
```

```
# 注意: 当 k 为 0 时, 不能进行模运算
```

```
remainder = prefix_sum % k if k != 0 else prefix_sum
```

```
# 处理负数余数的情况
```

```
if remainder < 0 and k != 0:
```

```
    remainder += k
```

```
# 如果当前余数之前出现过, 并且两个位置之间的距离至少为 2
```

```
if remainder in remainder_map:
```

```
    if i - remainder_map[remainder] >= 2:
```

```
        return True
```

```
else:
```

```
# 记录当前余数第一次出现的位置
```

```

remainder_map[remainder] = i

# 没有找到满足条件的子数组
return False

# 测试用例
if __name__ == "__main__":
    solution = Solution()

# 测试用例 1
nums1 = [23, 2, 4, 6, 7]
k1 = 6
result1 = solution.checkSubarraySum(nums1, k1)
# 预期输出: True
print("测试用例 1:", result1)

# 测试用例 2
nums2 = [23, 2, 6, 4, 7]
k2 = 6
result2 = solution.checkSubarraySum(nums2, k2)
# 预期输出: True
print("测试用例 2:", result2)

# 测试用例 3
nums3 = [23, 2, 6, 4, 7]
k3 = 13
result3 = solution.checkSubarraySum(nums3, k3)
# 预期输出: False
print("测试用例 3:", result3)

# 测试用例 4 - k=0 的情况
nums4 = [0, 0]
k4 = 0
result4 = solution.checkSubarraySum(nums4, k4)
# 预期输出: True
print("测试用例 4:", result4)

```

=====

文件: Code15\_RangeSumQueryImmutable.cpp

=====

```
#include <iostream>
```

```
#include <vector>
#include <stdexcept>
using namespace std;

/***
 * 区间和查询 - 不可变 (Range Sum Query - Immutable)
 *
 * 题目描述:
 * 给定一个整数数组 nums，计算索引 left 和 right（包含 left 和 right）之间的元素的和，其中 left
 * <= right。
 *
 * 实现 NumArray 类:
 * NumArray(int[] nums) 使用数组 nums 初始化对象
 * int sumRange(int left, int right) 返回数组 nums 中索引 left 和 right 之间的元素的总和，包含
 * left 和 right 两点
 *
 * 示例:
 * 输入:
 * ["NumArray", "sumRange", "sumRange", "sumRange"]
 * [[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
 * 输出:
 * [null, 1, -1, -3]
 * 解释:
 * NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
 * numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1
 * numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1
 * numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3
 *
 * 提示:
 * 1 <= nums.length <= 10^4
 * -10^5 <= nums[i] <= 10^5
 * 0 <= left <= right < nums.length
 * 最多调用 10^4 次 sumRange 方法
 *
 * 题目链接: https://leetcode.com/problems/range-sum-query-immutable/
 *
 * 解题思路:
 * 使用前缀和数组预处理原数组，使得每次查询区间的操作时间复杂度为 O(1)。
 * 1. 计算前缀和数组 prefixSum，其中 prefixSum[i] 表示原数组 nums 中前 i 个元素的和
 * 2. 对于区间查询 [left, right]，区间和为 prefixSum[right+1] - prefixSum[left]
 *
 * 时间复杂度:
 * - 初始化: O(n) - 预处理前缀和数组
 * - 查询: O(1) - 直接利用前缀和数组计算区间和
```

\* 空间复杂度: O(n) - 存储前缀和数组

\*/

```
class NumArray {  
private:  
    // 前缀和数组  
    vector<long long> prefixSum; // 使用 long long 避免整数溢出  
  
public:  
    /**  
     * 使用数组 nums 初始化对象，预处理计算前缀和数组  
     *  
     * @param nums 输入数组  
     */  
    NumArray(vector<int>& nums) {  
        // 初始化前缀和数组，长度为 nums.size() + 1  
        // prefixSum[0] = 0 表示前 0 个元素的和为 0  
        // prefixSum[i] 表示前 i 个元素的和，即 nums[0] + nums[1] + ... + nums[i-1]  
        prefixSum.resize(nums.size() + 1, 0);  
  
        // 计算前缀和  
        for (int i = 0; i < nums.size(); i++) {  
            prefixSum[i + 1] = prefixSum[i] + nums[i];  
        }  
    }  
  
    /**  
     * 返回数组 nums 中索引 left 和 right 之间的元素的总和，包含两个端点  
     *  
     * @param left 左边界索引  
     * @param right 右边界索引  
     * @return 区间 [left, right] 的和  
     * @throws invalid_argument 如果索引参数无效  
     */  
    int sumRange(int left, int right) {  
        // 参数合法性检查  
        if (prefixSum.empty()) {  
            throw invalid_argument("数组为空");  
        }  
        if (left < 0 || left >= prefixSum.size() - 1) {  
            throw invalid_argument("左边界索引无效");  
        }  
        if (right < 0 || right >= prefixSum.size() - 1) {  
            throw invalid_argument("右边界索引无效");  
        }  
        return prefixSum[right + 1] - prefixSum[left];  
    }  
}
```

```
        throw invalid_argument("右边界索引无效");
    }

    if (left > right) {
        throw invalid_argument("左边界不能大于右边界");
    }

    // 利用前缀和数组计算区间和
    // [left, right] 的和 = prefixSum[right+1] - prefixSum[left]
    return static_cast<int>(prefixSum[right + 1] - prefixSum[left]);
}

};

int main() {
    try {
        // 创建测试用例数组
        vector<int> nums = {-2, 0, 3, -5, 2, -1};

        // 初始化 NumArray 对象
        NumArray numArray(nums);

        // 测试区间和查询
        // 测试用例 1: [0, 2] 预期输出: 1
        cout << "区间 [0, 2] 的和: " << numArray.sumRange(0, 2) << endl;

        // 测试用例 2: [2, 5] 预期输出: -1
        cout << "区间 [2, 5] 的和: " << numArray.sumRange(2, 5) << endl;

        // 测试用例 3: [0, 5] 预期输出: -3
        cout << "区间 [0, 5] 的和: " << numArray.sumRange(0, 5) << endl;

        // 测试边界情况
        // 测试用例 4: [3, 3] 预期输出: -5
        cout << "区间 [3, 3] 的和: " << numArray.sumRange(3, 3) << endl;

        // 测试空数组
        vector<int> emptyNums;
        NumArray emptyArray(emptyNums);
        // 这行应该抛出异常
        emptyArray.sumRange(0, 0);
    } catch (const invalid_argument& e) {
        cout << "异常捕获: " << e.what() << endl;
    }
}
```

```
    return 0;  
}
```

=====

文件: Code15\_RangeSumQueryImmutable.java

=====

```
package class046;
```

```
/**  
 * 区间和查询 - 不可变 (Range Sum Query - Immutable)  
 *  
 * 题目描述:  
 * 给定一个整数数组 nums，计算索引 left 和 right（包含 left 和 right）之间的元素的和，其中 left  
 * <= right。  
 * 实现 NumArray 类:  
 * NumArray(int[] nums) 使用数组 nums 初始化对象  
 * int sumRange(int left, int right) 返回数组 nums 中索引 left 和 right 之间的元素的总和，包含  
 * left 和 right 两点  
 *  
 * 示例:  
 * 输入:  
 * ["NumArray", "sumRange", "sumRange", "sumRange"]  
 * [[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]  
 * 输出:  
 * [null, 1, -1, -3]  
 * 解释:  
 * NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);  
 * numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1  
 * numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1  
 * numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3  
 *  
 * 提示:  
 * 1 <= nums.length <= 10^4  
 * -10^5 <= nums[i] <= 10^5  
 * 0 <= left <= right < nums.length  
 * 最多调用 10^4 次 sumRange 方法  
 *  
 * 题目链接: https://leetcode.com/problems/range-sum-query-immutable/  
 *  
 * 解题思路:  
 * 使用前缀和数组预处理原数组，使得每次查询区间的操作时间复杂度为 O(1)。  
 * 1. 计算前缀和数组 prefixSum，其中 prefixSum[i] 表示原数组 nums 中前 i 个元素的和
```

```

* 2. 对于区间查询 [left, right]，区间和为 prefixSum[right+1] - prefixSum[left]
*
* 时间复杂度：
* - 初始化: O(n) - 预处理前缀和数组
* - 查询: O(1) - 直接利用前缀和数组计算区间和
* 空间复杂度: O(n) - 存储前缀和数组
*/
public class Code15_RangeSumQueryImmutable {
    // 前缀和数组
    private int[] prefixSum;

    /**
     * 使用数组 nums 初始化对象，预处理计算前缀和数组
     *
     * @param nums 输入数组
     */
    public Code15_RangeSumQueryImmutable(int[] nums) {
        // 边界检查
        if (nums == null || nums.length == 0) {
            prefixSum = new int[0];
            return;
        }

        // 初始化前缀和数组，长度为 nums.length + 1
        // prefixSum[0] = 0 表示前 0 个元素的和为 0
        // prefixSum[i] 表示前 i 个元素的和，即 nums[0] + nums[1] + ... + nums[i-1]
        prefixSum = new int[nums.length + 1];

        // 计算前缀和
        for (int i = 0; i < nums.length; i++) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }
    }

