==========
文件夹: class081_BinaryIndexedTree
==========

[Markdown 文件]
==========
文件: 树状数组全面学习指南.md
==========
树状数组（Fenwick Tree / Binary Indexed Tree）全面学习指南

1. 树状数组概述

树状数组是一种高效处理前缀和查询和单点更新的数据结构，由 Peter Fenwick 于 1994 年提出。它的优势在于能够在 O(log n)时间内完成单点更新和前缀和查询操作，比线段树实现更简洁，常数更小。

核心优势
- **时间复杂度**：单点更新和前缀和查询均为 O(log n)
- **空间复杂度**：O(n)
- **实现简洁**：代码量小，常数因子低
- **高效实用**：对于特定问题比线段树更高效

2. 树状数组的基础实现与原理解析

2.1 基本原理与 lowbit 操作

树状数组的核心是 lowbit 操作，它用于获取一个整数二进制表示中最低位 1 所对应的值：

```java
public static int lowbit(int i) {
    return i & -i;
}
```

lowbit 操作的数学原理：
- 对于任意整数 x，其二进制补码表示中，-x 等于 x 按位取反加 1
- 当 x 与-x 进行按位与时，只有最低位的 1 会被保留
- 例如：x = 12 (二进制 1100)，-x = ...11110100，x & -x = 0100 = 4

2.2 树状数组的树形结构可视化

树状数组可以看作一个部分覆盖的树形结构，其中每个节点 i 维护的是从 i-lowbit(i)+1 到 i 的区间和：

```
```

```
index: 1(001) covers [1,1]
index: 2(010) covers [1,2]
index: 3(011) covers [3,3]
index: 4(100) covers [1,4]
index: 5(101) covers [5,5]
index: 6(110) covers [5,6]
index: 7(111) covers [7,7]
index: 8(1000) covers [1,8]
```

这种结构使得树状数组能够通过累加或分解区间来高效地执行更新和查询操作。

3. 树状数组的四种经典实现

3.1 单点更新 + 区间查询（基础版本）

最基本的树状数组实现，支持单点增加和区间和查询。

核心操作：
- `add(int i, int v)`：在位置 i 增加 v
- `sum(int i)`：计算从 1 到 i 的前缀和
- `range(int l, int r)`：计算区间[l,r]的和（sum(r) - sum(l-1)）

典型应用场景：
- 动态维护数组的区间和
- 统计逆序对数量
- 频率数组的动态更新与查询
- 求第 k 小元素

3.2 区间更新 + 单点查询（差分版本）

通过结合差分数组的思想，实现区间增加和单点查询。

核心原理：
- 使用树状数组维护差分数组的前缀和
- 区间[l,r]增加 v 转换为：差分数组 l 处加 v，r+1 处减 v
- 单点查询相当于查询差分数组的前缀和

代码示例：
```java
// 区间更新：对区间[l,r]增加 v
public static void rangeAdd(int l, int r, int v) {
    add(l, v);      // 在差分数组的 l 位置增加 v
```

```
    add(r + 1, -v); // 在差分数组的 r+1 位置减少 v
}

// 单点查询：获取位置 i 的值
public static int query(int i) {
    return sum(i); // 差分数组的前缀和即为原数组的值
}
```

典型应用场景：
- 区间标记与查询问题
- 会议预约系统
- 区间增量的累计查询
- 二维范围查询（扩展应用）

3.3 区间更新 + 区间查询（双树状数组版本）

使用两个树状数组维护，支持区间增加和区间查询操作。

核心原理：
- 维护两个树状数组，分别保存差分数组 d[i]和 i*d[i]
- 利用数学推导将区间和转换为两个前缀和的组合
- 通过二维前缀和的计算公式实现区间查询

数学推导：
原数组 a 的区间和[l,r]可以表示为：
```
sum_{i=l}^r a[i] = (r+1)*sum1(r) - sum2(r) - l*sum1(l-1) + sum2(l-1)
```
其中 sum1 和 sum2 分别是差分数组 d[i]和 i*d[i]的前缀和。

3.4 二维树状数组

将树状数组扩展到二维，支持二维平面上的区间操作。

二维单点更新 + 区间查询：
- 类似一维树状数组，但需要处理两个维度
- lowbit 操作分别应用于行和列
- 支持二维前缀和查询

二维区间更新 + 区间查询：
- 使用四个树状数组维护二维差分数组的不同组合项
- 每个树状数组维护不同权重的差分数组（d[i][j], i*d[i][j], j*d[i][j], i*j*d[i][j]）

- 通过数学公式组合四个树状数组的查询结果

4. 树状数组的典型应用案例

4.1 逆序对统计

问题描述：统计数组中逆序对的数量（i<j 且 a[i]>a[j]的对数）

解决方案：
1. 对数组元素进行离散化处理
2. 从右向左遍历数组，对于每个元素 a[i]，查询当前树状数组中小于 a[i]的元素数量
3. 将 a[i]插入树状数组中
4. 累加查询结果即为逆序对数量

时间复杂度：O(n log n)

4.2 二维区间累加与查询

应用场景：图像处理中的矩形区域亮度调整、二维热图动态更新

输入输出示例：
```
输入：
初始化一个 3x3 矩阵为全 0
rangeAdd(1,1,3,3,5) // 整个矩阵增加 5
rangeAdd(1,1,2,2,3) // 左上 2x2 区域再增加 3
rangeQuery(1,1,3,3) // 查询整个矩阵的和

输出：62
// 计算：(1,1)=8, (1,2)=8, (1,3)=5,
//       (2,1)=8, (2,2)=8, (2,3)=5,
//       (3,1)=5, (3,2)=5, (3,3)=5
// 总和=8+8+5+8+8+5+5+5+5=62
```

4.3 频率统计与排名查询

应用场景：在线考试系统的实时排名、用户活跃度统计

解决方案：
- 使用树状数组维护分数或活跃度的频率分布
- 查询某个分数以下的总人数（前缀和）
- 支持动态更新分数或活跃度

4.4 离线二维范围查询

问题描述：在二维平面上，多次对矩形区域增加一个值，然后查询单点的值

解决思路：扩展一维差分树状数组到二维，维护二维差分数组

时间复杂度：O(log n * log m)，其中 n 和 m 是二维数组的维度

5. 性能优化与工程实践

5.1 数据类型优化

- **防止溢出**：对于可能溢出的情况，及时切换到更大的数据类型
- **Java**：使用 long 替代 int
- **C++**：使用 long long 替代 int
- **Python**：自动处理大数，无需额外处理

5.2 输入输出优化

- **快速 IO**：在大规模数据情况下尤为重要
- **Java 示例**：
```java
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
StringTokenizer st = new StringTokenizer(br.readLine());
```
- **C++示例**：
```cpp
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

5.3 离散化技术

当数据范围很大但实际使用的数据点较少时，使用离散化技术可以显著节省内存空间：

1. 收集所有需要处理的数据点
2. 排序并去重
3. 建立映射关系，将原始数据映射到连续的较小整数范围内

5.4 内存访问模式优化

- 调整循环顺序，提高缓存命中率

- 优先按行访问，然后按列访问，符合内存的行优先存储
- 避免频繁的列方向跳跃访问

5.5 并行化处理

对于超大型数据，可以考虑分块并行处理：
- 将数据分成多个子块
- 每个线程处理一部分数据
- 适用于多核处理器，可显著提高大规模数据处理速度

6. 树状数组与其他数据结构对比

6.1 与线段树对比

特性	树状数组	线段树
实现复杂度	简单	复杂
代码量	少	多
常数因子	小	大
单点更新	O(log n)	O(log n)
区间查询	O(log n)	O(log n)
区间更新	部分支持，需要数学转换	完全支持，直接实现
区间最值	不支持	支持
适用场景	前缀和相关操作	更通用的区间操作

6.2 与前缀和数组对比

特性	树状数组	前缀和数组
动态更新	支持，O(log n)	不支持
区间查询	O(log n)	O(1)
空间复杂度	O(n)	O(n)
适用场景	动态数据	静态数据

6.3 与块状数组对比

特性	树状数组	块状数组
实现复杂度	中等	简单
时间复杂度	O(log n)	O(√n)
适用场景	大多数场景	特定优化场景

7. 常见陷阱与调试技巧

7.1 索引问题

- **1-based vs 0-based**：严格区分树状数组（通常从 1 开始）和原数组（可能从 0 开始）的索引
- **边界检查**：确保不会出现索引越界错误，特别是在区间更新时的 r+1 操作

7.2 数学公式错误

- 确保前缀和计算的数学公式正确无误
- 仔细检查多个树状数组的组合方式
- 使用小例子手动验证公式

7.3 数据溢出

- 及时使用更大的数据类型，如 long/long long
- 监控中间计算结果，防止溢出
- 对于极端情况进行测试

7.4 调试技巧

- 使用小例子测试各个操作的正确性
- 打印中间状态以验证树状数组的更新是否正确
- 对比一维情况，逐步扩展到多维
- 编写全面的单元测试，覆盖各种边界情况

8. 代码模板总结

8.1 一维单点更新 + 区间查询模板

```java
public class FenwickTree {
    private int[] tree;
    private int n;

    public FenwickTree(int size) {
        n = size;
        tree = new int[n + 1]; // 索引从 1 开始
    }

    private int lowbit(int x) {
        return x & -x;
    }
```

```
    // 单点更新：在位置 i 增加 v
    public void add(int i, int v) {
        while (i <= n) {
            tree[i] += v;
            i += lowbit(i);
        }
    }

    // 前缀和查询：计算[1,i]的和
    public int sum(int i) {
        int ans = 0;
        while (i > 0) {
            ans += tree[i];
            i -= lowbit(i);
        }
        return ans;
    }

    // 区间查询：计算[l,r]的和
    public int rangeQuery(int l, int r) {
        return sum(r) - sum(l - 1);
    }
}
```

8.2 一维区间更新 + 单点查询模板

```java
public class FenwickTree {
    private int[] tree;
    private int n;

    public FenwickTree(int size) {
        n = size;
        tree = new int[n + 2]; // 多分配一个位置处理 r+1
    }

    private int lowbit(int x) {
        return x & -x;
    }

    private void add(int i, int v) {
        while (i <= n) {
```

```
            tree[i] += v;
            i += lowbit(i);
        }
    }

    // 区间更新：对区间[l,r]增加 v
    public void rangeAdd(int l, int r, int v) {
        add(l, v);
        add(r + 1, -v);
    }

    // 单点查询：获取位置 i 的值
    public int query(int i) {
        int ans = 0;
        while (i > 0) {
            ans += tree[i];
            i -= lowbit(i);
        }
        return ans;
    }
}
```

8.3 一维区间更新 + 区间查询模板

```java
public class FenwickTree {
    private long[] tree1; // 维护 d[i]
    private long[] tree2; // 维护 i*d[i]
    private int n;

    public FenwickTree(int size) {
        n = size;
        tree1 = new long[n + 2];
        tree2 = new long[n + 2];
    }

    private int lowbit(int x) {
        return x & -x;
    }

    private void add(long[] tree, int i, long v) {
        while (i <= n) {
```

```
            tree[i] += v;
            i += lowbit(i);
        }
    }

    private long sum(long[] tree, int i) {
        long ans = 0;
        while (i > 0) {
            ans += tree[i];
            i -= lowbit(i);
        }
        return ans;
    }

    // 区间更新：对区间[l,r]增加 v
    public void rangeAdd(int l, int r, long v) {
        add(tree1, l, v);
        add(tree1, r + 1, -v);
        add(tree2, l, (long)l * v);
        add(tree2, r + 1, (long)(r + 1) * (-v));
    }

    // 前缀和查询：计算[1,i]的和
    public long prefixSum(int i) {
        return (i + 1) * sum(tree1, i) - sum(tree2, i);
    }

    // 区间查询：计算[l,r]的和
    public long rangeQuery(int l, int r) {
        return prefixSum(r) - prefixSum(l - 1);
    }
}
```

8.4 二维区间更新 + 区间查询模板

```java
public class TwoDimensionFenwickTree {
    private long[][] info1; // 维护 d[i][j]
    private long[][] info2; // 维护 d[i][j] * i
    private long[][] info3; // 维护 d[i][j] * j
    private long[][] info4; // 维护 d[i][j] * i * j
    private int n; // 行数
```

```
    private int m; // 列数

    public TwoDimensionFenwickTree(int n, int m) {
        this.n = n;
        this.m = m;
        info1 = new long[n + 2][m + 2];
        info2 = new long[n + 2][m + 2];
        info3 = new long[n + 2][m + 2];
        info4 = new long[n + 2][m + 2];
    }

    private int lowbit(int i) {
        return i & -i;
    }

    private void add(int x, int y, long v) {
        long v1 = v;
        long v2 = v * x;
        long v3 = v * y;
        long v4 = v * x * y;

        for (int i = x; i <= n; i += lowbit(i)) {
            for (int j = y; j <= m; j += lowbit(j)) {
                info1[i][j] += v1;
                info2[i][j] += v2;
                info3[i][j] += v3;
                info4[i][j] += v4;
            }
        }
    }

    private long sum(int x, int y) {
        long ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) {
            for (int j = y; j > 0; j -= lowbit(j)) {
                ans += (x + 1) * (y + 1) * info1[i][j] -
                       (y + 1) * info2[i][j] -
                       (x + 1) * info3[i][j] +
                       info4[i][j];
            }
        }
        return ans;
    }
```

```
    // 区间更新：对矩形区域(a,b)~(c,d)增加 v
    public void rangeAdd(int a, int b, int c, int d, long v) {
        add(a, b, v);
        add(a, d + 1, -v);
        add(c + 1, b, -v);
        add(c + 1, d + 1, v);
    }

    // 区间查询：查询矩形区域(a,b)~(c,d)的和
    public long rangeQuery(int a, int b, int c, int d) {
        return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
    }
}
```

9. 总结与学习建议

树状数组是一种强大而高效的数据结构，特别适合处理前缀和相关的操作。通过掌握树状数组的各种实现变体，我们可以解决从一维到多维的各种区间操作问题。

学习建议：
1. 先掌握基础的单点更新+区间查询版本
2. 理解差分数组思想，学习区间更新+单点查询版本
3. 掌握数学推导，理解区间更新+区间查询版本
4. 扩展到二维情况，学习二维树状数组
5. 通过实际应用案例巩固所学知识
6. 对比不同数据结构，在实际问题中选择最合适的实现

通过系统学习和实践，树状数组将成为解决各类区间操作问题的有力工具。

==========

文件: 树状数组技术总结.md
==========
树状数组（Fenwick Tree）全面技术总结

本文档是树状数组（Binary Indexed Tree 或 Fenwick Tree）的全面技术总结，涵盖从基本原理到高级应用的各个方面，帮助您深入理解并熟练应用这一强大的数据结构。

1. 树状数组基本概念

1.1 树状数组的定义与原理

树状数组是一种能够高效处理数组前缀和查询与单点更新的数据结构，特别适合解决区间查询和更新问题。其核心思想是通过维护一系列"部分和"来实现高效操作。

核心操作：lowbit 函数

树状数组的核心是`lowbit`操作，它用于获取一个整数二进制表示中最低位的 1 所对应的值：

```java
private int lowbit(int x) {
    return x & -x;
}
```

这个操作在树状数组中起到了关键作用，它决定了每个节点管辖的数据范围。

1.2 树状数组的直观理解

树状数组可以看作是一棵有特殊性质的树结构：
- 每个节点存储的是其管辖范围内的元素和
- 节点 i 的父节点是 i + lowbit(i)
- 节点 i 的子节点们的父节点是 i - lowbit(i)

这种结构使得树状数组能够在 O(log n)时间内完成更新和查询操作。

2. 一维树状数组实现

2.1 单点更新与区间查询

这是树状数组最基本的实现，适用于需要频繁更新单个元素并查询区间和的场景。

Java 实现：
```java
class FenwickTree {
    private int[] tree;  // 树状数组，索引从 1 开始
    private int n;       // 数组长度

    public FenwickTree(int size) {
        n = size;
        tree = new int[n + 1];  // 索引 0 不使用
    }

    // lowbit 函数
```

```
    private int lowbit(int x) {
        return x & -x;
    }

    // 单点更新：在 index 位置增加 val
    public void update(int index, int val) {
        while (index <= n) {
            tree[index] += val;
            index += lowbit(index);
        }
    }

    // 查询前缀和：计算[1,index]的和
    public int query(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    // 查询区间和：计算[left,right]的和
    public int rangeQuery(int left, int right) {
        return query(right) - query(left - 1);
    }
}
```

2.2 区间更新与单点查询

利用差分数组的思想，将区间更新转换为两个单点更新。

核心思想：
- 维护差分数组`d[i]`，其中`d[i] = a[i] - a[i-1]`
- 要将区间`[l,r]`增加`val`，只需设置`d[l] += val`和`d[r+1] -= val`
- 查询单点值就是查询差分数组的前缀和

Java 实现：
```java
class FenwickTreeRangeAddPointQuery {
    private int[] tree;  // 维护差分数组的树状数组
    private int n;
```

```
    public FenwickTreeRangeAddPointQuery(int size) {
        n = size;
        tree = new int[n + 2];  // 多分配一个位置防止溢出
    }

    private int lowbit(int x) {
        return x & -x;
    }

    // 更新差分数组中的单点
    private void updateTree(int index, int val) {
        while (index <= n) {
            tree[index] += val;
            index += lowbit(index);
        }
    }

    // 查询差分数组前缀和，即原数组的单点值
    private int queryTree(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    // 区间更新：将[l,r]增加 val
    public void rangeAdd(int l, int r, int val) {
        updateTree(l, val);
        updateTree(r + 1, -val);
    }

    // 单点查询：获取原数组中 index 位置的值
    public int pointQuery(int index) {
        return queryTree(index);
    }
}
```

2.3 区间更新与区间查询

这是树状数组的高级应用，需要使用两个树状数组来维护不同的组合项。

核心原理：
通过数学推导，我们可以将区间查询转化为使用两个树状数组的查询：
- 树状数组 1 维护差分数组 d[i]
- 树状数组 2 维护 i*d[i]

数学推导：
对于原数组 a[i]，其差分数组 d[i] = a[i] - a[i-1]，则有：
```
a[1] + a[2] + ... + a[x] = x*d[1] + (x-1)*d[2] + ... + 1*d[x] = x*(d[1]+d[2]+...+d[x]) - (1*d[2]+2*d[3]+...+(x-1)*d[x])
```

Java 实现：
```java
class FenwickTreeRangeAddRangeQuery {
    private long[] tree1;  // 维护 d[i]
    private long[] tree2;  // 维护 i*d[i]
    private int n;

    public FenwickTreeRangeAddRangeQuery(int size) {
        n = size;
        tree1 = new long[n + 2];  // 多分配一个位置防止溢出
        tree2 = new long[n + 2];
    }

    private int lowbit(int x) {
        return x & -x;
    }

    // 更新树状数组
    private void updateTree(long[] tree, int index, long val) {
        while (index <= n) {
            tree[index] += val;
            index += lowbit(index);
        }
    }

    // 查询树状数组前缀和
    private long queryTree(long[] tree, int index) {
        long sum = 0;
        while (index > 0) {
```

```
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

    // 区间更新：将[l,r]增加 val
    public void rangeAdd(int l, int r, long val) {
        // 更新 tree1
        updateTree(tree1, l, val);
        updateTree(tree1, r + 1, -val);

        // 更新 tree2
        updateTree(tree2, l, val * (l - 1));
        updateTree(tree2, r + 1, -val * r);
    }

    // 查询前缀和：计算[1,index]的和
    public long prefixSum(int index) {
        return index * queryTree(tree1, index) - queryTree(tree2, index);
    }

    // 区间查询：计算[l,r]的和
    public long rangeQuery(int l, int r) {
        return prefixSum(r) - prefixSum(l - 1);
    }
}
```

3. 二维树状数组实现

3.1 单点更新与区间查询

二维树状数组将一维树状数组的概念扩展到二维平面，用于处理二维区域的更新和查询。

Java 实现：
```java
class TwoDimensionalFenwickTreePointUpdateRangeQuery {
    private long[][] tree;  // 二维树状数组
    private int n;          // 行数
    private int m;          // 列数

    public TwoDimensionalFenwickTreePointUpdateRangeQuery(int rows, int cols) {
```

```
        n = rows;
        m = cols;
        tree = new long[n + 1][m + 1];  // 索引从1开始
    }

    private int lowbit(int x) {
        return x & -x;
    }

    // 单点更新：在(x,y)位置增加val
    public void update(int x, int y, long val) {
        for (int i = x; i <= n; i += lowbit(i)) {
            for (int j = y; j <= m; j += lowbit(j)) {
                tree[i][j] += val;
            }
        }
    }

    // 查询前缀和：计算从(1,1)到(x,y)的矩形区域和
    public long query(int x, int y) {
        long sum = 0;
        for (int i = x; i > 0; i -= lowbit(i)) {
            for (int j = y; j > 0; j -= lowbit(j)) {
                sum += tree[i][j];
            }
        }
        return sum;
    }

    // 区域查询：计算从(a,b)到(c,d)的矩形区域和
    public long rangeQuery(int a, int b, int c, int d) {
        // 使用容斥原理
        return query(c, d) - query(a - 1, d) - query(c, b - 1) + query(a - 1, b - 1);
    }
}
```

3.2 区间更新与区间查询

二维树状数组的区间更新区间查询是最复杂的实现，需要维护四个树状数组来支持复杂的区间操作。

核心原理：
结合二维差分数组和数学推导，使用四个树状数组维护不同的组合项：

- 树状数组 1: 维护 d[i][j]
- 树状数组 2: 维护 i*d[i][j]
- 树状数组 3: 维护 j*d[i][j]
- 树状数组 4: 维护 i*j*d[i][j]

Java 实现：
```java
class TwoDimensionalFenwickTreeRangeAddRangeQuery {
    private long[][] info1;  // 维护 d[i][j]
    private long[][] info2;  // 维护 i*d[i][j]
    private long[][] info3;  // 维护 j*d[i][j]
    private long[][] info4;  // 维护 i*j*d[i][j]
    private int n;           // 行数
    private int m;           // 列数

    public TwoDimensionalFenwickTreeRangeAddRangeQuery(int rows, int cols) {
        n = rows;
        m = cols;
        info1 = new long[n + 2][m + 2];  // 多分配位置防止溢出
        info2 = new long[n + 2][m + 2];
        info3 = new long[n + 2][m + 2];
        info4 = new long[n + 2][m + 2];
    }

    private int lowbit(int x) {
        return x & -x;
    }

    // 在点(x,y)处更新四个树状数组
    private void add(int x, int y, long v) {
        long v1 = v;
        long v2 = v * x;
        long v3 = v * y;
        long v4 = v * x * y;

        for (int i = x; i <= n; i += lowbit(i)) {
            for (int j = y; j <= m; j += lowbit(j)) {
                info1[i][j] += v1;
                info2[i][j] += v2;
                info3[i][j] += v3;
                info4[i][j] += v4;
            }
        }
```

```
    }

    // 计算前缀和(1,1)~(x,y)
    private long sum(int x, int y) {
        long ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) {
            for (int j = y; j > 0; j -= lowbit(j)) {
                ans += (x + 1) * (y + 1) * info1[i][j]
                     - (y + 1) * info2[i][j]
                     - (x + 1) * info3[i][j]
                     + info4[i][j];
            }
        }
        return ans;
    }

    // 区间更新：将矩形区域(a,b)~(c,d)的所有元素加 v
    public void rangeAdd(int a, int b, int c, int d, long v) {
        // 利用二维差分数组的特性，更新四个角点
        add(a, b, v);
        add(a, d + 1, -v);
        add(c + 1, b, -v);
        add(c + 1, d + 1, v);
    }