    /**
     * 返回数组 nums 中索引 left 和 right 之间的元素的总和，包含两个端点
     *
     * @param left 左边界索引
     * @param right 右边界索引
     * @return 区间 [left, right] 的和
     * @throws IllegalArgumentException 如果索引参数无效
     */
    public int sumRange(int left, int right) {

```

```
// 参数合法性检查
if (prefixSum.length == 0) {
    throw new IllegalArgumentException("数组为空");
}
if (left < 0 || left >= prefixSum.length - 1) {
    throw new IllegalArgumentException("左边界索引无效: " + left);
}
if (right < 0 || right >= prefixSum.length - 1) {
    throw new IllegalArgumentException("右边界索引无效: " + right);
}
if (left > right) {
    throw new IllegalArgumentException("左边界不能大于右边界: " + left + " > " + right);
}

// 利用前缀和数组计算区间和
// [left, right] 的和 = prefixSum[right+1] - prefixSum[left]
return prefixSum[right + 1] - prefixSum[left];
}

/**
 * 测试用例
 */
public static void main(String[] args) {
    // 创建测试用例数组
    int[] nums = {-2, 0, 3, -5, 2, -1};

    // 初始化 NumArray 对象
    Code15_RangeSumQueryImmutable numArray = new Code15_RangeSumQueryImmutable(nums);

    // 测试区间和查询
    // 测试用例 1: [0, 2] 预期输出: 1
    System.out.println("区间 [0, 2] 的和: " + numArray.sumRange(0, 2));

    // 测试用例 2: [2, 5] 预期输出: -1
    System.out.println("区间 [2, 5] 的和: " + numArray.sumRange(2, 5));

    // 测试用例 3: [0, 5] 预期输出: -3
    System.out.println("区间 [0, 5] 的和: " + numArray.sumRange(0, 5));

    // 测试边界情况
    // 测试用例 4: [3, 3] 预期输出: -5
    System.out.println("区间 [3, 3] 的和: " + numArray.sumRange(3, 3));
}
```

}

=====

文件: Code15\_RangeSumQueryImmutable.py

=====

"""

区间和查询 - 不可变 (Range Sum Query - Immutable)

题目描述:

给定一个整数数组 `nums`, 计算索引 `left` 和 `right` (包含 `left` 和 `right`) 之间的元素的和, 其中 `left <= right`。

实现 `NumArray` 类:

`NumArray(int[] nums)` 使用数组 `nums` 初始化对象

`int sumRange(int left, int right)` 返回数组 `nums` 中索引 `left` 和 `right` 之间的元素的总和, 包含 `left` 和 `right` 两点

示例:

输入:

```
[“NumArray”, “sumRange”, “sumRange”, “sumRange”]  
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```

输出:

```
[null, 1, -1, -3]
```

解释:

```
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);  
numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1  
numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1  
numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3
```

提示:

$1 \leq \text{nums.length} \leq 10^4$

$-10^5 \leq \text{nums}[i] \leq 10^5$

$0 \leq \text{left} \leq \text{right} < \text{nums.length}$

最多调用  $10^4$  次 `sumRange` 方法

题目链接: <https://leetcode.com/problems/range-sum-query-immutable/>

解题思路:

使用前缀和数组预处理原数组, 使得每次查询区间和的操作时间复杂度为  $O(1)$ 。

1. 计算前缀和数组 `prefixSum`, 其中 `prefixSum[i]` 表示原数组 `nums` 中前 `i` 个元素的和
2. 对于区间查询 `[left, right]`, 区间和为 `prefixSum[right+1] - prefixSum[left]`

时间复杂度:

- 初始化:  $O(n)$  - 预处理前缀和数组
  - 查询:  $O(1)$  - 直接利用前缀和数组计算区间和
- 空间复杂度:  $O(n)$  - 存储前缀和数组
- """

```
class NumArray:  
    def __init__(self, nums):  
        """  
        使用数组 nums 初始化对象，预处理计算前缀和数组  
  
        Args:  
            nums (List[int]): 输入数组  
        """  
        # 边界检查  
        if not nums:  
            self.prefix_sum = []  
            return  
  
        # 初始化前缀和数组，长度为 len(nums) + 1  
        # prefix_sum[0] = 0 表示前 0 个元素的和为 0  
        # prefix_sum[i] 表示前 i 个元素的和，即 nums[0] + nums[1] + ... + nums[i-1]  
        self.prefix_sum = [0] * (len(nums) + 1)  
  
        # 计算前缀和  
        for i in range(len(nums)):  
            self.prefix_sum[i + 1] = self.prefix_sum[i] + nums[i]  
  
    def sumRange(self, left, right):  
        """  
        返回数组 nums 中索引 left 和 right 之间的元素的总和，包含两个端点  
  
        Args:  
            left (int): 左边界索引  
            right (int): 右边界索引  
  
        Returns:  
            int: 区间 [left, right] 的和  
  
        Raises:  
            ValueError: 如果索引参数无效  
        """  
        # 参数合法性检查  
        if not self.prefix_sum:
```

```
    raise ValueError("数组为空")
    if left < 0 or left >= len(self.prefix_sum) - 1:
        raise ValueError(f"左边界索引无效: {left}")
    if right < 0 or right >= len(self.prefix_sum) - 1:
        raise ValueError(f"右边界索引无效: {right}")
    if left > right:
        raise ValueError(f"左边界不能大于右边界: {left} > {right}")

# 利用前缀和数组计算区间和
# [left, right] 的和 = prefix_sum[right+1] - prefix_sum[left]
return self.prefix_sum[right + 1] - self.prefix_sum[left]

# 测试用例
if __name__ == "__main__":
    # 创建测试用例数组
    nums = [-2, 0, 3, -5, 2, -1]

    # 初始化 NumArray 对象
    num_array = NumArray(nums)

    # 测试区间和查询
    # 测试用例 1: [0, 2] 预期输出: 1
    print(f"区间 [0, 2] 的和: {num_array.sumRange(0, 2)}")

    # 测试用例 2: [2, 5] 预期输出: -1
    print(f"区间 [2, 5] 的和: {num_array.sumRange(2, 5)}")

    # 测试用例 3: [0, 5] 预期输出: -3
    print(f"区间 [0, 5] 的和: {num_array.sumRange(0, 5)}")

    # 测试边界情况
    # 测试用例 4: [3, 3] 预期输出: -5
    print(f"区间 [3, 3] 的和: {num_array.sumRange(3, 3)}")

    # 测试空数组
    try:
        empty_array = NumArray([])
        empty_array.sumRange(0, 0)
    except ValueError as e:
        print(f"空数组测试: {e}")

=====
```

文件: Code16\_RangeSumQuery2DImmutable.cpp

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

/***
 * 二维区域和检索 - 不可变 (Range Sum Query 2D - Immutable)
 *
 * 题目描述:
 * 给定一个二维矩阵 matrix，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1)，右下角为 (row2, col2)。
 * 实现 NumMatrix 类:
 * NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化
 * int sumRegion(int row1, int col1, int row2, int col2) 返回左上角 (row1, col1)、右下角 (row2, col2) 的子矩阵的元素总和。
 *
 * 示例:
 * 输入:
 * ["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
 * [[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]
 * 输出:
 * [null, 8, 11, 12]
 * 解释:
 * NumMatrix numMatrix = new NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]);
 * numMatrix.sumRegion(2, 1, 4, 3); // return 8 (红色矩形框的元素总和)
 * numMatrix.sumRegion(1, 1, 2, 2); // return 11 (绿色矩形框的元素总和)
 * numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩形框的元素总和)
 *
 * 提示:
 * m == matrix.length
 * n == matrix[i].length
 * 1 <= m, n <= 200
 * -10^5 <= matrix[i][j] <= 10^5
 * 0 <= row1 <= row2 < m
 * 0 <= col1 <= col2 < n
 * 最多调用 10^4 次 sumRegion 方法
 *
 * 题目链接: https://leetcode.com/problems/range-sum-query-2d-immutable/
```

```

*
* 解题思路:
* 使用二维前缀和数组预处理原矩阵, 使得每次查询子矩阵和的操作时间复杂度为 O(1)。
* 1. 计算二维前缀和数组 prefixSum, 其中 prefixSum[i][j] 表示从左上角 (0, 0) 到右下角 (i-1, j-1) 的子矩阵元素和
* 2. 对于子矩阵查询 (row1, col1) 到 (row2, col2), 可以利用容斥原理计算:
*   prefixSum[row2+1][col2+1] - prefixSum[row1][col2+1] - prefixSum[row2+1][col1] +
prefixSum[row1][col1]
*
* 时间复杂度:
* - 初始化: O(m*n) - 预处理二维前缀和数组
* - 查询: O(1) - 直接利用二维前缀和数组计算子矩阵和
* 空间复杂度: O(m*n) - 存储二维前缀和数组
*/

```