    // 区间查询：计算矩形区域(a,b)~(c,d)的和
    public long rangeQuery(int a, int b, int c, int d) {
        return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
    }
}
```

4. 树状数组经典应用场景

4.1 逆序对统计

树状数组是解决逆序对问题的高效方法，通过离散化技术可以处理大规模数据。

核心思路：
1. 将数组元素离散化，映射到连续的整数范围
2. 从右到左遍历数组，统计已处理元素中比当前元素小的数量
3. 将当前元素插入树状数组

4.2 二维平面统计

二维树状数组常用于处理二维平面上的统计问题，如：
- 图像像素统计
- 二维区域热度图
- 地理信息系统中的范围查询

4.3 动态频率统计

树状数组可以高效地维护动态频率分布，支持：
- 增加/减少某个值的频率
- 查询小于/大于某个值的元素个数

4.4 多维前缀和与差分

树状数组可以扩展到三维及更高维度，用于处理多维空间中的查询和更新操作，但随着维度增加，性能会指数下降。

5. 性能优化与工程实现注意事项

5.1 索引管理

树状数组通常使用 1-based 索引，而大多数编程语言使用 0-based 索引。在实现时需要注意索引转换，避免错误。

5.2 数据类型选择

根据实际数据范围选择合适的数据类型，避免整数溢出：
- 对于小规模数据，可以使用 int
- 对于大规模数据或可能产生大数的情况，应使用 long/long long
- 考虑使用无符号整数以扩大表示范围

5.3 内存优化

对于大型矩阵，可以考虑以下优化方法：
- 使用稀疏表示，只存储非零元素
- 采用离散化技术处理大范围稀疏数据
- 分块处理超大型矩阵

5.4 缓存优化

调整内存访问模式以提高缓存命中率：
- 优先按行访问，然后按列访问，符合内存的行优先存储

- 避免频繁的列方向跳跃访问

5.5 输入输出优化

对于大数据量的输入输出，需要使用快速 IO 方法：
- Java 中使用 BufferedReader 替代 Scanner
- C++中使用 scanf/printf 替代 cin/cout
- Python 中使用 sys.stdin.readline()或读取全部输入后处理

6. 树状数组与其他数据结构对比

6.1 与线段树对比

特性	树状数组	线段树
实现复杂度	较低	较高
代码量	少	多
常数因子	小	大
支持的操作	前缀和、区间和查询	前缀和、区间和、区间最值等更丰富的操作
适用场景	前缀和相关的查询和更新	复杂的区间操作，如区间最值查询

6.2 与前缀和数组对比

特性	树状数组	前缀和数组
更新时间	O(log n)	O(n)
查询时间	O(log n)	O(1)
适用场景	动态数据，频繁更新	静态数据，更新操作少

6.3 与块状数组对比

特性	树状数组	块状数组
时间复杂度	O(log n)	O(√n)
实现复杂度	中等	较低
适用场景	大规模数据	中小规模数据，实现简单的场景

7. 调试与常见错误

7.1 索引越界

树状数组的索引错误是最常见的问题，特别是在二维实现中。需要注意：
- 严格区分 1-based 和 0-based 索引

- 确保 c+1 和 d+1 等操作不会超出数组边界

7.2 数学公式错误

在区间更新区间查询的实现中，数学公式的推导容易出错。建议：
- 使用小例子手动验证公式正确性
- 仔细检查四个树状数组的组合方式

7.3 数据溢出

当处理大规模数据时，容易发生整数溢出。解决方法：
- 及时使用更大的数据类型
- 监控中间计算结果
- 必要时使用高精度数据类型

7.4 初始化错误

树状数组的正确初始化至关重要。确保：
- 树状数组的大小足够
- 正确初始化所有元素
- 区分初始化为 0 和初始化为原始数组值的情况

8. 学习路径与进阶建议

8.1 入门阶段

1. 理解树状数组的基本原理和 lowbit 操作
2. 掌握一维单点更新区间查询的实现
3. 学习差分数组思想，实现区间更新单点查询

8.2 进阶阶段

1. 推导并实现一维区间更新区间查询
2. 学习二维树状数组的单点更新区间查询
3. 理解并实现二维区间更新区间查询
4. 解决逆序对统计等经典问题

8.3 高级阶段

1. 掌握离散化技术
2. 解决复杂的二维统计问题
3. 探索树状数组在其他领域的应用
4. 研究树状数组的扩展和变种

9. 总结

树状数组是一种非常高效且灵活的数据结构，特别适合处理前缀和相关的查询和更新操作。通过本文的学习，您应该能够掌握：

1. 树状数组的基本原理和核心操作
2. 一维和二维树状数组的各种实现
3. 树状数组在不同问题场景中的应用
4. 性能优化和工程实现的注意事项
5. 与其他数据结构的对比和选择策略

通过大量练习和实际应用，树状数组将成为您解决各类区间操作问题的有力工具。

==========

文件: 树状数组经典题目集.md
==========
树状数组（Fenwick Tree）经典题目集

本文档收集了各大数据结构与算法平台上以树状数组为最优解的经典题目，每个题目都配有详细的解答代码和解析。

1. 基础题目：单点更新与区间查询

1.1 LeetCode 307. 区域和检索 - 数组可修改

题目描述：给定一个整数数组 nums，处理以下两种类型的操作：
1. 更新数组下标 i 处的值为 val
2. 返回数组中索引 left 和索引 right 之间的元素和（包含 left 和 right）

题目链接：https://leetcode.cn/problems/range-sum-query-mutable/

难度：中等

解答思路：
- 使用基础的单点更新+区间查询树状数组
- 维护原数组和树状数组，通过差分进行更新

Java 代码：
```java
class NumArray {
    private int[] tree;  // 树状数组
```

```
private int[] nums;  // 原始数组
private int n;       // 数组长度

public NumArray(int[] nums) {
    this.nums = nums;
    n = nums.length;
    tree = new int[n + 1];  // 树状数组索引从 1 开始

    // 初始化树状数组
    for (int i = 0; i < n; i++) {
        updateTree(i + 1, nums[i]);
    }
}

// lowbit 函数
private int lowbit(int x) {
    return x & -x;
}

// 更新树状数组
private void updateTree(int index, int val) {
    while (index <= n) {
        tree[index] += val;
        index += lowbit(index);
    }
}

// 查询前缀和
private int prefixSum(int index) {
    int sum = 0;
    while (index > 0) {
        sum += tree[index];
        index -= lowbit(index);
    }
    return sum;
}

// 更新原数组值并调整树状数组
public void update(int index, int val) {
    int diff = val - nums[index];
    nums[index] = val;
    updateTree(index + 1, diff);  // 转换为树状数组索引（+1）
}
```

```
    // 查询区间和
    public int sumRange(int left, int right) {
        // 区间和 = 右边界前缀和 - 左边界-1 的前缀和
        return prefixSum(right + 1) - prefixSum(left);
    }
}
```

C++代码：
```cpp
class NumArray {
private:
    vector<int> tree;  // 树状数组
    vector<int> nums;  // 原始数组
    int n;             // 数组长度

    // lowbit 函数
    int lowbit(int x) {
        return x & -x;
    }

    // 更新树状数组
    void updateTree(int index, int val) {
        while (index <= n) {
            tree[index] += val;
            index += lowbit(index);
        }
    }

    // 查询前缀和
    int prefixSum(int index) {
        int sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }

public:
    NumArray(vector<int>& nums) {
        this->nums = nums;
```

```
        n = nums.size();
        tree.resize(n + 1, 0);  // 树状数组索引从 1 开始

        // 初始化树状数组
        for (int i = 0; i < n; i++) {
            updateTree(i + 1, nums[i]);
        }
    }

    // 更新原数组值并调整树状数组
    void update(int index, int val) {
        int diff = val - nums[index];
        nums[index] = val;
        updateTree(index + 1, diff);  // 转换为树状数组索引（+1）
    }

    // 查询区间和
    int sumRange(int left, int right) {
        return prefixSum(right + 1) - prefixSum(left);
    }
};
```

Python 代码：
```python
class NumArray:
    def __init__(self, nums):
        self.nums = nums
        self.n = len(nums)
        self.tree = [0] * (self.n + 1)  # 树状数组索引从 1 开始

        # 初始化树状数组
        for i in range(self.n):
            self._update_tree(i + 1, nums[i])

    # lowbit 函数
    def _lowbit(self, x):
        return x & -x

    # 更新树状数组
    def _update_tree(self, index, val):
        while index <= self.n:
            self.tree[index] += val
```

```
            index += self._lowbit(index)

    # 查询前缀和
    def _prefix_sum(self, index):
        sum_val = 0
        while index > 0:
            sum_val += self.tree[index]
            index -= self._lowbit(index)
        return sum_val

    # 更新原数组值并调整树状数组
    def update(self, index, val):
        diff = val - self.nums[index]
        self.nums[index] = val
        self._update_tree(index + 1, diff)  # 转换为树状数组索引（+1）

    # 查询区间和
    def sumRange(self, left, right):
        return self._prefix_sum(right + 1) - self._prefix_sum(left)
```

2. 进阶题目：区间更新与单点查询

2.1 洛谷 P3368 【模板】树状数组 2

题目描述：给定一个长度为 n 的数组，支持两种操作：
1. 将区间[l, r]加上 k
2. 输出第 x 个数的值

题目链接：https://www.luogu.com.cn/problem/P3368

难度：普及+

解答思路：
- 使用差分思想结合树状数组
- 树状数组维护差分数组，单点查询即差分数组的前缀和

Java 代码：
```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
```

```
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Main {
    static int MAXN = 500001;
    static int[] tree = new int[MAXN];  // 树状数组，维护差分数组
    static int[] original;  // 原始数组
    static int n, m;

    public static int lowbit(int x) {
        return x & -x;
    }

    // 更新树状数组中某个位置的值
    public static void update(int pos, int val) {
        while (pos <= n) {
            tree[pos] += val;
            pos += lowbit(pos);
        }
    }

    // 查询差分数组前缀和（即原数组某个位置的值）
    public static int query(int pos) {
        int res = 0;
        while (pos > 0) {
            res += tree[pos];
            pos -= lowbit(pos);
        }
        return res;
    }

    // 区间更新，将[l,r]增加 val
    public static void rangeUpdate(int l, int r, int val) {
        update(l, val);       // 在差分数组的 l 位置加 val
        update(r + 1, -val);  // 在差分数组的 r+1 位置减 val
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        StreamTokenizer st = new StreamTokenizer(br);

        // 读取 n 和 m
```

```
        st.nextToken();
        n = (int) st.nval;
        st.nextToken();
        m = (int) st.nval;

        original = new int[n + 1];
        // 读取原始数组
        for (int i = 1; i <= n; i++) {
            st.nextToken();
            original[i] = (int) st.nval;
        }

        // 构建差分数组的树状数组表示
        for (int i = 1; i <= n; i++) {
            rangeUpdate(i, i, original[i]);
        }

        // 处理操作
        for (int i = 0; i < m; i++) {
            st.nextToken();
            int op = (int) st.nval;

            if (op == 1) {
                // 区间更新操作
                st.nextToken();
                int l = (int) st.nval;
                st.nextToken();
                int r = (int) st.nval;
                st.nextToken();
                int val = (int) st.nval;
                rangeUpdate(l, r, val);
            } else {
                // 单点查询操作
                st.nextToken();
                int pos = (int) st.nval;
                out.println(query(pos));
            }
        }

        out.flush();
        out.close();
        br.close();
    }
```

}
```

## 3. 高级题目：区间更新与区间查询

### 3.1 洛谷 P3372 【模板】树状数组 1

**题目描述**：给定一个长度为 n 的数组，支持两种操作：
1. 将区间[l, r]加上 k
2. 查询区间[l, r]的和

**题目链接**：https://www.luogu.com.cn/problem/P3372

**难度**：提高+/省选-

**解答思路**：
- 使用两个树状数组维护，结合数学推导实现区间更新和区间查询
- 树状数组 1 维护差分数组 d[i]
- 树状数组 2 维护 d[i]*(i-1)

**Java 代码**：
```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.io.StreamTokenizer;

public class Main {
    static int MAXN = 100001;
    static long[] tree1 = new long[MAXN];  // 维护 d[i]
    static long[] tree2 = new long[MAXN];  // 维护 d[i]*(i-1)
    static int n, m;

    public static int lowbit(int x) {
        return x & -x;
    }

    // 更新树状数组
    public static void update(long[] tree, int pos, long val) {
        while (pos <= n) {
            tree[pos] += val;
```
```

```
            pos += lowbit(pos);
        }
    }

    // 查询树状数组前缀和
    public static long query(long[] tree, int pos) {
        long res = 0;
        while (pos > 0) {
            res += tree[pos];
            pos -= lowbit(pos);
        }
        return res;
    }

    // 区间更新，将[l,r]增加 val
    public static void rangeUpdate(int l, int r, long val) {
        // 更新 tree1：d[l] += val, d[r+1] -= val
        update(tree1, l, val);
        update(tree1, r + 1, -val);

        // 更新 tree2：d[l]*(l-1) += val*(l-1), d[r+1]*r -= val*r
        update(tree2, l, val * (l - 1));
        update(tree2, r + 1, -val * r);
    }

    // 前缀和查询，计算[1,pos]的和
    public static long prefixSum(int pos) {
        return pos * query(tree1, pos) - query(tree2, pos);
    }

    // 区间查询，计算[l,r]的和
    public static long rangeQuery(int l, int r) {
        return prefixSum(r) - prefixSum(l - 1);
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        StreamTokenizer st = new StreamTokenizer(br);

        // 读取 n 和 m
        st.nextToken();
        n = (int) st.nval;
```

```
        st.nextToken();
        m = (int) st.nval;