```

class NumMatrix {
private:
    // 二维前缀和数组
    vector<vector<long long>> prefixSum; // 使用 long long 避免整数溢出

public:
    /**
     * 使用二维矩阵 matrix 初始化对象, 预处理计算二维前缀和数组
     *
     * @param matrix 输入的二维矩阵
     */
    NumMatrix(vector<vector<int>>& matrix) {
        // 边界检查
        if (matrix.empty() || matrix[0].empty()) {
            return;
        }

        int m = matrix.size(); // 矩阵的行数
        int n = matrix[0].size(); // 矩阵的列数

        // 初始化前缀和数组, 大小为 (m+1) × (n+1)
        // prefixSum[0][*] 和 prefixSum[*][0] 都是边界, 值为 0
        // prefixSum[i][j] 表示从 (0, 0) 到 (i-1, j-1) 的子矩阵元素和
        prefixSum.resize(m + 1, vector<long long>(n + 1, 0));

        // 计算二维前缀和
        for (int i = 0; i < m; i++) {
            long long rowSum = 0; // 用于优化计算, 记录当前行的累加和

```

```

        for (int j = 0; j < n; j++) {
            // 当前行的累加和
            rowSum += matrix[i][j];
            // 前缀和计算: 当前行累加和 + 上方子矩阵和
            prefixSum[i + 1][j + 1] = rowSum + prefixSum[i][j + 1];
        }
    }
}

/***
 * 返回左上角 (row1, col1) 、右下角 (row2, col2) 的子矩阵的元素总和
 *
 * @param row1 左上角行索引
 * @param col1 左上角列索引
 * @param row2 右下角行索引
 * @param col2 右下角列索引
 * @return 子矩阵 [row1, row2] × [col1, col2] 的元素和
 * @throws invalid_argument 如果索引参数无效
 */
int sumRegion(int row1, int col1, int row2, int col2) {
    // 参数合法性检查
    if (prefixSum.empty() || prefixSum[0].empty()) {
        throw invalid_argument("矩阵为空");
    }

    int m = prefixSum.size() - 1;
    int n = prefixSum[0].size() - 1;

    // 检查行索引是否有效
    if (row1 < 0 || row1 >= m || row2 < 0 || row2 >= m) {
        throw invalid_argument("行索引无效");
    }

    // 检查列索引是否有效
    if (col1 < 0 || col1 >= n || col2 < 0 || col2 >= n) {
        throw invalid_argument("列索引无效");
    }

    // 检查索引范围是否有效
    if (row1 > row2 || col1 > col2) {
        throw invalid_argument("无效的索引范围");
    }

    // 利用容斥原理计算子矩阵和
    // 区域和 = 右下角前缀和 - 左侧前缀和 - 上方前缀和 + 左上角重叠区域前缀和
}

```

```

        return static_cast<int>(
            prefixSum[row2 + 1][col2 + 1] -
            prefixSum[row1][col2 + 1] -
            prefixSum[row2 + 1][col1] +
            prefixSum[row1][col1]
        );
    }
};

int main() {
    try {
        // 创建测试用例矩阵
        vector<vector<int>> matrix = {
            {3, 0, 1, 4, 2},
            {5, 6, 3, 2, 1},
            {1, 2, 0, 1, 5},
            {4, 1, 0, 1, 7},
            {1, 0, 3, 0, 5}
        };
    }

    // 初始化 NumMatrix 对象
    NumMatrix numMatrix(matrix);

    // 测试子矩阵和查询
    // 测试用例 1: (2, 1, 4, 3) 预期输出: 8
    cout << "子矩阵 [2,1] 到 [4,3] 的和: " << numMatrix.sumRegion(2, 1, 4, 3) << endl;

    // 测试用例 2: (1, 1, 2, 2) 预期输出: 11
    cout << "子矩阵 [1,1] 到 [2,2] 的和: " << numMatrix.sumRegion(1, 1, 2, 2) << endl;

    // 测试用例 3: (1, 2, 2, 4) 预期输出: 12
    cout << "子矩阵 [1,2] 到 [2,4] 的和: " << numMatrix.sumRegion(1, 2, 2, 4) << endl;

    // 测试边界情况
    // 测试用例 4: (0, 0, 0, 0) 预期输出: 3
    cout << "子矩阵 [0,0] 到 [0,0] 的和: " << numMatrix.sumRegion(0, 0, 0, 0) << endl;

    // 测试用例 5: (0, 0, 4, 4) 预期输出: 所有元素的和 48
    cout << "子矩阵 [0,0] 到 [4,4] 的和: " << numMatrix.sumRegion(0, 0, 4, 4) << endl;

    // 测试空矩阵
    vector<vector<int>> emptyMatrix;
    NumMatrix emptyNumMatrix(emptyMatrix);
}

```

```

    // 这行应该抛出异常
    emptyNumMatrix.sumRegion(0, 0, 0, 0);
} catch (const invalid_argument& e) {
    cout << "异常捕获: " << e.what() << endl;
}

return 0;
}
=====
```

文件: Code16\_RangeSumQuery2DImmutable.java

```
=====
package class046;
```

```

/**
 * 二维区域和检索 - 不可变 (Range Sum Query 2D - Immutable)
 *
 * 题目描述:
 * 给定一个二维矩阵 matrix，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1)，右下角为 (row2, col2)。
 * 实现 NumMatrix 类:
 * NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化
 * int sumRegion(int row1, int col1, int row2, int col2) 返回左上角 (row1, col1)、右下角 (row2, col2) 的子矩阵的元素总和。
 *
 * 示例:
 * 输入:
 * ["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
 * [[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]
 * 输出:
 * [null, 8, 11, 12]
 * 解释:
 * NumMatrix numMatrix = new NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]);
 * numMatrix.sumRegion(2, 1, 4, 3); // return 8 (红色矩形框的元素总和)
 * numMatrix.sumRegion(1, 1, 2, 2); // return 11 (绿色矩形框的元素总和)
 * numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩形框的元素总和)
 *
 * 提示:
 * m == matrix.length
 * n == matrix[i].length

```

```

* 1 <= m, n <= 200
* -10^5 <= matrix[i][j] <= 10^5
* 0 <= row1 <= row2 < m
* 0 <= col1 <= col2 < n
* 最多调用 10^4 次 sumRegion 方法
*
* 题目链接: https://leetcode.com/problems/range-sum-query-2d-immutable/
*
* 解题思路:
* 使用二维前缀和数组预处理原矩阵，使得每次查询子矩阵和的操作时间复杂度为 O(1)。
* 1. 计算二维前缀和数组 prefixSum，其中 prefixSum[i][j] 表示从左上角 (0, 0) 到右下角 (i-1, j-1) 的子矩阵元素和
* 2. 对于子矩阵查询 (row1, col1) 到 (row2, col2)，可以利用容斥原理计算：
*   prefixSum[row2+1][col2+1] - prefixSum[row1][col2+1] - prefixSum[row2+1][col1] +
prefixSum[row1][col1]
*
* 时间复杂度：
* - 初始化: O(m*n) - 预处理二维前缀和数组
* - 查询: O(1) - 直接利用二维前缀和数组计算子矩阵和
* 空间复杂度: O(m*n) - 存储二维前缀和数组
*/
public class Code16_RangeSumQuery2DImmutable {
    // 二维前缀和数组
    private int[][] prefixSum;

    /**
     * 使用二维矩阵 matrix 初始化对象，预处理计算二维前缀和数组
     *
     * @param matrix 输入的二维矩阵
     */
    public Code16_RangeSumQuery2DImmutable(int[][] matrix) {
        // 边界检查
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            prefixSum = new int[0][0];
            return;
        }

        int m = matrix.length;      // 矩阵的行数
        int n = matrix[0].length;  // 矩阵的列数

        // 初始化前缀和数组，大小为 (m+1) × (n+1)
        // prefixSum[0][*] 和 prefixSum[*][0] 都是边界，值为 0
        // prefixSum[i][j] 表示从 (0, 0) 到 (i-1, j-1) 的子矩阵元素和
    }
}

```

```

prefixSum = new int[m + 1][n + 1];

// 计算二维前缀和
for (int i = 0; i < m; i++) {
    int rowSum = 0; // 用于优化计算，记录当前行的累加和
    for (int j = 0; j < n; j++) {
        // 当前行的累加和
        rowSum += matrix[i][j];
        // 前缀和计算：当前行累加和 + 上方子矩阵和 - 左上角重复计算的部分
        prefixSum[i + 1][j + 1] = rowSum + prefixSum[i][j + 1];
    }
}

/**
 * 返回左上角 (row1, col1) 、右下角 (row2, col2) 的子矩阵的元素总和
 *
 * @param row1 左上角行索引
 * @param col1 左上角列索引
 * @param row2 右下角行索引
 * @param col2 右下角列索引
 * @return 子矩阵 [row1, row2] × [col1, col2] 的元素和
 * @throws IllegalArgumentException 如果索引参数无效
 */
public int sumRegion(int row1, int col1, int row2, int col2) {
    // 参数合法性检查
    if (prefixSum.length == 0 || prefixSum[0].length == 0) {
        throw new IllegalArgumentException("矩阵为空");
    }

    int m = prefixSum.length - 1;
    int n = prefixSum[0].length - 1;

    // 检查行索引是否有效
    if (row1 < 0 || row1 >= m || row2 < 0 || row2 >= m) {
        throw new IllegalArgumentException("行索引无效");
    }

    // 检查列索引是否有效
    if (col1 < 0 || col1 >= n || col2 < 0 || col2 >= n) {
        throw new IllegalArgumentException("列索引无效");
    }