        // 初始化数组
        long[] a = new long[n + 1];
        for (int i = 1; i <= n; i++) {
            st.nextToken();
            a[i] = (long) st.nval;
            // 通过区间更新[i,i]来设置初始值
            rangeUpdate(i, i, a[i]);
        }

        // 处理操作
        for (int i = 0; i < m; i++) {
            st.nextToken();
            int op = (int) st.nval;

            if (op == 1) {
                // 区间更新操作
                st.nextToken();
                int l = (int) st.nval;
                st.nextToken();
                int r = (int) st.nval;
                st.nextToken();
                long val = (long) st.nval;
                rangeUpdate(l, r, val);
            } else {
                // 区间查询操作
                st.nextToken();
                int l = (int) st.nval;
                st.nextToken();
                int r = (int) st.nval;
                out.println(rangeQuery(l, r));
            }
        }

        out.flush();
        out.close();
        br.close();
    }
}
```

4. 二维树状数组题目

4.1 LeetCode 308. 二维区域和检索 - 可变

题目描述：给定一个二维矩阵，支持两种操作：
1. 更新矩阵中某个位置的值
2. 计算子矩阵元素的和

题目链接：https://leetcode.cn/problems/range-sum-query-2d-mutable/

难度：困难

解答思路：
- 使用二维树状数组实现单点更新和区域查询
- 树状数组索引从 1 开始，注意与原始矩阵索引的转换

Java 代码：
```java
class NumMatrix {
    private int[][] tree;  // 二维树状数组
    private int[][] nums;  // 原始矩阵
    private int n;         // 行数
    private int m;         // 列数

    public NumMatrix(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return;
        }

        n = matrix.length;
        m = matrix[0].length;
        tree = new int[n + 1][m + 1];  // 树状数组索引从 1 开始
        nums = new int[n + 1][m + 1];  // 原始数组也从 1 开始存储

        // 初始化树状数组
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                update(i, j, matrix[i][j]);
            }
        }
    }

    // lowbit 函数
```

```
    private int lowbit(int x) {
        return x & -x;
    }

    // 更新树状数组中的某个位置
    private void updateTree(int row, int col, int val) {
        for (int i = row; i <= n; i += lowbit(i)) {
            for (int j = col; j <= m; j += lowbit(j)) {
                tree[i][j] += val;
            }
        }
    }

    // 查询从(1,1)到(row,col)的前缀和
    private int queryTree(int row, int col) {
        int sum = 0;
        for (int i = row; i > 0; i -= lowbit(i)) {
            for (int j = col; j > 0; j -= lowbit(j)) {
                sum += tree[i][j];
            }
        }
        return sum;
    }

    // 更新原始矩阵中的值
    public void update(int row, int col, int val) {
        int diff = val - nums[row + 1][col + 1];  // 计算差值
        nums[row + 1][col + 1] = val;  // 更新原始数组
        updateTree(row + 1, col + 1, diff);  // 更新树状数组
    }

    // 查询子矩阵和
    public int sumRegion(int row1, int col1, int row2, int col2) {
        // 转换为树状数组索引并应用容斥原理
        return queryTree(row2 + 1, col2 + 1)
             - queryTree(row1, col2 + 1)
             - queryTree(row2 + 1, col1)
             + queryTree(row1, col1);
    }
}
```

5. 逆序对统计题目

5.1 LeetCode 315. 计算右侧小于当前元素的个数

题目描述：给你一个整数数组 nums ，按要求返回一个新数组 counts 。数组 counts 有该性质：counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

题目链接：https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

难度：困难

解答思路：
- 将问题转化为逆序对统计
- 使用树状数组 + 离散化处理重复元素
- 从右到左遍历，统计比当前元素小的已处理元素数量

Java 代码：
```java
import java.util.*;

class Solution {
    public List<Integer> countSmaller(int[] nums) {
        int n = nums.length;
        List<Integer> result = new ArrayList<>(n);

        // 离散化处理
        Set<Integer> set = new HashSet<>();
        for (int num : nums) {
            set.add(num);
        }
        List<Integer> sortedNums = new ArrayList<>(set);
        Collections.sort(sortedNums);
        Map<Integer, Integer> valueToIndex = new HashMap<>();
        for (int i = 0; i < sortedNums.size(); i++) {
            valueToIndex.put(sortedNums.get(i), i + 1);  // 树状数组索引从 1 开始
        }

        // 初始化树状数组
        FenwickTree tree = new FenwickTree(sortedNums.size());

        // 从右向左遍历数组
        for (int i = n - 1; i >= 0; i--) {
            int index = valueToIndex.get(nums[i]);
            // 查询比当前元素小的已处理元素数量
```

```
            result.add(0, tree.query(index - 1));
            // 将当前元素插入树状数组
            tree.update(index, 1);
        }

        return result;
    }

    // 树状数组类
    class FenwickTree {
        private int[] tree;
        private int n;

        public FenwickTree(int size) {
            n = size;
            tree = new int[n + 1];
        }

        private int lowbit(int x) {
            return x & -x;
        }

        public void update(int index, int delta) {
            while (index <= n) {
                tree[index] += delta;
                index += lowbit(index);
            }
        }

        public int query(int index) {
            int sum = 0;
            while (index > 0) {
                sum += tree[index];
                index -= lowbit(index);
            }
            return sum;
        }
    }
}
```

6. 高维树状数组应用

6.1 Codeforces 61E Enemy is weak

题目描述：给定一个数列，求满足 i<j<k 且 a[i]>a[j]>a[k] 的三元组数目。

题目链接：https://codeforces.com/problemset/problem/61/E

难度：高级

解答思路：
- 使用两个树状数组分别统计每个元素左边有多少比它大的元素，右边有多少比它小的元素
- 对于每个元素 a[j]，其贡献为 left[j] * right[j]
- 总和即为所有满足条件的三元组数目

C++代码：
```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
using namespace std;

class FenwickTree {
private:
    vector<long long> tree;
    int n;

    int lowbit(int x) {
        return x & -x;
    }

public:
    FenwickTree(int size) {
        n = size;
        tree.resize(n + 1, 0);
    }

    void update(int index, long long delta) {
        while (index <= n) {
            tree[index] += delta;
            index += lowbit(index);
        }
    }
```

```
    long long query(int index) {
        long long sum = 0;
        while (index > 0) {
            sum += tree[index];
            index -= lowbit(index);
        }
        return sum;
    }
};

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n;
    cin >> n;
    vector<long long> a(n);
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }

    // 离散化
    vector<long long> sorted_a(a);
    sort(sorted_a.begin(), sorted_a.end());
    sorted_a.erase(unique(sorted_a.begin(), sorted_a.end()), sorted_a.end());
    map<long long, int> value_to_index;
    for (int i = 0; i < sorted_a.size(); i++) {
        value_to_index[sorted_a[i]] = i + 1;  // 树状数组索引从 1 开始
    }

    // 计算每个元素右边比它小的元素个数
    vector<long long> right(n, 0);
    FenwickTree right_tree(sorted_a.size());
    for (int i = n - 1; i >= 0; i--) {
        int index = value_to_index[a[i]];
        right[i] = right_tree.query(index - 1);
        right_tree.update(index, 1);
    }

    // 计算每个元素左边比它大的元素个数
    vector<long long> left(n, 0);
    FenwickTree left_tree(sorted_a.size());
    for (int i = 0; i < n; i++) {
```

```
        int index = value_to_index[a[i]];
        left[i] = left_tree.query(sorted_a.size()) - left_tree.query(index);
        left_tree.update(index, 1);
    }

    // 计算总贡献
    long long result = 0;
    for (int i = 0; i < n; i++) {
        result += left[i] * right[i];
    }

    cout << result << endl;

    return 0;
}
```

7. 实际应用题目

7.1 HackerRank Reverse Shuffle Merge

题目描述：给定一个字符串 s，找到一个字典序最小的字符串 a，使得 s 可以通过将 a 的一个排列与 a 的逆序排列合并得到。

题目链接：https://www.hackerrank.com/challenges/reverse-shuffle-merge/problem

难度：中等

解答思路：
- 使用贪心算法结合树状数组来高效地选择字符
- 树状数组用于快速查询剩余可用字符的数量

Python 代码：
```python
def reverseShuffleMerge(s):
    from collections import defaultdict

    # 统计每个字符出现的总次数
    char_count = defaultdict(int)
    for char in s:
        char_count[char] += 1

    # 每个字符在结果中出现的次数应为总次数的一半
```

```
required = {char: cnt // 2 for char, cnt in char_count.items()}
remaining = char_count.copy()

# 用于跟踪结果中已使用的字符数量
used = defaultdict(int)

# 树状数组类，用于查询剩余字符中最小字符的位置
class FenwickTree:
    def __init__(self, size):
        self.n = size
        self.tree = [0] * (self.n + 1)

    def update(self, index, delta):
        while index <= self.n:
            self.tree[index] += delta
            index += index & -index

    def query(self, index):
        res = 0
        while index > 0:
            res += self.tree[index]
            index -= index & -index
        return res

# 字符到索引的映射（a=1, b=2, ..., z=26）
char_to_idx = {chr(ord('a') + i): i + 1 for i in range(26)}
idx_to_char = {i + 1: chr(ord('a') + i) for i in range(26)}

# 初始化树状数组，初始时所有字符都可用
ft = FenwickTree(26)
for char, cnt in remaining.items():
    if cnt > 0:
        ft.update(char_to_idx[char], cnt)

result = []
# 从后往前遍历 s 的逆序（相当于从 s 的开头开始处理）
for char in reversed(s):
    # 如果当前字符已经达到需要的数量，跳过
    if used[char] == required[char]:
        remaining[char] -= 1
        ft.update(char_to_idx[char], -1)
        continue
```

```
        # 尝试跳过当前字符，但需要确保后面还有足够的字符来构成结果
        while result and result[-1] > char and used[result[-1]] + remaining[result[-1]] >
required[result[-1]]:
            # 可以丢弃最后一个字符，回退使用计数
            last_char = result.pop()
            used[last_char] -= 1
            ft.update(char_to_idx[last_char], 1)

        # 使用当前字符
        result.append(char)
        used[char] += 1
        remaining[char] -= 1
        ft.update(char_to_idx[char], -1)

    return ''.join(result)
```

8. 总结与学习建议

树状数组是一种非常强大的数据结构，尤其适合处理前缀和相关的查询和更新操作。通过以上题目的练习，可以全面掌握树状数组的各种应用场景：

1. **单点更新 + 区间查询**：基础操作，适用于动态维护数组和查询区间和
2. **区间更新 + 单点查询**：利用差分数组思想，将区间更新转换为两个单点更新
3. **区间更新 + 区间查询**：使用两个树状数组维护，结合数学推导实现复杂操作
4. **二维树状数组**：将一维树状数组扩展到二维平面，处理二维区域查询
5. **逆序对统计**：结合离散化技术，高效统计逆序对或相关问题
6. **组合问题**：与其他算法（如贪心）结合，解决更复杂的问题

学习建议：
1. 先掌握一维树状数组的基本操作和差分数组思想
2. 理解区间更新和区间查询的数学推导过程
3. 尝试实现二维树状数组，理解其在二维平面上的工作原理
4. 通过逆序对问题，学习树状数组结合离散化的应用
5. 尝试解决一些综合性问题，如 Codeforces 上的题目
6. 对比树状数组与线段树的优劣，在实际问题中选择最合适的数据结构

通过系统学习和大量练习，树状数组将成为解决各类区间操作问题的有力工具。

==========

[代码文件]
==========

文件: Code01_IndexTreeSingleAddIntervalQuery.java

```
==========================================
package class108;

/**
 * 树状数组单点增加、范围查询模板
 *
 * 树状数组（Binary Indexed Tree 或 Fenwick Tree）是一种高效的数据结构，
 * 用于处理数组的前缀和查询和单点更新操作，两者的时间复杂度均为 O(log n)。
 *
 * 本实现支持两种操作：
 * 1. 单点更新：在数组的某个位置增加一个值
 * 2. 区间查询：查询数组中某一区间内所有元素的和
 *
 * 测试链接: https://www.luogu.com.cn/problem/P3374
 * 提交时请把类名改成"Main"，可以直接通过
 *
 * 时间复杂度分析：
 * - 单点更新(add): O(log n)，因为每次循环 i 的变化量至少翻倍，树高为 log n
 * - 前缀和查询(sum): O(log n)，因为每次循环 i 至少减少一半，最多查询 log n 个节点
 * - 区间查询(range): O(log n)，由两次前缀和查询组成
 * 空间复杂度: O(n)
 *
 * 树状数组的核心思想深度解析：
 *
 * 1. 二进制索引表示法：
 *    树状数组巧妙利用了数字的二进制表示特性。对于任何正整数 i，
 *    我们可以通过 lowbit(i) = i & -i 操作找到其二进制表示中最低位的 1。
 *    例如：6 的二进制是 110，lowbit(6)=2，即 10
 *
 * 2. 树状数组的树形结构：
 *    树状数组构建了一个虚拟的树形结构，其中：
 *    - 每个节点 tree[i]存储原始数组中某一段连续区间的和
 *    - 该区间的范围是[i - lowbit(i) + 1, i]
 *    - 每个节点的父节点是 i + lowbit(i)
 *
 * 3. 树形结构可视化（以 n=8 为例）：
 *    tree[1] (0001) 管辖范围: [1, 1]
 *    tree[2] (0010) 管辖范围: [1, 2]
 *    tree[3] (0011) 管辖范围: [3, 3]
 *    tree[4] (0100) 管辖范围: [1, 4]
 *    tree[5] (0101) 管辖范围: [5, 5]
 *    tree[6] (0110) 管辖范围: [5, 6]
```

```
 *    tree[7] (0111) 管辖范围: [7, 7]
 *    tree[8] (1000) 管辖范围: [1, 8]
 *
 * 4. 更新与查询的过程：
 *    - 更新操作：从当前节点开始，一直向上更新所有包含它的父节点
 *    - 查询操作：从当前节点开始，一直向上累加所有父节点的值
 *    这两个过程都只需要 O(log n)时间
 *
 * 树状数组与其他数据结构的对比：
 * - 相比前缀和数组：支持高效更新，但查询操作略慢
 * - 相比线段树：代码更简洁，常数更小，但功能较局限
 * - 相比平衡二叉搜索树：实现简单，适用于特定场景
 */

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

public class Code01_IndexTreeSingleAddIntervalQuery {

    /**
     * 最大数据范围，根据题目要求设置
     * 注意：在实际应用中，应根据具体问题调整此值，避免内存浪费
     */
    public static int MAXN = 500001;

    /**
     * 树状数组的核心数组，用于存储区间和信息
     * 注意：树状数组的下标必须从 1 开始，不从 0 开始，这是由树状数组的性质决定的
     */
    public static int[] tree = new int[MAXN];

    /**
     * n: 数组长度
     * m: 操作次数
     */
    public static int n, m;

    /**
     * lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
```

```
     * 例如：lowbit(12) = lowbit(1100) = 4 = 100
     *
     * 实现原理：利用计算机补码的特性，-i 是 i 的补码表示
     * 当 i & (-i)时，只有最低位的 1 会被保留
     *
     * @param i 输入的整数
     * @return i 的二进制表示中最低位的 1 所对应的值
     */
    public static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点更新操作：在位置 i 上增加 v
     * 时间复杂度：O(log n)，因为每次循环 i 的变化量至少翻倍
     *
     * @param i 要更新的位置（从 1 开始）
     * @param v 要增加的值
     */
    public static void add(int i, int v) {
        // 当 i 超过数组长度时停止更新
        while (i <= n) {
            // 更新当前节点的值
            tree[i] += v;
            // 移动到父节点继续更新
            i += lowbit(i);
        }
    }

    /**
     * 前缀和查询操作：查询从 1 到 i 的元素和
     * 时间复杂度：O(log n)，因为每次循环 i 至少减少一半
     *
     * @param i 前缀的结束位置（从 1 开始）
     * @return 1 到 i 位置的元素和
     */
    public static int sum(int i) {
        int ans = 0;
        // 当 i 小于等于 0 时停止查询
        while (i > 0) {
            // 累加当前节点的值
            ans += tree[i];
            // 移动到父节点继续查询
```

```
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 区间查询操作：查询从 l 到 r 的元素和
 * 时间复杂度：O(log n)，由两次前缀和查询组成
 *
 * @param l 区间的起始位置（从 1 开始）
 * @param r 区间的结束位置（从 1 开始）
 * @return l 到 r 位置的元素和
 */
public static int range(int l, int r) {
    // 利用前缀和的性质：区间和 = 前缀和(r) - 前缀和(l-1)
    return sum(r) - sum(l - 1);
}

/**
 * 主函数，处理输入输出
 * 使用高效的输入输出方式（BufferedReader + StreamTokenizer + PrintWriter）
 * 以应对大规模数据输入输出的情况
 */
public static void main(String[] args) throws IOException {
    // 初始化输入输出流
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度 n 和操作次数 m
    in.nextToken();
    n = (int) in.nval;
    in.nextToken();
    m = (int) in.nval;

    // 初始化数组：逐个读取初始值并插入树状数组
    for (int i = 1, v; i <= n; i++) {
        in.nextToken();
        v = (int) in.nval;
        // 调用 add 函数将初始值插入树状数组
        add(i, v);
    }
```

```
        // 处理 m 个操作
        for (int i = 1, a, b, c; i <= m; i++) {
            in.nextToken();
            a = (int) in.nval;  // 操作类型
            in.nextToken();
            b = (int) in.nval;  // 第一个参数（位置或左边界）
            in.nextToken();
            c = (int) in.nval;  // 第二个参数（值或右边界）

            // 判断操作类型
            if (a == 1) {
                // 操作 1：单点更新，在位置 b 增加 c
                add(b, c);
            } else {
                // 操作 2：区间查询，查询区间[b, c]的和
                out.println(range(b, c));
            }
        }

        // 刷新输出流并关闭资源
        out.flush();
        out.close();
        br.close();
    }

    /**
 * 树状数组的性能优化与工程实践：
 *
 * 1. 数据类型优化：
 *    - 使用 long 或 long long 代替 int 避免溢出
 *    - 对于不同问题，选择合适的数据类型平衡内存和精度
 *
 * 2. 输入输出优化：
 *    - 对于大规模数据，使用快速 IO 方法（如 C++的 scanf/printf 或关闭同步）
 *    - 在 Java 中使用 BufferedReader 和 PrintWriter
 *    - 在 Python 中一次性读取所有输入再处理
 *
 * 3. 离散化技术：
 *    - 当元素范围很大（如 1e9）但数量有限（如 1e5）时，必须离散化
 *    - 离散化步骤：去重、排序、映射
 *    - 注意处理相等元素的情况
 *
 * 4. 边界条件处理：
```

```
 *     - 树状数组下标从 1 开始，需要注意转换
 *     - 处理区间端点时避免越界
 *     - 考虑初始条件和特殊输入（如空数组）
 *
 * 5. 常见陷阱：
 *     - 整数溢出：忘记使用大整数类型
 *     - 下标错误：混淆从 0 开始和从 1 开始的索引
 *     - 范围错误：查询或更新时超出数组范围
 *     - 离散化错误：未正确处理重复元素或排序
 *
 * 6. 测试用例设计：
 *     - 基本功能测试：确保单点更新和区间查询正确
 *     - 边界测试：空数组、单元素数组、最大范围
 *     - 性能测试：大规模数据下的运行时间
 *     - 极端测试：多次更新同一位置、大数值操作
 *
 * 7. 线程安全考虑：
 *     - 树状数组默认不是线程安全的
 *     - 在并发环境中，需要添加适当的同步机制
 *     - 考虑使用读写锁提高并发性能
 */
}

/**
 * 以下是 C++实现的树状数组（Fenwick Tree）单点更新区间查询代码
 *
 * #include <iostream>
 * #include <vector>
 * using namespace std;
 *
 * const int MAXN = 500001; // 最大数据范围
 *
 * class IndexTree {
 * private:
 *     vector<long long> tree; // 使用 long long 防止溢出
 *     int n; // 数组长度
 *
 *     // lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
 *     int lowbit(int i) {
 *         return i & -i;
 *     }
 *
 * public:
```

```
*     // 构造函数
*     IndexTree(int size) {
*         n = size;
*         tree.resize(n + 1, 0); // 树状数组下标从 1 开始
*     }
*
*     // 单点更新操作：在位置 i 上增加 v
*     void add(int i, long long v) {
*         while (i <= n) {
*             tree[i] += v;
*             i += lowbit(i);
*         }
*     }
*
*     // 前缀和查询操作：查询从 1 到 i 的元素和
*     long long sum(int i) {
*         long long ans = 0;
*         while (i > 0) {
*             ans += tree[i];
*             i -= lowbit(i);
*         }
*         return ans;
*     }
*
*     // 区间查询操作：查询从 l 到 r 的元素和
*     long long range(int l, int r) {
*         return sum(r) - sum(l - 1);
*     }
* };
*
* int main() {
*     ios::sync_with_stdio(false); // 关闭同步，加速输入输出
*     cin.tie(nullptr);
*     cout.tie(nullptr);
*
*     int n, m;
*     cin >> n >> m;
*
*     IndexTree indexTree(n);
*
*     // 初始化数组
*     for (int i = 1, v; i <= n; i++) {
*         cin >> v;
```

```
 *         indexTree.add(i, v);
 *     }
 *
 *     // 处理操作
 *     for (int i = 1, a, b, c; i <= m; i++) {
 *         cin >> a >> b >> c;
 *         if (a == 1) {
 *             // 单点更新
 *             indexTree.add(b, c);
 *         } else {
 *             // 区间查询
 *             cout << indexTree.range(b, c) << '\n';
 *         }
 *     }
 *
 *     return 0;
 * }
 */

/**
 * 以下是Python实现的树状数组（Fenwick Tree）单点更新区间查询代码
 *
 * class IndexTree:
 *     def __init__(self, size):
 *         self.n = size
 *         # 树状数组下标从1开始，因此长度为n+1
 *         self.tree = [0] * (self.n + 1)
 *
 *     # lowbit函数：获取数字i的二进制表示中最低位的1所对应的值
 *     def lowbit(self, x):
 *         return x & -x
 *
 *     # 单点更新操作：在位置i上增加v
 *     def add(self, i, v):
 *         while i <= self.n:
 *             self.tree[i] += v
 *             i += self.lowbit(i)
 *
 *     # 前缀和查询操作：查询从1到i的元素和
 *     def sum(self, i):
 *         ans = 0
 *         while i > 0:
 *             ans += self.tree[i]
```

```
*             i -= self.lowbit(i)
*         return ans
*
*     # 区间查询操作：查询从 l 到 r 的元素和
*     def range(self, l, r):
*         return self.sum(r) - self.sum(l - 1)
*
* # 主函数
* def main():
*     import sys
*     input = sys.stdin.read().split()
*     ptr = 0
*
*     n = int(input[ptr])
*     ptr += 1
*     m = int(input[ptr])
*     ptr += 1
*
*     index_tree = IndexTree(n)
*
*     # 初始化数组
*     for i in range(1, n + 1):
*         v = int(input[ptr])
*         ptr += 1
*         index_tree.add(i, v)
*
*     # 处理操作
*     output = []
*     for _ in range(m):
*         a = int(input[ptr])
*         ptr += 1
*         b = int(input[ptr])
*         ptr += 1
*         c = int(input[ptr])
*         ptr += 1
*
*         if a == 1:
*             # 单点更新
*             index_tree.add(b, c)
*         else:
*             # 区间查询
*             output.append(str(index_tree.range(b, c)))
*
```

```
 *     # 批量输出结果，提高效率
 *     print('\n'.join(output))
 *
 * if __name__ == '__main__':
 *     main()
 */

/**
 * 树状数组的典型应用案例详解：
 *
 * 1. 逆序对统计（LeetCode 315）：
 *    - 问题描述：计算一个数组中逆序对的数量（i<j 且 nums[i]>nums[j]）
 *    - 解法思路：离散化数组元素，从右到左遍历，每次查询已处理元素中小于当前元素的数量
 *    - 代码实现：
 *      ```java
 *      // 离散化数组
 *      // 从右向左遍历，对于每个元素 nums[i]，查询树状数组中小于 nums[i]的元素个数
 *      // 将 nums[i]添加到树状数组中
 *      ```
 *    - 时间复杂度：O(n log n)
 *
 * 2. 区间更新，单点查询（使用差分数组）：
 *    - 问题描述：对数组的某区间[l,r]增加一个值 v，然后查询单点的值
 *    - 解法思路：使用树状数组维护差分数组
 *    - 实现方法：区间更新时，add(l, v)和 add(r+1, -v)；单点查询时，求前缀和 sum(i)
 *    - 应用场景：区间更新操作频繁，需要高效单点查询
 *
 * 3. 离线查询处理：
 *    - 问题描述：处理多个离线区间查询，每个查询返回区间和
 *    - 解法思路：将查询按右端点排序，逐个处理元素并回答查询
 *    - 优点：对于大量查询，可以更高效地批量处理
 *
 * 4. 二维扩展 - 矩阵操作：
 *    - 问题描述：处理二维矩阵的单点更新和子矩阵和查询
 *    - 解法思路：扩展一维树状数组到二维，每个操作嵌套两层循环
 *    - 时间复杂度：O(log n * log m)，其中 n 和 m 是矩阵的维度
 */