    // 检查索引范围是否有效
    if (row1 > row2 || col1 > col2) {

```

```
        throw new IllegalArgumentException("无效的索引范围");
    }

    // 利用容斥原理计算子矩阵和
    // 区域和 = 右下角前缀和 - 左侧前缀和 - 上方前缀和 + 左上角重叠区域前缀和
    return prefixSum[row2 + 1][col2 + 1] - prefixSum[row1][col2 + 1] -
        prefixSum[row2 + 1][col1] + prefixSum[row1][col1];
}

/***
 * 测试用例
 */
public static void main(String[] args) {
    // 创建测试用例矩阵
    int[][] matrix = {
        {3, 0, 1, 4, 2},
        {5, 6, 3, 2, 1},
        {1, 2, 0, 1, 5},
        {4, 1, 0, 1, 7},
        {1, 0, 3, 0, 5}
    };

    // 初始化 NumMatrix 对象
    Code16_RangeSumQuery2DImmutable numMatrix = new Code16_RangeSumQuery2DImmutable(matrix);

    // 测试子矩阵和查询
    // 测试用例 1: (2, 1, 4, 3) 预期输出: 8
    System.out.println("子矩阵 [2,1] 到 [4,3] 的和: " + numMatrix.sumRegion(2, 1, 4, 3));

    // 测试用例 2: (1, 1, 2, 2) 预期输出: 11
    System.out.println("子矩阵 [1,1] 到 [2,2] 的和: " + numMatrix.sumRegion(1, 1, 2, 2));

    // 测试用例 3: (1, 2, 2, 4) 预期输出: 12
    System.out.println("子矩阵 [1,2] 到 [2,4] 的和: " + numMatrix.sumRegion(1, 2, 2, 4));

    // 测试边界情况
    // 测试用例 4: (0, 0, 0, 0) 预期输出: 3
    System.out.println("子矩阵 [0,0] 到 [0,0] 的和: " + numMatrix.sumRegion(0, 0, 0, 0));

    // 测试用例 5: (0, 0, 4, 4) 预期输出: 所有元素的和 48
    System.out.println("子矩阵 [0,0] 到 [4,4] 的和: " + numMatrix.sumRegion(0, 0, 4, 4));
}
```

=====

文件: Code16\_RangeSumQuery2DImmutable.py

=====

"""

二维区域和检索 - 不可变 (Range Sum Query 2D - Immutable)

题目描述:

给定一个二维矩阵 matrix，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1)，右下角为 (row2, col2)。

实现 NumMatrix 类:

NumMatrix(int[][] matrix) 给定整数矩阵 matrix 进行初始化

int sumRegion(int row1, int col1, int row2, int col2) 返回左上角 (row1, col1)、右下角 (row2, col2) 的子矩阵的元素总和。

示例:

输入:

```
["NumMatrix", "sumRegion", "sumRegion", "sumRegion"]
[[[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]], [2, 1, 4, 3], [1, 1, 2, 2], [1, 2, 2, 4]]]
```

输出:

```
[null, 8, 11, 12]
```

解释:

```
NumMatrix numMatrix = new NumMatrix([[3, 0, 1, 4, 2], [5, 6, 3, 2, 1], [1, 2, 0, 1, 5], [4, 1, 0, 1, 7], [1, 0, 3, 0, 5]]);
numMatrix.sumRegion(2, 1, 4, 3); // return 8 (红色矩形框的元素总和)
numMatrix.sumRegion(1, 1, 2, 2); // return 11 (绿色矩形框的元素总和)
numMatrix.sumRegion(1, 2, 2, 4); // return 12 (蓝色矩形框的元素总和)
```

提示:

```
m == matrix.length
n == matrix[i].length
1 <= m, n <= 200
-10^5 <= matrix[i][j] <= 10^5
0 <= row1 <= row2 < m
0 <= col1 <= col2 < n
```

最多调用  $10^4$  次 sumRegion 方法

题目链接: <https://leetcode.com/problems/range-sum-query-2d-immutable/>

解题思路:

使用二维前缀和数组预处理原矩阵，使得每次查询子矩阵和的操作时间复杂度为  $O(1)$ 。

1. 计算二维前缀和数组 `prefix_sum`, 其中 `prefix_sum[i][j]` 表示从左上角  $(0, 0)$  到右下角  $(i-1, j-1)$  的子矩阵元素和

2. 对于子矩阵查询  $(\text{row1}, \text{col1})$  到  $(\text{row2}, \text{col2})$ , 可以利用容斥原理计算:

```
prefix_sum[row2+1][col2+1] - prefix_sum[row1][col2+1] - prefix_sum[row2+1][col1] +
prefix_sum[row1][col1]
```

时间复杂度:

- 初始化:  $O(m*n)$  - 预处理二维前缀和数组
- 查询:  $O(1)$  - 直接利用二维前缀和数组计算子矩阵和

空间复杂度:  $O(m*n)$  - 存储二维前缀和数组

"""

```
class NumMatrix:
```

```
    def __init__(self, matrix):
```

"""

使用二维矩阵 `matrix` 初始化对象, 预处理计算二维前缀和数组

Args:

```
    matrix (List[List[int]]): 输入的二维矩阵
```

"""

# 边界检查

```
    if not matrix or not matrix[0]:
```

```
        self.prefix_sum = []
```

```
        return
```

```
    m = len(matrix)      # 矩阵的行数
```

```
    n = len(matrix[0])   # 矩阵的列数
```

# 初始化前缀和数组, 大小为  $(m+1) \times (n+1)$

# `prefix_sum[0][*]` 和 `prefix_sum[*][0]` 都是边界, 值为 0

# `prefix_sum[i][j]` 表示从  $(0, 0)$  到  $(i-1, j-1)$  的子矩阵元素和

```
    self.prefix_sum = [[0] * (n + 1) for _ in range(m + 1)]
```

# 计算二维前缀和

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

```
        row_sum = 0 # 用于优化计算, 记录当前行的累加和
```

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

# 当前行的累加和

```
        row_sum += matrix[i][j]
```

# 前缀和计算: 当前行累加和 + 上方子矩阵和 - 左上角重复计算的部分

```
        self.prefix_sum[i + 1][j + 1] = row_sum + self.prefix_sum[i][j + 1]
```

```
def sumRegion(self, row1, col1, row2, col2):
```

"""

返回左上角 (row1, col1) 、右下角 (row2, col2) 的子矩阵的元素总和

Args:

row1 (int): 左上角行索引  
col1 (int): 左上角列索引  
row2 (int): 右下角行索引  
col2 (int): 右下角列索引

Returns:

int: 子矩阵 [row1, row2] × [col1, col2] 的元素和

Raises:

ValueError: 如果索引参数无效

"""

# 参数合法性检查

```
if not self.prefix_sum:  
    raise ValueError("矩阵为空")
```

m = len(self.prefix\_sum) - 1

n = len(self.prefix\_sum[0]) - 1

# 检查行索引是否有效

```
if row1 < 0 or row1 >= m or row2 < 0 or row2 >= m:  
    raise ValueError("行索引无效")
```

# 检查列索引是否有效

```
if col1 < 0 or col1 >= n or col2 < 0 or col2 >= n:  
    raise ValueError("列索引无效")
```

# 检查索引范围是否有效

```
if row1 > row2 or col1 > col2:  
    raise ValueError("无效的索引范围")
```

# 利用容斥原理计算子矩阵和

# 区域和 = 右下角前缀和 - 左侧前缀和 - 上方前缀和 + 左上角重叠区域前缀和

return (

```
    self.prefix_sum[row2 + 1][col2 + 1]  
    - self.prefix_sum[row1][col2 + 1]  
    - self.prefix_sum[row2 + 1][col1]  
    + self.prefix_sum[row1][col1]
```

)

# 测试用例

```

if __name__ == "__main__":
    # 创建测试用例矩阵
    matrix = [
        [3, 0, 1, 4, 2],
        [5, 6, 3, 2, 1],
        [1, 2, 0, 1, 5],
        [4, 1, 0, 1, 7],
        [1, 0, 3, 0, 5]
    ]

    # 初始化 NumMatrix 对象
    num_matrix = NumMatrix(matrix)

    # 测试子矩阵和查询
    # 测试用例 1: (2, 1, 4, 3) 预期输出: 8
    print(f"子矩阵 [2,1] 到 [4,3] 的和: {num_matrix.sumRegion(2, 1, 4, 3)}")

    # 测试用例 2: (1, 1, 2, 2) 预期输出: 11
    print(f"子矩阵 [1,1] 到 [2,2] 的和: {num_matrix.sumRegion(1, 1, 2, 2)}")

    # 测试用例 3: (1, 2, 2, 4) 预期输出: 12
    print(f"子矩阵 [1,2] 到 [2,4] 的和: {num_matrix.sumRegion(1, 2, 2, 4)}")

    # 测试边界情况
    # 测试用例 4: (0, 0, 0, 0) 预期输出: 3
    print(f"子矩阵 [0,0] 到 [0,0] 的和: {num_matrix.sumRegion(0, 0, 0, 0)}")

    # 测试用例 5: (0, 0, 4, 4) 预期输出: 所有元素的和 48
    print(f"子矩阵 [0,0] 到 [4,4] 的和: {num_matrix.sumRegion(0, 0, 4, 4)}")

    # 测试空矩阵
    try:
        empty_matrix = NumMatrix([])
        empty_matrix.sumRegion(0, 0, 0, 0)
    except ValueError as e:
        print(f"空矩阵测试: {e}")

```

=====

文件: Code22\_PrefixSumQueries.cpp

=====