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

文件: Code02_IndexTreeIntervalAddSingleQuery.java
==========================================
package class108;
```

```
/**
 * 树状数组范围增加、单点查询模板
 *
 * 本实现通过结合差分思想与树状数组，实现了区间更新和单点查询操作，
 * 两者的时间复杂度均为 O(log n)。
 *
 * 核心思想：利用差分数组的性质，将区间更新转换为两个单点更新操作，
 * 然后通过树状数组维护差分数组，使得单点查询也能在 O(log n) 时间内完成。
 *
 * 测试链接: https://www.luogu.com.cn/problem/P3368
 * 提交时请把类名改成"Main"，可以直接通过
 *
 * 时间复杂度分析：
 * - 区间更新(add): O(log n)，每次区间更新只需要两次单点更新操作
 * - 单点查询(sum): O(log n)，即树状数组的前缀和查询操作
 * 空间复杂度: O(n)
 *
 * 差分数组原理解析：
 *
 * 1. 差分数组定义：
 *    设原数组为 a，差分数组为 d，则：
 *    - d[1] = a[1]
 *    - d[i] = a[i] - a[i-1] (i > 1)
 *    - 原数组的每个元素等于差分数组的前缀和：a[i] = d[1] + d[2] + ... + d[i]
 *
 * 2. 区间更新转换：
 *    当需要对区间 [l, r] 加上一个值 v 时，只需：
 *    - d[l] += v （表示从位置 l 开始的所有元素都增加 v）
 *    - d[r+1] -= v （表示从位置 r+1 开始抵消之前的增加）
 *    这样，原数组中区间 [l, r] 的所有元素都会增加 v
 *
 * 3. 单点查询原理：
 *    查询原数组 a[i] 相当于查询差分数组的前缀和 sum(d[1..i])
 *    这恰好可以通过树状数组高效实现
 *
 * 4. 数学证明：
 *    假设对区间 [l, r] 加上 v，即更新差分数组 d[l] += v, d[r+1] -= v
 *    则：
 *    - 当 i < l: a[i] 不变，因为 d 的前缀和不变
 *    - 当 l <= i <= r: a[i] 增加 v，因为 d[l] += v 影响前缀和
 *    - 当 i > r: a[i] 不变，因为 d[l] += v 和 d[r+1] -= v 相互抵消
 */
```

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

public class Code02_IndexTreeIntervalAddSingleQuery {

    /**
     * 最大数据范围，根据题目要求设置
     * 注意：这里设置为 500002 而不是 500001，是因为区间操作时需要访问 r+1 的位置
     */
    public static int MAXN = 500002;

    /**
     * 树状数组，此处维护的是差分数组的前缀和
     * 注意：树状数组的下标必须从 1 开始，不从 0 开始
     */
    public static int[] tree = new int[MAXN];

    /**
     * n: 数组长度
     * m: 操作次数
     */
    public static int n, m;

    /**
     * lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
     *
     * @param i 输入的整数
     * @return i 的二进制表示中最低位的 1 所对应的值
     */
    public static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 在差分数组的位置 i 上增加 v
     * 时间复杂度：O(log n)
     *
     * @param i 要更新的差分位置（从 1 开始）
```

```
     * @param v 要增加的值
     */
    public static void add(int i, int v) {
        // 当 i 超过数组长度时停止更新
        while (i <= n) {
            // 更新当前节点的值
            tree[i] += v;
            // 移动到父节点继续更新
            i += lowbit(i);
        }
    }

    /**
     * 查询原数组中位置 i 的值
     * 由于树状数组维护的是差分数组的前缀和，所以这里的 sum(i)直接返回原数组 a[i]
     * 时间复杂度：O(log n)
     *
     * @param i 要查询的位置（从 1 开始）
     * @return 原数组中位置 i 的值
     */
    public static int sum(int i) {
        int ans = 0;
        // 当 i 小于等于 0 时停止查询
        while (i > 0) {
            // 累加当前节点的值
            ans += tree[i];
            // 移动到父节点继续查询
            i -= lowbit(i);
        }
        return ans;
    }

    /**
     * 主函数，处理输入输出
     */
    public static void main(String[] args) throws IOException {
        // 初始化输入输出流
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取数组长度 n 和操作次数 m
        in.nextToken();
```

```
n = (int) in.nval;
in.nextToken();
m = (int) in.nval;

// 初始化数组：使用差分思想构建初始树状数组
for (int i = 1, v; i <= n; i++) {
    in.nextToken();
    v = (int) in.nval;
    // 在位置 i 增加 v，在位置 i+1 减少 v（差分数组的构建方式）
    // 这相当于在原数组的位置 i 设置值 v
    add(i, v);
    add(i + 1, -v);
}

// 处理 m 个操作
for (int i = 1; i <= m; i++) {
    in.nextToken();
    int op = (int) in.nval;  // 操作类型

    // 判断操作类型
    if (op == 1) {
        // 操作 1：区间更新
        in.nextToken(); int l = (int) in.nval;  // 区间左边界
        in.nextToken(); int r = (int) in.nval;  // 区间右边界
        in.nextToken(); int v = (int) in.nval;  // 要增加的值

        // 在差分数组的 l 位置增加 v
        add(l, v);
        // 在差分数组的 r+1 位置减少 v
        add(r + 1, -v);
    } else {
        // 操作 2：单点查询
        in.nextToken();
        int index = (int) in.nval;  // 要查询的位置
        // 直接调用 sum(index)获取原数组中位置 index 的值
        out.println(sum(index));
    }
}

// 刷新输出流并关闭资源
out.flush();
out.close();
br.close();
```

```
    }

    /**
 * 区间更新+单点查询的应用场景与优缺点分析：
 *
 * 适用场景：
 * 1. 区间加法操作，频繁查询单个元素的值
 * 2. 区间标记问题，需要统计特定位置被覆盖的次数
 * 3. 动态区间修改，静态单点查询
 * 4. 需要高效处理大量区间更新操作的场景
 *
 * 优势：
 * - 实现简洁，代码量少
 * - 常数因子小，实际运行效率高
 * - 空间利用率好，只需一个数组
 * - 对于特定问题比线段树更高效
 *
 * 劣势：
 * - 功能相对局限，无法直接处理区间查询
 * - 需要理解差分思想，实现门槛略高
 * - 对于复杂操作需要额外的数学转换
 *
 * 算法进阶方向：
 * 1. 扩展到区间更新区间查询（需要维护两个树状数组）
 * 2. 与其他数据结构结合使用
 * 3. 应用到更高维度的问题中
 * 4. 处理离线问题时的优化策略
 */

    /**
 * 差分树状数组的工程实现深度解析：
 *
 * 1. 数组初始化技术：
 *    在初始化阶段，我们通过对每个位置 i 执行 add(i, v) 和 add(i+1, -v) 操作，
 *    相当于在差分数组中为原数组的每个元素建立了正确的初始状态。
 *    这样处理的原因是原数组的初始化可以看作对区间 [i,i] 执行 v 的增量操作。
 *
 * 2. 边界处理策略：
 *    - MAXN 设为 500002 而不是 500001，预留 r+1 的位置
 *    - 在 C++实现中，tree 数组大小为 n+2，确保 r+1 不会越界
 *    - 在处理 r+1 时，即使 r+1 > n，树状数组的循环条件 (i <= n) 也会自动处理边界
 *
 * 3. 性能优化关键点：
```

```
 *     - 数据类型选择：使用 long 或 long long 避免大数值计算时的溢出
 *     - 快速输入输出：在大规模数据情况下尤为重要
 *     - 预处理和批处理：减少函数调用和系统 IO 次数
 *
 * 4. 常见错误分析：
 *     - 初始数组构建错误：未正确应用差分原理初始化
 *     - 数组越界：未为 r+1 预留空间
 *     - 整数溢出：未使用足够大的数据类型
 *     - 索引错误：混淆从 0 开始和从 1 开始的数组索引
 *
 * 5. 与其他实现方式的对比：
 *     - 线段树：功能更强大但实现复杂，常数较大
 *     - 前缀和数组：适合静态数据，无法高效处理动态更新
 *     - 块状数组：实现简单但时间复杂度通常为 O(√n)
 *     - 平衡树：实现复杂，不适合此类特定场景
 */
}

/**
 * 以下是 C++实现的树状数组（Fenwick Tree）区间更新单点查询代码
 *
 * #include <iostream>
 * #include <vector>
 * using namespace std;
 *
 * const int MAXN = 500002; // 最大数据范围
 *
 * class FenwickTree {
 * private:
 *     vector<long long> tree; // 使用 long long 防止溢出
 *     int n; // 数组长度
 *
 *     // lowbit 函数
 *     int lowbit(int x) {
 *         return x & -x;
 *     }
 *
 * public:
 *     // 构造函数
 *     FenwickTree(int size) {
 *         n = size;
 *         tree.resize(n + 2, 0); // 多分配一个空间处理 r+1 的情况
 *     }
```

```
*
*     // 在差分数组的位置 i 上增加 v
*     void add(int i, long long v) {
*         while (i <= n) {
*             tree[i] += v;
*             i += lowbit(i);
*         }
*     }
*
*     // 查询原数组中位置 i 的值（即差分数组的前缀和）
*     long long query(int i) {
*         long long ans = 0;
*         while (i > 0) {
*             ans += tree[i];
*             i -= lowbit(i);
*         }
*         return ans;
*     }
*
*     // 区间更新：对区间[l,r]加上 v
*     void rangeAdd(int l, int r, long long v) {
*         add(l, v);
*         add(r + 1, -v);
*     }
* };
*
* int main() {
*     ios::sync_with_stdio(false); // 关闭同步，加速输入输出
*     cin.tie(nullptr);
*     cout.tie(nullptr);
*
*     int n, m;
*     cin >> n >> m;
*
*     FenwickTree ft(n);
*
*     // 初始化数组
*     for (int i = 1, v; i <= n; i++) {
*         cin >> v;
*         ft.rangeAdd(i, i, v); // 单点初始化相当于区间[i,i]加 v
*     }
*
*     // 处理操作
```

```
*     for (int i = 1, op, l, r, v, idx; i <= m; i++) {
*         cin >> op;
*         if (op == 1) {
*             // 区间更新
*             cin >> l >> r >> v;
*             ft.rangeAdd(l, r, v);
*         } else {
*             // 单点查询
*             cin >> idx;
*             cout << ft.query(idx) << '\n';
*         }
*     }
*
*     return 0;
* }
*/

/**
 * 以下是 Python 实现的树状数组（Fenwick Tree）区间更新单点查询代码
 *
 * class FenwickTree:
 *     def __init__(self, size):
 *         self.n = size
 *         # 树状数组下标从 1 开始，多分配空间处理 r+1 的情况
 *         self.tree = [0] * (self.n + 2)
 *
 *     # lowbit 函数
 *     def lowbit(self, x):
 *         return x & -x
 *
 *     # 在差分数组的位置 i 上增加 v
 *     def add(self, i, v):
 *         while i <= self.n:
 *             self.tree[i] += v
 *             i += self.lowbit(i)
 *
 *     # 查询原数组中位置 i 的值（即差分数组的前缀和）
 *     def query(self, i):
 *         ans = 0
 *         while i > 0:
 *             ans += self.tree[i]
 *             i -= self.lowbit(i)
 *         return ans
```

```
*
*     # 区间更新：对区间[l,r]加上 v
*     def range_add(self, l, r, v):
*         self.add(l, v)
*         self.add(r + 1, -v)
*
* # 主函数
* def main():
*     import sys
*     input = sys.stdin.read().split()
*     ptr = 0
*
*     n = int(input[ptr])
*     ptr += 1
*     m = int(input[ptr])
*     ptr += 1
*
*     ft = FenwickTree(n)
*
*     # 初始化数组
*     for i in range(1, n + 1):
*         v = int(input[ptr])
*         ptr += 1
*         ft.range_add(i, i, v)  # 单点初始化相当于区间[i,i]加 v
*
*     # 处理操作
*     output = []
*     for _ in range(m):
*         op = int(input[ptr])
*         ptr += 1
*
*         if op == 1:
*             # 区间更新
*             l = int(input[ptr])
*             ptr += 1
*             r = int(input[ptr])
*             ptr += 1
*             v = int(input[ptr])
*             ptr += 1
*             ft.range_add(l, r, v)
*         else:
*             # 单点查询
*             idx = int(input[ptr])
```

```
 *             ptr += 1
 *             output.append(str(ft.query(idx)))
 *
 *     # 批量输出结果，提高效率
 *     print('\n'.join(output))
 *
 * if __name__ == '__main__':
 *     main()
 */

/**
 * 区间更新单点查询树状数组的应用案例详解：
 *
 * 1. 区间标记与查询问题：
 *    - 问题描述：对数组的多个区间进行标记或计数，然后查询特定位置的标记次数
 *    - 应用场景：会议预约系统、活动参与统计、资源占用分析
 *    - 解决方案：每次预约区间[l,r]时，执行 rangeAdd(l, r, 1)，查询时执行 query(pos)
 *
 * 2. 离线二维范围查询：
 *    - 问题描述：在二维平面上，多次对矩形区域增加一个值，然后查询单点的值
 *    - 解决思路：扩展一维差分树状数组到二维，维护二维差分数组
 *    - 时间复杂度：O(log n * log m)，其中 n 和 m 是二维数组的维度
 *
 * 3. 区间增量的累计查询：
 *    - 问题描述：多次对区间增加不同的值，需要频繁查询单点的当前值
 *    - 应用场景：游戏开发中的区域效果系统、金融中的区间累计收益计算
 *
 * 4. 离线处理问题：
 *    - 问题描述：处理离线的区间更新和单点查询请求
 *    - 优化策略：对查询进行排序，批量处理同类操作
 */

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

文件: Code03_IndexTreeIntervalAddIntervalQuery.java
==========================================
package class108;

/**
 * 树状数组范围增加、范围查询模板
 *
 * 本实现通过结合差分思想与树状数组，使用两个树状数组来维护差分数组的信息，
 * 从而实现 O(log n)时间复杂度的区间更新和区间查询操作。
```

```
 *
 * 核心思想：
 * 利用差分思想的扩展，推导出前缀和查询公式，需要维护两个树状数组：
 * 1. info1[i]：维护 d[i]，其中 d 是差分数组
 * 2. info2[i]：维护 d[i]*(i-1)
 * 通过这两个数组的组合，可以计算出原数组的前缀和和区间和