```

// Prefix Sum Queries (CSES 2166)
//
```

```
// 题目描述:  
// 给定一个数组，支持两种操作：  
// 1. 更新位置 k 的值为 u  
// 2. 查询区间 [a, b] 内的最大前缀和  
  
//  
// 示例：  
// 输入：  
// 5 3  
// 2 2 2 2 2  
// 2 1 5  
// 1 3 4  
// 2 1 5  
// 输出：  
// 10  
// 12  
  
// 提示：  
// 1 <= n, q <= 2 * 10^5  
// -10^9 <= x <= 10^9  
  
// 题目链接：https://cses.fi/problemset/task/2166  
  
// 解题思路：  
// 使用线段树维护区间信息，每个节点存储：  
// 1. 区间和  
// 2. 区间最大前缀和  
// 3. 区间最大后缀和  
// 4. 区间最大子段和  
  
// 时间复杂度：  
// - 初始化：O(n) - 需要遍历整个数组构建线段树  
// - 更新：O(log n) - 每次更新操作的时间复杂度  
// - 查询：O(log n) - 每次查询操作的时间复杂度  
// 空间复杂度：O(n) - 线段树需要额外的空间  
  
// 工程化考量：  
// 1. 边界条件处理：空数组、单元素数组  
// 2. 性能优化：使用线段树提供高效的区间查询和更新  
// 3. 数据结构选择：线段树适合频繁的区间操作  
// 4. 大数处理：元素值可能很大，需要确保整数范围  
  
// 最优解分析：  
// 这是最优解，因为需要支持动态更新和查询操作，线段树提供了 O(log n) 的时间复杂度。
```

```
// 对于频繁的区间操作，线段树是最佳选择。  
//  
// 算法核心：  
// 线段树的合并操作：  
// - 区间和 = 左子树区间和 + 右子树区间和  
// - 区间最大前缀和 = max(左子树最大前缀和, 左子树区间和 + 右子树最大前缀和)  
// - 区间最大后缀和 = max(右子树最大后缀和, 右子树区间和 + 左子树最大后缀和)  
// - 区间最大子段和 = max(左子树最大子段和, 右子树最大子段和, 左子树最大后缀和 + 右子树最大前缀和)  
//  
// 算法调试技巧：  
// 1. 打印中间过程：显示线段树的构建和更新过程  
// 2. 边界测试：测试空数组、单元素数组等特殊情况  
// 3. 性能测试：测试大规模数组下的性能表现  
//  
// 语言特性差异：  
// C++中数组需要手动管理内存。  
// 与 Java 相比，C++需要手动释放内存。  
// 与 Python 相比，C++是静态类型语言，需要显式声明类型。
```

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <climits>  
using namespace std;  
  
// 线段树节点结构  
struct SegmentTreeNode {  
    long long sum;           // 区间和  
    long long maxPrefixSum; // 区间最大前缀和  
    long long maxSuffixSum; // 区间最大后缀和  
    long long maxSubarraySum; // 区间最大子段和  
  
    SegmentTreeNode() : sum(0), maxPrefixSum(0), maxSuffixSum(0), maxSubarraySum(0) {}  
};  
  
class SegmentTree {  
private:  
    vector<SegmentTreeNode> tree;  
    vector<long long> arr;  
    int n;  
  
    // 构建线段树
```

```

void build(int node, int start, int end) {
    // 叶子节点
    if (start == end) {
        tree[node].sum = arr[start];
        tree[node].maxPrefixSum = arr[start];
        tree[node].maxSuffixSum = arr[start];
        tree[node].maxSubarraySum = arr[start];
        return;
    }

    int mid = (start + end) / 2;
    // 递归构建左右子树
    build(2 * node, start, mid);
    build(2 * node + 1, mid + 1, end);

    // 合并左右子树的信息
    merge(node);
}

// 合并左右子树的信息
void merge(int node) {
    int leftChild = 2 * node;
    int rightChild = 2 * node + 1;

    // 区间和 = 左子树区间和 + 右子树区间和
    tree[node].sum = tree[leftChild].sum + tree[rightChild].sum;

    // 区间最大前缀和 = max(左子树最大前缀和, 左子树区间和 + 右子树最大前缀和)
    tree[node].maxPrefixSum = max(
        tree[leftChild].maxPrefixSum,
        tree[leftChild].sum + tree[rightChild].maxPrefixSum
    );

    // 区间最大后缀和 = max(右子树最大后缀和, 右子树区间和 + 左子树最大后缀和)
    tree[node].maxSuffixSum = max(
        tree[rightChild].maxSuffixSum,
        tree[rightChild].sum + tree[leftChild].maxSuffixSum
    );

    // 区间最大子段和 = max(左子树最大子段和, 右子树最大子段和, 左子树最大后缀和 + 右子树最大
    // 前缀和)
    tree[node].maxSubarraySum = max(
        max(tree[leftChild].maxSubarraySum, tree[rightChild].maxSubarraySum),

```

```

        tree[leftChild].maxSuffixSum + tree[rightChild].maxPrefixSum
    );
}

// 更新线段树中的值
void update(int node, int start, int end, int index, long long value) {
    // 叶子节点
    if (start == end) {
        tree[node].sum = value;
        tree[node].maxPrefixSum = value;
        tree[node].maxSuffixSum = value;
        tree[node].maxSubarraySum = value;
        return;
    }

    int mid = (start + end) / 2;
    // 根据索引决定更新左子树还是右子树
    if (index <= mid) {
        update(2 * node, start, mid, index, value);
    } else {
        update(2 * node + 1, mid + 1, end, index, value);
    }

    // 更新后重新合并信息
    merge(node);
}

// 查询区间[left, right]内的最大前缀和
long long queryMaxPrefixSum(int node, int start, int end, int left, int right) {
    // 完全不在查询区间内
    if (start > right || end < left) {
        return LLONG_MIN;
    }

    // 完全在查询区间内
    if (start >= left && end <= right) {
        return tree[node].maxPrefixSum;
    }

    int mid = (start + end) / 2;
    // 递归查询左右子树
    long long leftResult = queryMaxPrefixSum(2 * node, start, mid, left, right);
    long long rightResult = queryMaxPrefixSum(2 * node + 1, mid + 1, end, left, right);
}

```

```

    // 返回较大值
    return max(leftResult, rightResult);
}

public:
    // 构造函数, 初始化线段树
    SegmentTree(const vector<long long>& array) {
        this->n = array.size();
        this->arr = array;
        // 线段树数组大小通常为 4*n
        this->tree.resize(4 * n);
        // 构建线段树
        build(1, 0, n - 1);
    }

    // 更新数组中某个位置的值
    void update(int index, long long value) {
        arr[index] = value;
        update(1, 0, n - 1, index, value);
    }

    // 查询区间[0, end]内的最大前缀和
    long long queryMaxPrefixSum(int end) {
        return queryMaxPrefixSum(1, 0, n - 1, 0, end);
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, q;
    cin >> n >> q;

    vector<long long> arr(n);
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    SegmentTree segTree(arr);

    for (int i = 0; i < q; i++) {

```

```

int type;
cin >> type;

if (type == 1) {
    // 更新操作
    int k;
    long long u;
    cin >> k >> u;
    segTree.update(k - 1, u); // 转换为 0-based 索引
} else {
    // 查询操作
    int a, b;
    cin >> a >> b;
    long long result = segTree.queryMaxPrefixSum(b - 1); // 转换为 0-based 索引
    cout << result << "\n";
}
}

return 0;
}

```

=====

文件: Code22\_PrefixSumQueries.java

=====

```

package class046;

/**
 * Prefix Sum Queries (CSES 2166)
 *
 * 题目描述:
 * 给定一个数组, 支持两种操作:
 * 1. 更新位置 k 的值为 u
 * 2. 查询区间 [a, b] 内的最大前缀和
 *
 * 示例:
 * 输入:
 * 5 3
 * 2 2 2 2 2
 * 2 1 5
 * 1 3 4
 * 2 1 5
 * 输出:

```

```
* 10
* 12
*
* 提示:
* 1 <= n, q <= 2 * 10^5
* -10^9 <= x <= 10^9
*
* 题目链接: https://cses.fi/problemset/task/2166
*
* 解题思路:
* 使用线段树维护区间信息, 每个节点存储:
* 1. 区间和
* 2. 区间最大前缀和
* 3. 区间最大后缀和
* 4. 区间最大子段和
*
* 时间复杂度:
* - 初始化: O(n) - 需要遍历整个数组构建线段树
* - 更新: O(log n) - 每次更新操作的时间复杂度
* - 查询: O(log n) - 每次查询操作的时间复杂度
* 空间复杂度: O(n) - 线段树需要额外的空间
*
* 工程化考量:
* 1. 边界条件处理: 空数组、单元素数组
* 2. 性能优化: 使用线段树提供高效的区间查询和更新
* 3. 数据结构选择: 线段树适合频繁的区间操作
* 4. 大数处理: 元素值可能很大, 需要确保整数范围
*
* 最优解分析:
* 这是最优解, 因为需要支持动态更新和查询操作, 线段树提供了 O(log n) 的时间复杂度。
* 对于频繁的区间操作, 线段树是最佳选择。
*
* 算法核心:
* 线段树的合并操作:
* - 区间和 = 左子树区间和 + 右子树区间和
* - 区间最大前缀和 = max(左子树最大前缀和, 左子树区间和 + 右子树最大前缀和)
* - 区间最大后缀和 = max(右子树最大后缀和, 右子树区间和 + 左子树最大后缀和)
* - 区间最大子段和 = max(左子树最大子段和, 右子树最大子段和, 左子树最大后缀和 + 右子树最大前缀和)
*
* 算法调试技巧:
* 1. 打印中间过程: 显示线段树的构建和更新过程
* 2. 边界测试: 测试空数组、单元素数组等特殊情况
```

```
* 3. 性能测试：测试大规模数组下的性能表现
*
* 语言特性差异：
* Java 中数组是对象，可以直接访问 length 属性。
* 与 C++相比，Java 有自动内存管理，无需手动释放内存。
* 与 Python 相比，Java 是静态类型语言，需要显式声明类型。
*/
```

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

public class Code22_PrefixSumQueries {
    static class SegmentTreeNode {
        long sum;           // 区间和
        long maxPrefixSum; // 区间最大前缀和
        long maxSuffixSum; // 区间最大后缀和
        long maxSubarraySum; // 区间最大子段和

        SegmentTreeNode() {
            sum = 0;
            maxPrefixSum = 0;
            maxSuffixSum = 0;
            maxSubarraySum = 0;
        }
    }

    static class SegmentTree {
        private SegmentTreeNode[] tree;
        private long[] arr;
        private int n;

        /**
         * 构造函数，初始化线段树
         *
         * @param array 输入数组
         */
        public SegmentTree(long[] array) {
            this.n = array.length;
            this.arr = array.clone();
            // 线段树数组大小通常为 4*n
            this.tree = new SegmentTreeNode[4 * n];
            for (int i = 0; i < 4 * n; i++) {
                tree[i] = new SegmentTreeNode();
            }
        }
    }
}
```

```

    }

    // 构建线段树
    build(1, 0, n - 1);
}

/***
 * 构建线段树
 *
 * @param node 当前节点索引
 * @param start 区间起始位置
 * @param end 区间结束位置
 */
private void build(int node, int start, int end) {
    // 叶子节点
    if (start == end) {
        tree[node].sum = arr[start];
        tree[node].maxPrefixSum = arr[start];
        tree[node].maxSuffixSum = arr[start];
        tree[node].maxSubarraySum = arr[start];
        return;
    }

    int mid = (start + end) / 2;
    // 递归构建左右子树
    build(2 * node, start, mid);
    build(2 * node + 1, mid + 1, end);

    // 合并左右子树的信息
    merge(node);
}

/***
 * 合并左右子树的信息
 *
 * @param node 当前节点索引
 */
private void merge(int node) {
    int leftChild = 2 * node;
    int rightChild = 2 * node + 1;

    // 区间和 = 左子树区间和 + 右子树区间和
    tree[node].sum = tree[leftChild].sum + tree[rightChild].sum;
}

```

```

// 区间最大前缀和 = max(左子树最大前缀和, 左子树区间和 + 右子树最大前缀和)
tree[node].maxPrefixSum = Math.max(
    tree[leftChild].maxPrefixSum,
    tree[leftChild].sum + tree[rightChild].maxPrefixSum
);

// 区间最大后缀和 = max(右子树最大后缀和, 右子树区间和 + 左子树最大后缀和)
tree[node].maxSuffixSum = Math.max(
    tree[rightChild].maxSuffixSum,
    tree[rightChild].sum + tree[leftChild].maxSuffixSum
);

// 区间最大子段和 = max(左子树最大子段和, 右子树最大子段和, 左子树最大后缀和 + 右子树
最大前缀和)
tree[node].maxSubarraySum = Math.max(
    Math.max(tree[leftChild].maxSubarraySum, tree[rightChild].maxSubarraySum),
    tree[leftChild].maxSuffixSum + tree[rightChild].maxPrefixSum
);

}

/***
 * 更新数组中某个位置的值
 *
 * @param index 要更新的位置 (0-based)
 * @param value 新的值
 */
public void update(int index, long value) {
    arr[index] = value;
    update(1, 0, n - 1, index, value);
}

/***
 * 更新线段树中的值
 *
 * @param node 当前节点索引
 * @param start 区间起始位置
 * @param end 区间结束位置
 * @param index 要更新的位置
 * @param value 新的值
 */
private void update(int node, int start, int end, int index, long value) {
    // 叶子节点
    if (start == end) {

```

```

        tree[node].sum = value;
        tree[node].maxPrefixSum = value;
        tree[node].maxSuffixSum = value;
        tree[node].maxSubarraySum = value;
        return;
    }

    int mid = (start + end) / 2;
    // 根据索引决定更新左子树还是右子树
    if (index <= mid) {
        update(2 * node, start, mid, index, value);
    } else {
        update(2 * node + 1, mid + 1, end, index, value);
    }

    // 更新后重新合并信息
    merge(node);
}

/***
 * 查询区间[0, end]内的最大前缀和
 *
 * @param end 区间结束位置
 * @return 区间[0, end]内的最大前缀和
 */
public long queryMaxPrefixSum(int end) {
    return queryMaxPrefixSum(1, 0, n - 1, 0, end);
}

/***
 * 查询区间[left, right]内的最大前缀和
 *
 * @param node 当前节点索引
 * @param start 区间起始位置
 * @param end 区间结束位置
 * @param left 查询区间起始位置
 * @param right 查询区间结束位置
 * @return 区间[left, right]内的最大前缀和
 */
private long queryMaxPrefixSum(int node, int start, int end, int left, int right) {
    // 完全不在查询区间内
    if (start > right || end < left) {
        return Long.MIN_VALUE;
    }
}

```

```

    }

    // 完全在查询区间内
    if (start >= left && end <= right) {
        return tree[node].maxPrefixSum;
    }

    int mid = (start + end) / 2;
    // 递归查询左右子树
    long leftResult = queryMaxPrefixSum(2 * node, start, mid, left, right);
    long rightResult = queryMaxPrefixSum(2 * node + 1, mid + 1, end, left, right);

    // 返回较大值
    return Math.max(leftResult, rightResult);
}

}

/***
 * 主函数 - 测试入口
 */
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    StringTokenizer st = new StringTokenizer(br.readLine());
    int n = Integer.parseInt(st.nextToken());
    int q = Integer.parseInt(st.nextToken());

    long[] arr = new long[n];
    st = new StringTokenizer(br.readLine());
    for (int i = 0; i < n; i++) {
        arr[i] = Long.parseLong(st.nextToken());
    }

    SegmentTree segTree = new SegmentTree(arr);

    for (int i = 0; i < q; i++) {
        st = new StringTokenizer(br.readLine());
        int type = Integer.parseInt(st.nextToken());

        if (type == 1) {
            // 更新操作
            int k = Integer.parseInt(st.nextToken()) - 1; // 转换为 0-based 索引
        }
    }
}

```

```

        long u = Long.parseLong(st.nextToken());
        segTree.update(k, u);
    } else {
        // 查询操作
        int a = Integer.parseInt(st.nextToken()) - 1; // 转换为 0-based 索引
        int b = Integer.parseInt(st.nextToken()) - 1; // 转换为 0-based 索引
        long result = segTree.queryMaxPrefixSum(b);
        out.println(result);
    }
}

out.flush();
out.close();
br.close();
}
}
=====

文件: Code22_PrefixSumQueries.py
=====

# Prefix Sum Queries (CSES 2166)
#
# 题目描述:
# 给定一个数组，支持两种操作：
# 1. 更新位置 k 的值为 u
# 2. 查询区间 [a, b] 内的最大前缀和
#
# 示例:
# 输入:
# 5 3
# 2 2 2 2 2
# 2 1 5
# 1 3 4
# 2 1 5
# 输出:
# 10
# 12
#
# 提示:
# 1 <= n, q <= 2 * 10^5
# -10^9 <= x <= 10^9
#

```