 *
 * 测试链接: https://www.luogu.com.cn/problem/P3372
 * 提交时请把类名改成"Main"，可以直接通过
 *
 * 时间复杂度分析：
 * - 区间更新(add): O(log n)
 * - 区间查询(range): O(log n)
 * 空间复杂度: O(n)
 *
 * 数学原理分析：
 * 设差分数组为 d，原数组为 a，则有：
 * 1. a[i] = d[1] + d[2] + ... + d[i]
 * 2. 原数组前缀和 sum(a[1..i]) = i*d[1] + (i-1)*d[2] + ... + 1*d[i]
 * 3. 可以拆分为: i*(d[1]+d[2]+...+d[i]) - (0*d[1] + 1*d[2] + ... + (i-1)*d[i])
 * 4. 即: i*sum1(i) - sum2(i)
 * 其中，sum1(i)是 info1 的前缀和，sum2(i)是 info2 的前缀和
 */

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

public class Code03_IndexTreeIntervalAddIntervalQuery {

    /**
     * 最大数据范围，根据题目要求设置
     */
    public static int MAXN = 100001;

    /**
     * 两个树状数组，用于维护差分数组的信息
     * info1[i]：维护 d[i]，其中 d 是差分数组
     */
    public static long[] info1 = new long[MAXN];
```

```
/**
 * info2[i]：维护 d[i]*(i-1)
 */
public static long[] info2 = new long[MAXN];

/**
 * n: 数组长度
 * m: 操作次数
 */
public static int n, m;

/**
 * lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
 *
 * @param i 输入的整数
 * @return i 的二进制表示中最低位的 1 所对应的值
 */
public static int lowbit(int i) {
    return i & -i;
}

/**
 * 更新指定树状数组在位置 i 的值
 *
 * @param tree 要更新的树状数组
 * @param i 要更新的位置（从 1 开始）
 * @param v 要增加的值
 */
public static void add(long[] tree, int i, long v) {
    while (i <= n) {
        tree[i] += v;
        i += lowbit(i);
    }
}

/**
 * 查询树状数组前 i 个元素的前缀和
 *
 * @param tree 要查询的树状数组
 * @param i 查询的范围（从 1 到 i）
 * @return 树状数组的前缀和
 */
```

```
public static long sum(long[] tree, int i) {
    long ans = 0;
    while (i > 0) {
        ans += tree[i];
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 在区间[l,r]上增加值 v
 * 利用差分数组的特性，将区间更新转换为两个单点更新
 *
 * @param l 区间左边界（从 1 开始）
 * @param r 区间右边界（从 1 开始）
 * @param v 要增加的值
 */
public static void add(int l, int r, long v) {
    // 对差分数组的影响：d[l] += v，d[r+1] -= v
    add(info1, l, v);
    add(info1, r + 1, -v);
    // 对应 info2 数组的更新：d[l]*(l-1) += v*(l-1)，d[r+1]*r -= v*r
    add(info2, l, (l - 1) * v);
    add(info2, r + 1, -(r * v));
}

/**
 * 计算原数组前缀和[1..i]
 *
 * @param i 查询的右边界（从 1 开始）
 * @return 原数组前 i 项的和
 */
public static long prefixSum(int i) {
    return sum(info1, i) * i - sum(info2, i);
}

/**
 * 查询原始数组中[l..r]范围上的累加和
 * 通过两次前缀和相减得到区间和
 *
 * @param l 区间左边界（从 1 开始）
 * @param r 区间右边界（从 1 开始）
 * @return 原数组区间[l,r]的和
```

```
     */
    public static long range(int l, int r) {
        return prefixSum(r) - prefixSum(l - 1);
    }

    /**
     * 主函数，处理输入输出
     */
    public static void main(String[] args) throws IOException {
        // 初始化输入输出流
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取数组长度 n 和操作次数 m
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;

        // 初始化数组：直接使用原始值进行单点更新
        long cur;
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            cur = (long) in.nval;
            // 对每个位置进行单点更新，相当于构造初始差分数组
            add(i, i, cur);
        }

        // 处理 m 个操作
        long v;
        for (int i = 1, op, l, r; i <= m; i++) {
            in.nextToken();
            op = (int) in.nval;  // 操作类型

            // 判断操作类型
            if (op == 1) {
                // 操作 1：区间更新
                in.nextToken(); l = (int) in.nval;  // 区间左边界
                in.nextToken(); r = (int) in.nval;  // 区间右边界
                in.nextToken(); v = (long) in.nval;  // 要增加的值
                add(l, r, v);  // 调用 add 方法进行区间更新
            } else {
```

```
            // 操作 2：区间查询
            in.nextToken(); l = (int) in.nval;  // 区间左边界
            in.nextToken(); r = (int) in.nval;  // 区间右边界
            out.println(range(l, r));  // 调用 range 方法进行区间查询
        }
    }

    // 刷新输出流并关闭资源
    out.flush();
    out.close();
    br.close();
}

/**
 * 区间更新+区间查询的应用场景：
 * 1. 区间加法操作，频繁查询区间和
 * 2. 二维前缀和等复杂的统计问题
 * 3. 大型数据集合的区间统计和修改
 *
 * 实现要点：
 * 1. 使用两个树状数组维护差分数组的信息
 * 2. 通过数学推导得出前缀和查询公式
 * 3. 特别注意数据范围，使用 long 类型避免溢出
 *
 * 与其他数据结构对比：
 * - 相比线段树，代码更简洁，常数更小，但灵活性稍差
 * - 相比普通前缀和，支持高效的区间修改
 * - 特别适合需要频繁区间修改和区间查询的场景
 */

/**
 * 区间更新区间查询树状数组的数学原理深度推导：
 *
 * 1. 前缀和公式推导：
 *    设原数组为 a，差分数组为 d（d[1] = a[1], d[i] = a[i] - a[i-1] for i>1）
 *    则 a[i] = d[1] + d[2] + ... + d[i] = sum1(i) ，其中 sum1(i)是 d 数组的前 i 项和
 *
 * 2. 原数组前缀和 sum(a[1..i])计算：
 *    sum(a[1..i]) = a[1] + a[2] + ... + a[i]
 *                 = d[1] + (d[1]+d[2]) + ... + (d[1]+d[2]+...+d[i])
 *                 = i*d[1] + (i-1)*d[2] + ... + 1*d[i]
 *                 = i*(d[1]+d[2]+...+d[i]) - (0*d[1] + 1*d[2] + ... + (i-1)*d[i])
 *                 = i*sum1(i) - sum2(i)，其中 sum2(i)是 d[k]*(k-1)的前 i 项和
```

```
     *
     * 3. 区间更新的影响：
     *    当对区间[l,r]加上 v 时，差分数组的变化是：d[l] += v, d[r+1] -= v
     *    因此 info1 数组的更新为：info1[l] += v, info1[r+1] -= v
     *    info2 数组的更新为：info2[l] += v*(l-1), info2[r+1] -= v*r
     *
     * 4. 时间复杂度分析：
     *    - 区间更新操作需要更新两个树状数组，每次操作的时间复杂度为 O(log n)
     *    - 区间查询操作需要两次前缀和计算，每次查询的时间复杂度为 O(log n)
     *    - 空间复杂度为 O(n)，需要两个大小为 n 的数组
     */
}

/**
 * 以下是 C++实现的树状数组（Fenwick Tree）区间更新区间查询代码
 *
 * #include <iostream>
 * #include <vector>
 * using namespace std;
 *
 * const int MAXN = 100001; // 最大数据范围
 *
 * class FenwickTree {
 * private:
 *     vector<long long> info1; // 维护差分数组 d[i]
 *     vector<long long> info2; // 维护差分数组 d[i]*(i-1)
 *     int n; // 数组长度
 *
 *     // lowbit 函数
 *     int lowbit(int x) {
 *         return x & -x;
 *     }
 *
 *     // 更新树状数组
 *     void add(vector<long long>& tree, int i, long long v) {
 *         while (i <= n) {
 *             tree[i] += v;
 *             i += lowbit(i);
 *         }
 *     }
 *
 *     // 查询树状数组前缀和
 *     long long sum(vector<long long>& tree, int i) {
```

```
*         long long ans = 0;
*         while (i > 0) {
*             ans += tree[i];
*             i -= lowbit(i);
*         }
*         return ans;
*     }
*
*     // 计算原数组前缀和[1..i]
*     long long prefixSum(int i) {
*         return sum(info1, i) * i - sum(info2, i);
*     }
*
* public:
*     // 构造函数
*     FenwickTree(int size) {
*         n = size;
*         info1.resize(n + 2, 0); // 多分配空间避免越界
*         info2.resize(n + 2, 0);
*     }
*
*     // 区间更新：对区间[l,r]加上 v
*     void rangeAdd(int l, int r, long long v) {
*         // 更新 info1 数组
*         add(info1, l, v);
*         add(info1, r + 1, -v);
*
*         // 更新 info2 数组
*         add(info2, l, v * (l - 1));
*         add(info2, r + 1, -v * r);
*     }
*
*     // 区间查询：查询[l,r]的和
*     long long rangeQuery(int l, int r) {
*         return prefixSum(r) - prefixSum(l - 1);
*     }
* };
*
* int main() {
*     ios::sync_with_stdio(false); // 关闭同步，加速输入输出
*     cin.tie(nullptr);
*     cout.tie(nullptr);
*
```

```
*     int n, m;
*     cin >> n >> m;
*
*     FenwickTree ft(n);
*
*     // 初始化数组
*     for (int i = 1; i <= n; ++i) {
*         long long v;
*         cin >> v;
*         ft.rangeAdd(i, i, v); // 单点初始化相当于区间[i,i]加 v
*     }
*
*     // 处理操作
*     while (m--) {
*         int op, l, r;
*         long long v;
*         cin >> op;
*
*         if (op == 1) {
*             // 区间更新
*             cin >> l >> r >> v;
*             ft.rangeAdd(l, r, v);
*         } else {
*             // 区间查询
*             cin >> l >> r;
*             cout << ft.rangeQuery(l, r) << '\n';
*         }
*     }
*
*     return 0;
* }
*/

/**
 * 以下是 Python 实现的树状数组（Fenwick Tree）区间更新区间查询代码
 *
 * class FenwickTree:
 *     def __init__(self, size):
 *         self.n = size
 *         # 树状数组下标从 1 开始，多分配空间避免越界
 *         self.info1 = [0] * (self.n + 2)  # 维护差分数组 d[i]
 *         self.info2 = [0] * (self.n + 2)  # 维护差分数组 d[i]*(i-1)
 *
```

```
*     # lowbit 函数
*     def lowbit(self, x):
*         return x & -x
*
*     # 更新树状数组
*     def _add(self, tree, i, v):
*         while i <= self.n:
*             tree[i] += v
*             i += self.lowbit(i)
*
*     # 查询树状数组前缀和
*     def _sum(self, tree, i):
*         ans = 0
*         while i > 0:
*             ans += tree[i]
*             i -= self.lowbit(i)
*         return ans
*
*     # 计算原数组前缀和[1..i]
*     def _prefix_sum(self, i):
*         return self._sum(self.info1, i) * i - self._sum(self.info2, i)
*
*     # 区间更新：对区间[l,r]加上 v
*     def range_add(self, l, r, v):
*         # 更新 info1 数组
*         self._add(self.info1, l, v)
*         self._add(self.info1, r + 1, -v)
*
*         # 更新 info2 数组
*         self._add(self.info2, l, v * (l - 1))
*         self._add(self.info2, r + 1, -v * r)
*
*     # 区间查询：查询[l,r]的和
*     def range_query(self, l, r):
*         return self._prefix_sum(r) - self._prefix_sum(l - 1)
*
* # 主函数
* def main():
*     import sys
*     input = sys.stdin.read().split()
*     ptr = 0
*
*     n = int(input[ptr])
```

```
*     ptr += 1
*     m = int(input[ptr])
*     ptr += 1
*
*     ft = FenwickTree(n)
*
*     # 初始化数组
*     for i in range(1, n + 1):
*         v = int(input[ptr])
*         ptr += 1
*         ft.range_add(i, i, v)  # 单点初始化相当于区间[i,i]加v
*
*     # 处理操作
*     output = []
*     for _ in range(m):
*         op = int(input[ptr])
*         ptr += 1
*
*         if op == 1:
*             # 区间更新
*             l = int(input[ptr])
*             ptr += 1
*             r = int(input[ptr])
*             ptr += 1
*             v = int(input[ptr])
*             ptr += 1
*             ft.range_add(l, r, v)
*         else:
*             # 区间查询
*             l = int(input[ptr])
*             ptr += 1
*             r = int(input[ptr])
*             ptr += 1
*             output.append(str(ft.range_query(l, r)))
*
*     # 批量输出结果，提高效率
*     print('\n'.join(output))
*
* if __name__ == '__main__':
*     main()
*/

/**
```