```
# 题目链接: https://cses.fi/problemset/task/2166
#
# 解题思路:
# 使用线段树维护区间信息, 每个节点存储:
# 1. 区间和
# 2. 区间最大前缀和
# 3. 区间最大后缀和
# 4. 区间最大子段和
#
# 时间复杂度:
# - 初始化: O(n) - 需要遍历整个数组构建线段树
# - 更新: O(log n) - 每次更新操作的时间复杂度
# - 查询: O(log n) - 每次查询操作的时间复杂度
# 空间复杂度: O(n) - 线段树需要额外的空间
#
# 工程化考量:
# 1. 边界条件处理: 空数组、单元素数组
# 2. 性能优化: 使用线段树提供高效的区间查询和更新
# 3. 数据结构选择: 线段树适合频繁的区间操作
# 4. 大数处理: 元素值可能很大, 需要确保整数范围
#
# 最优解分析:
# 这是最优解, 因为需要支持动态更新和查询操作, 线段树提供了 O(log n) 的时间复杂度。
# 对于频繁的区间操作, 线段树是最佳选择。
#
# 算法核心:
# 线段树的合并操作:
# - 区间和 = 左子树区间和 + 右子树区间和
# - 区间最大前缀和 = max(左子树最大前缀和, 左子树区间和 + 右子树最大前缀和)
# - 区间最大后缀和 = max(右子树最大后缀和, 右子树区间和 + 左子树最大后缀和)
# - 区间最大子段和 = max(左子树最大子段和, 右子树最大子段和, 左子树最大后缀和 + 右子树最大前缀和)
#
# 算法调试技巧:
# 1. 打印中间过程: 显示线段树的构建和更新过程
# 2. 边界测试: 测试空数组、单元素数组等特殊情况
# 3. 性能测试: 测试大规模数组下的性能表现
#
# 语言特性差异:
# Python 中列表是动态数组, 可以直接获取长度。
# 与 Java 相比, Python 有自动内存管理, 无需手动释放内存。
# 与 C++相比, Python 是动态类型语言, 无需显式声明类型。
```

```
import sys
```

```

from math import inf

class SegmentTreeNode:
    def __init__(self):
        self.sum = 0          # 区间和
        self.maxPrefixSum = 0 # 区间最大前缀和
        self.maxSuffixSum = 0 # 区间最大后缀和
        self.maxSubarraySum = 0 # 区间最大子段和

class SegmentTree:
    def __init__(self, array):
        self.n = len(array)
        self.arr = array[:]
        # 线段树数组大小通常为 4*n
        self.tree = [SegmentTreeNode() for _ in range(4 * self.n)]
        # 构建线段树
        self.build(1, 0, self.n - 1)

    def build(self, node, start, end):
        # 叶子节点
        if start == end:
            self.tree[node].sum = self.arr[start]
            self.tree[node].maxPrefixSum = self.arr[start]
            self.tree[node].maxSuffixSum = self.arr[start]
            self.tree[node].maxSubarraySum = self.arr[start]
            return

        mid = (start + end) // 2
        # 递归构建左右子树
        self.build(2 * node, start, mid)
        self.build(2 * node + 1, mid + 1, end)

        # 合并左右子树的信息
        self.merge(node)

    def merge(self, node):
        leftChild = 2 * node
        rightChild = 2 * node + 1

        # 区间和 = 左子树区间和 + 右子树区间和
        self.tree[node].sum = self.tree[leftChild].sum + self.tree[rightChild].sum

        # 区间最大前缀和 = max(左子树最大前缀和, 左子树区间和 + 右子树最大前缀和)

```

```

        self.tree[node].maxPrefixSum = max(
            self.tree[leftChild].maxPrefixSum,
            self.tree[leftChild].sum + self.tree[rightChild].maxPrefixSum
        )

# 区间最大后缀和 = max(右子树最大后缀和, 右子树区间和 + 左子树最大后缀和)
        self.tree[node].maxSuffixSum = max(
            self.tree[rightChild].maxSuffixSum,
            self.tree[rightChild].sum + self.tree[leftChild].maxSuffixSum
        )

# 区间最大子段和 = max(左子树最大子段和, 右子树最大子段和, 左子树最大后缀和 + 右子树最大
前缀和)
        self.tree[node].maxSubarraySum = max(
            max(self.tree[leftChild].maxSubarraySum, self.tree[rightChild].maxSubarraySum),
            self.tree[leftChild].maxSuffixSum + self.tree[rightChild].maxPrefixSum
        )

def update(self, index, value):
    self.arr[index] = value
    self._update(1, 0, self.n - 1, index, value)

def _update(self, node, start, end, index, value):
    # 叶子节点
    if start == end:
        self.tree[node].sum = value
        self.tree[node].maxPrefixSum = value
        self.tree[node].maxSuffixSum = value
        self.tree[node].maxSubarraySum = value
        return

    mid = (start + end) // 2
    # 根据索引决定更新左子树还是右子树
    if index <= mid:
        self._update(2 * node, start, mid, index, value)
    else:
        self._update(2 * node + 1, mid + 1, end, index, value)

    # 更新后重新合并信息
    self.merge(node)

def queryMaxPrefixSum(self, end):
    return self._queryMaxPrefixSum(1, 0, self.n - 1, 0, end)

```

```

def _queryMaxPrefixSum(self, node, start, end, left, right):
    # 完全不在查询区间内
    if start > right or end < left:
        return -inf

    # 完全在查询区间内
    if start >= left and end <= right:
        return self.tree[node].maxPrefixSum

    mid = (start + end) // 2
    # 递归查询左右子树
    leftResult = self._queryMaxPrefixSum(2 * node, start, mid, left, right)
    rightResult = self._queryMaxPrefixSum(2 * node + 1, mid + 1, end, left, right)

    # 返回较大值
    return max(leftResult, rightResult)

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    q = int(data[1])

    arr = [int(data[i + 2]) for i in range(n)]

    segTree = SegmentTree(arr)

    index = 2 + n
    results = []

    for _ in range(q):
        type = int(data[index])
        index += 1

        if type == 1:
            # 更新操作
            k = int(data[index]) - 1 # 转换为 0-based 索引
            index += 1
            u = int(data[index])
            index += 1

```

```

segTree.update(k, u)
else:
    # 查询操作
    a = int(data[index]) - 1 # 转换为 0-based 索引
    index += 1
    b = int(data[index]) - 1 # 转换为 0-based 索引
    index += 1
    result = segTree.queryMaxPrefixSum(b)
    results.append(str(result))

print('\n'.join(results))

if __name__ == "__main__":
    main()

```

=====

文件: Code23\_StaticRangeSumQueries.cpp

=====

```

// Static Range Sum Queries (CSES 1646)
//
// 题目描述:
// 给定一个数组，处理多个查询：计算区间[a, b]内元素的和。
//
// 示例:
// 输入:
// 8 4
// 3 2 4 5 1 1 5 3
// 2 4
// 5 6
// 1 8
// 3 3
// 输出:
// 11
// 2
// 24
// 4
//
// 提示:
// 1 <= n, q <= 2 * 10^5
// -10^9 <= x <= 10^9
//
// 题目链接: https://cses.fi/problemset/task/1646

```

```
//  
// 解题思路:  
// 使用基础前缀和技巧，预处理前缀和数组，然后 O(1) 时间查询。  
  
//  
// 时间复杂度:  
// - 初始化: O(n) - 需要遍历整个数组构建前缀和数组  
// - 查询: O(1) - 每次查询只需要常数时间  
// 空间复杂度: O(n) - 需要额外的前缀和数组空间  
  
//  
// 工程化考量:  
// 1. 边界条件处理: 空数组、单元素数组  
// 2. 性能优化: 预处理前缀和，查询时 O(1) 时间  
// 3. 空间优化: 必须存储前缀和数组，无法避免  
// 4. 大数处理: 元素值可能很大，需要确保整数范围  
  
//  
// 最优解分析:  
// 这是最优解，因为查询次数可能很多，预处理后可以实现 O(1) 查询时间。  
// 对于静态数组的区间和查询，前缀和是最佳选择。  
  
//  
// 算法核心:  
// 前缀和公式:  
// prefixSum[i] = prefixSum[i-1] + arr[i-1]  
// 区间和公式:  
// sumRange(a, b) = prefixSum[b] - prefixSum[a-1]  
  
//  
// 算法调试技巧:  
// 1. 打印中间过程: 显示前缀和数组的计算过程  
// 2. 边界测试: 测试空数组、单元素数组等特殊情况  
// 3. 性能测试: 测试大规模数组下的性能表现  
  
//  
// 语言特性差异:  
// C++ 中数组需要手动管理内存。  
// 与 Java 相比，C++ 需要手动释放数组内存。  
// 与 Python 相比，C++ 是静态类型语言，需要显式声明数组类型。
```

```
#include <iostream>  
#include <vector>  
using namespace std;  
  
class PrefixSumArray {  
private:  
    vector<long long> prefixSum; // 前缀和数组
```

```

public:
    // 构造函数，初始化前缀和数组
    PrefixSumArray(const vector<int>& arr) {
        int n = arr.size();
        // 创建前缀和数组，大小为 n+1
        // 使用 n+1 可以避免边界检查
        prefixSum.resize(n + 1, 0);

        // 计算前缀和，时间复杂度 O(n)
        for (int i = 1; i <= n; i++) {
            // 前缀和公式：当前前缀和 = 前一个前缀和 + 当前元素
            prefixSum[i] = prefixSum[i - 1] + arr[i - 1];
            // 调试打印：显示前缀和计算过程
            // cout << "位置 " << i << ": prefixSum[" << i << "] = " << prefixSum[i] << endl;
        }
    }

    // 计算子数组区间和
    long long sumRange(int left, int right) {
        // 使用前缀和公式计算区间和，时间复杂度 O(1)
        // 公式：区间和 = 右边界前缀和 - 左边界前缀和
        long long result = prefixSum[right] - prefixSum[left - 1];

        // 调试打印：显示查询过程
        // cout << "查询区间 [" << left << ", " << right << "]: 结果 = " << result << endl;

        return result;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);

    int n, q;
    cin >> n >> q;

    vector<int> arr(n);
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    PrefixSumArray prefixSumArray(arr);
}

```

```
for (int i = 0; i < q; i++) {  
    int a, b;  
    cin >> a >> b;  
    long long result = prefixSumArray.sumRange(a, b);  
    cout << result << "\n";  
}  
  
return 0;  
}
```

=====

文件: Code23\_StaticRangeSumQueries.java

=====

```
package class046;  
  
/**  
 * Static Range Sum Queries (CSES 1646)  
 *  
 * 题目描述:  
 * 给定一个数组，处理多个查询：计算区间[a, b]内元素的和。  
 *  
 * 示例:  
 * 输入:  
 * 8 4  
 * 3 2 4 5 1 1 5 3  
 * 2 4  
 * 5 6  
 * 1 8  
 * 3 3  
 * 输出:  
 * 11  
 * 2  
 * 24  
 * 4  
 *  
 * 提示:  
 * 1 ≤ n, q ≤ 2 * 10^5  
 * -10^9 ≤ x ≤ 10^9  
 *  
 * 题目链接: https://cses.fi/problemset/task/1646  
 */
```

\* 解题思路:

\* 使用基础前缀和技巧，预处理前缀和数组，然后 O(1) 时间查询。

\*

\* 时间复杂度:

\* - 初始化: O(n) - 需要遍历整个数组构建前缀和数组

\* - 查询: O(1) - 每次查询只需要常数时间

\* 空间复杂度: O(n) - 需要额外的前缀和数组空间

\*

\* 工程化考量:

\* 1. 边界条件处理: 空数组、单元素数组

\* 2. 性能优化: 预处理前缀和，查询时 O(1) 时间

\* 3. 空间优化: 必须存储前缀和数组，无法避免

\* 4. 大数处理: 元素值可能很大，需要确保整数范围

\*

\* 最优解分析:

\* 这是最优解，因为查询次数可能很多，预处理后可以实现 O(1) 查询时间。

\* 对于静态数组的区间和查询，前缀和是最佳选择。

\*

\* 算法核心:

\* 前缀和公式:

\*  $\text{prefixSum}[i] = \text{prefixSum}[i-1] + \text{arr}[i-1]$

\* 区间和公式:

\*  $\text{sumRange}(a, b) = \text{prefixSum}[b] - \text{prefixSum}[a-1]$

\*

\* 算法调试技巧:

\* 1. 打印中间过程: 显示前缀和数组的计算过程

\* 2. 边界测试: 测试空数组、单元素数组等特殊情况

\* 3. 性能测试: 测试大规模数组下的性能表现

\*

\* 语言特性差异:

\* Java 中数组是对象，可以直接访问 length 属性。

\* 与 C++ 相比，Java 有自动内存管理，无需手动释放数组内存。

\* 与 Python 相比，Java 是静态类型语言，需要显式声明数组类型。

\*/

```
import java.io.*;
```

```
import java.util.*;
```

```
public class Code23_StaticRangeSumQueries {
```

```
    /**
```

```
     * PrefixSumArray 类用于处理静态区间和查询
```

```
     */
```

```

static class PrefixSumArray {
    private long[] prefixSum; // 前缀和数组

    /**
     * 构造函数，初始化前缀和数组
     *
     * @param arr 输入数组
     */
    public PrefixSumArray(int[] arr) {
        int n = arr.length;
        // 创建前缀和数组，大小为 n+1
        // 使用 n+1 可以避免边界检查
        prefixSum = new long[n + 1];

        // 计算前缀和，时间复杂度 O(n)
        for (int i = 1; i <= n; i++) {
            // 前缀和公式：当前前缀和 = 前一个前缀和 + 当前元素
            prefixSum[i] = prefixSum[i - 1] + arr[i - 1];
            // 调试打印：显示前缀和计算过程
            // System.out.println("位置 " + i + ": prefixSum[" + i + "] = " + prefixSum[i]);
        }
    }

    /**
     * 计算子数组区间和
     *
     * @param left 左边界 (1-based)
     * @param right 右边界 (1-based)
     * @return 子数组元素和
     */
    public long sumRange(int left, int right) {
        // 使用前缀和公式计算区间和，时间复杂度 O(1)
        // 公式：区间和 = 右边界前缀和 - 左边界前缀和
        long result = prefixSum[right] - prefixSum[left - 1];

        // 调试打印：显示查询过程
        // System.out.println("查询区间 [" + left + ", " + right + "]: 结果 = " + result);

        return result;
    }
}

/**

```

```

* 主函数 - 测试入口
*/
public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    StringTokenizer st = new StringTokenizer(br.readLine());
    int n = Integer.parseInt(st.nextToken());
    int q = Integer.parseInt(st.nextToken());

    int[] arr = new int[n];
    st = new StringTokenizer(br.readLine());
    for (int i = 0; i < n; i++) {
        arr[i] = Integer.parseInt(st.nextToken());
    }

    PrefixSumArray prefixSumArray = new PrefixSumArray(arr);

    for (int i = 0; i < q; i++) {
        st = new StringTokenizer(br.readLine());
        int a = Integer.parseInt(st.nextToken());
        int b = Integer.parseInt(st.nextToken());
        long result = prefixSumArray.sumRange(a, b);
        out.println(result);
    }

    out.flush();
    out.close();
    br.close();
}
}
=====
```

文件: Code23\_StaticRangeSumQueries.py

```

=====
# Static Range Sum Queries (CSES 1646)
#
# 题目描述:
# 给定一个数组，处理多个查询：计算区间[a, b]内元素的和。
#
# 示例:
# 输入:
```

```
# 8 4
# 3 2 4 5 1 1 5 3
# 2 4
# 5 6
# 1 8
# 3 3
# 输出:
# 11
# 2
# 24
# 4
#
# 提示:
#  $1 \leq n, q \leq 2 * 10^5$ 
#  $-10^9 \leq x \leq 10^9$ 
#
# 题目链接: https://cses.fi/problemset/task/1646
#
# 解题思路:
# 使用基础前缀和技巧, 预处理前缀和数组, 然后  $O(1)$  时间查询。
#
# 时间复杂度:
# - 初始化:  $O(n)$  - 需要遍历整个数组构建前缀和数组
# - 查询:  $O(1)$  - 每次查询只需要常数时间
# 空间复杂度:  $O(n)$  - 需要额外的前缀和数组空间
#
# 工程化考量:
# 1. 边界条件处理: 空数组、单元素数组
# 2. 性能优化: 预处理前缀和, 查询时  $O(1)$  时间
# 3. 空间优化: 必须存储前缀和数组, 无法避免
# 4. 大数处理: 元素值可能很大, 需要确保整数范围
#
# 最优解分析:
# 这是最优解, 因为查询次数可能很多, 预处理后可以实现  $O(1)$  查询时间。
# 对于静态数组的区间和查询, 前缀和是最佳选择。
#
# 算法核心:
# 前缀和公式:
#  $\text{prefixSum}[i] = \text{prefixSum}[i-1] + \text{arr}[i-1]$ 
# 区间和公式:
#  $\text{sumRange}(a, b) = \text{prefixSum}[b] - \text{prefixSum}[a-1]$ 
#
# 算法调试技巧:
```

```
# 1. 打印中间过程：显示前缀和数组的计算过程
# 2. 边界测试：测试空数组、单元素数组等特殊情况
# 3. 性能测试：测试大规模数组下的性能表现
#
# 语言特性差异：
# Python 中列表是动态数组，可以直接获取长度。
# 与 Java 相比，Python 有自动内存管理，无需手动释放数组内存。
# 与 C++相比，Python 是动态类型语言，无需显式声明数组类型。

import sys

class PrefixSumArray:
    def __init__(self, arr):
        n = len(arr)
        # 创建前缀和数组，大小为 n+1
        # 使用 n+1 可以避免边界检查
        self.prefixSum = [0] * (n + 1)

        # 计算前缀和，时间复杂度 O(n)
        for i in range(1, n + 1):
            # 前缀和公式：当前前缀和 = 前一个前缀和 + 当前元素
            self.prefixSum[i] = self.prefixSum[i - 1] + arr[i - 1]
            # 调试打印：显示前缀和计算过程
            # print(f"位置 {i}: prefixSum[{i}] = {self.prefixSum[i]}")

    def sumRange(self, left, right):
        # 使用前缀和公式计算区间和，时间复杂度 O(1)
        # 公式：区间和 = 右边界前缀和 - 左边界前缀和
        result = self.prefixSum[right] - self.prefixSum[left - 1]

        # 调试打印：显示查询过程
        # print(f"查询区间 [{left}, {right}]: 结果 = {result}")

        return result

def main():
    import sys
    input = sys.stdin.read
    data = input().split()

    n = int(data[0])
    q = int(data[1])
```

```
arr = [int(data[i + 2]) for i in range(n)]
```

```
prefixSumArray = PrefixSumArray(arr)
```

```
index = 2 + n
```

```
results = []
```

```
for _ in range(q):
```

```
    a = int(data[index])
```

```
    index += 1
```

```
    b = int(data[index])
```

```
    index += 1
```

```
    result = prefixSumArray.sumRange(a, b)
```

```
    results.append(str(result))
```

```
print('\n'.join(results))
```

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

```
    main()
```

```
=====
```