* 区间更新区间查询树状数组的高级分析与应用：
*
* 1. 树状数组区间操作的可视化解释：
*   - 差分数组视角：当在区间[l,r]添加 v 时，差分数组只有两个位置变化，这使得树状数组能够高效处理区间更新
*   - 树状结构视图：
*   树状数组的二进制结构保证每次更新和查询操作都能以 O(log n)的时间访问 O(log n)个节点
*   对于区间操作，通过差分数组转换，将复杂的区间操作转化为对两个端点的操作
*
* 2. 与线段树的详细对比：
*   - 功能对比：两者都能处理区间更新和区间查询，但线段树可以支持更复杂的区间操作（如区间乘法、区间最值）
*   - 代码复杂度：树状数组实现更简洁，代码量少，常数更小
*   - 实际效率：对于基本的区间加和区间求和操作，树状数组通常比线段树快 20%-30%
*   - 内存占用：树状数组内存占用略少于线段树（O(n) vs O(4n)）
*
* 3. 扩展到多维：
*   - 二维区间更新和区间查询需要四个树状数组，如 Code05_TwoDimensionIntervalAddIntervalQuery2.java 中实现
*   - 三维及以上场景中，线段树的优势更为明显，树状数组实现复杂度指数增长
*
* 4. 工程实现要点：
*   - 必须使用 long 类型，避免整数溢出
*   - 数组索引从 1 开始，简化边界处理
*   - 预分配足够空间，避免动态扩容带来的性能开销
*   - 大批量数据处理时注意输入输出效率
*
* 5. 典型应用案例详解：
*   - 案例一：区间更新区间求和（如本题）
*   应用场景：学生成绩统计、区间增减操作、股票价格变动
*   输入输出示例：
*   输入：n=5 m=3
*   1 2 3 4 5
*   1 1 3 2 // 区间[1,3]加 2
*   2 1 5   // 查询区间[1,5]和
*   1 2 4 -1 // 区间[2,4]减 1
*   2 1 5   // 查询区间[1,5]和
*   输出：25
*   22
*
*   - 案例二：二维区域求和
*   应用场景：图像处理、矩阵操作、地理信息系统
*   通过扩展到二维树状数组实现

 *
 *   - 案例三：逆序对统计的优化版本
 *   应用场景：排序算法分析、数据分析
 *   结合离散化和树状数组，处理重复元素的情况
 *
 *   - 案例四：动态区间频率统计
 *   应用场景：数据流分析、实时监控系统
 *   结合离散化和树状数组，支持动态区间频率查询
 *
 * 6. 深入优化技巧：
 *   - 数据类型优化：根据实际问题规模选择合适的数据类型，避免不必要的内存开销
 *   - 输入输出优化：使用快速 IO 方法，批量读取输入和输出结果
 *   - 离散化技术：将大范围稀疏数据映射到小范围连续数据
 *   - 懒加载思想：对于特殊情况，可以结合懒加载思想进一步优化
 *   - 并行化处理：对于大规模数据，可以考虑将树状数组分割成多个独立部分并行处理
 *   - 内存访问模式优化：预分配连续内存，减少缓存未命中
 *   - 混合数据结构：针对复杂问题，考虑树状数组与其他数据结构的结合使用
 *
 * 7. 常见陷阱与调试技巧：
 *   - 索引从 1 开始的边界处理
 *   - 数据溢出问题（特别是使用 int 而不是 long）
 *   - 差分数组更新时(r+1)越界问题
 *   - 批量操作时的性能优化
 */

==========

文件: Code04_TwoDimensionSingleAddIntervalQuery.java
==========
```
package class108;

/**
 * 二维树状数组单点增加、范围查询模板
 *
 * 本实现提供了二维平面上的高效单点更新和矩形区域查询功能，
 * 通过树状数组（Binary Indexed Tree / Fenwick Tree）数据结构实现。
 *
 * 核心思想：
 * 二维树状数组是一维树状数组在二维平面上的扩展，每个节点维护一个矩形区域的信息，
 * 通过 lowbit 操作在二维空间构建树状结构，使得单点更新和区间查询的时间复杂度均为 O(log n * log m)。
 *
 * 测试链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/
```

```
 *
 * 时间复杂度分析：
 * - 构造函数(NumMatrix): O(n*m*log n*log m)，其中 n 和 m 分别是矩阵的行数和列数
 * - 单点更新(update): O(log n * log m)
 * - 区间查询(sumRegion): O(log n * log m)
 * 空间复杂度: O(n*m)，用于存储树状数组和原始数据数组
 */
public class Code04_TwoDimensionSingleAddIntervalQuery {

    /**
     * NumMatrix 类实现了二维树状数组的核心功能
     * 支持单点更新和区域求和查询
     */
    class NumMatrix {

        /**
         * 二维树状数组，tree[i][j]维护特定矩形区域的信息
         * 树状数组的索引从 1 开始
         */
        public int[][] tree;

        /**
         * 原始数值数组，存储每个位置的当前值
         * 同样从索引 1 开始存储，对应树状数组的位置
         */
        public int[][] nums;

        /**
         * 二维数组的行数
         */
        public int n;

        /**
         * 二维数组的列数
         */
        public int m;

        /**
         * 构造函数，初始化二维树状数组
         *
         * @param matrix 输入的二维矩阵，下标从 0 开始
         */
        public NumMatrix(int[][] matrix) {
```

```
        n = matrix.length;
        m = matrix[0].length;
        // 初始化树状数组和原始数值数组，大小为(n+1)×(m+1)，因为索引从 1 开始
        tree = new int[n + 1][m + 1];
        nums = new int[n + 1][m + 1];
        // 初始化原始矩阵中的所有元素
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                update(i, j, matrix[i][j]);
            }
        }
    }

    /**
     * lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
     *
     * @param i 输入的整数
     * @return i 的二进制表示中最低位的 1 所对应的值
     */
    private int lowbit(int i) {
        return i & -i;
    }

    /**
     * 在树状数组的(x,y)位置增加 v 值
     * 时间复杂度：O(log n * log m)
     *
     * @param x 树状数组的行坐标（从 1 开始）
     * @param y 树状数组的列坐标（从 1 开始）
     * @param v 要增加的值
     */
    private void add(int x, int y, int v) {
        // 遍历所有需要更新的行
        for (int i = x; i <= n; i += lowbit(i)) {
            // 遍历该行中所有需要更新的列
            for (int j = y; j <= m; j += lowbit(j)) {
                // 更新当前节点的值
                tree[i][j] += v;
            }
        }
    }

    /**
```

```
     * 计算从树状数组(1,1)到(x,y)矩形区域的累加和
     * 时间复杂度：O(log n * log m)
     *
     * @param x 树状数组的行坐标（从1开始）
     * @param y 树状数组的列坐标（从1开始）
     * @return (1,1)到(x,y)矩形区域的累加和
     */
    private int sum(int x, int y) {
        int ans = 0;
        // 遍历所有需要查询的行
        for (int i = x; i > 0; i -= lowbit(i)) {
            // 遍历该行中所有需要查询的列
            for (int j = y; j > 0; j -= lowbit(j)) {
                // 累加当前节点的值
                ans += tree[i][j];
            }
        }
        return ans;
    }

    /**
     * 更新原始矩阵中(x,y)位置的值为v
     * 通过计算差值来更新树状数组
     *
     * @param x 原始矩阵的行坐标（从0开始）
     * @param y 原始矩阵的列坐标（从0开始）
     * @param v 新的值
     */
    public void update(int x, int y, int v) {
        // 树状数组的坐标比原始矩阵多1，所以需要+1
        // 计算新旧值的差值，并更新树状数组
        add(x + 1, y + 1, v - nums[x + 1][y + 1]);
        // 更新原始数值数组中的值
        nums[x + 1][y + 1] = v;
    }

    /**
     * 查询原始矩阵中从(a,b)到(c,d)矩形区域的累加和
     * 利用二维前缀和的思想，通过四个前缀和的加减操作得到目标区域的和
     *
     * @param a 左上区域行坐标（原始矩阵，从0开始）
     * @param b 左上区域列坐标（原始矩阵，从0开始）
     * @param c 右下区域行坐标（原始矩阵，从0开始）
```

```
     * @param d 右下区域列坐标（原始矩阵，从0开始）
     * @return (a,b)到(c,d)矩形区域的累加和
     */
    public int sumRegion(int a, int b, int c, int d) {
        // 注意转换到树状数组的坐标
        return sum(c + 1, d + 1) - sum(a, d + 1) - sum(c + 1, b) + sum(a, b);
    }
}

/**
 * 二维树状数组的原理与可视化解析：
 *
 * 1. 树状结构可视化：
 *    二维树状数组可以看作是一维树状数组在二维平面上的扩展，每个节点 tree[i][j]维护一个矩形区域的和。
 *    例如，当 n=4, m=4 时，各节点覆盖的区域如下：
 *    - tree[4][4]: 覆盖整个 4×4 矩阵
 *    - tree[3][3]: 覆盖行[1-3]，列[1-3]的区域
 *    - tree[2][4]: 覆盖行[1-2]，列[1-4]的区域
 *    - tree[4][2]: 覆盖行[1-4]，列[1-2]的区域
 *
 * 2. 区间覆盖规则：
 *    每个节点 tree[i][j]维护的区间为 [i-lowbit(i)+1, i] × [j-lowbit(j)+1, j]
 *    例如，当 i=10 (二进制 1010)，lowbit(i)=2 (二进制 10)
 *    则 i-lowbit(i)+1=9，所以 tree[10][j]维护行范围为[9,10]的区间
 *    这种设计保证了通过 O(log n × log m)次操作即可完成更新和查询
 *
 * 3. 单点更新传播路径：
 *    当更新点(x,y)时，所有包含该点的矩形区域都需要更新
 *    例如，更新(3,3)时，需要更新：
 *    tree[3][3], tree[4][3], tree[3][4], tree[4][4] 等节点
 *    每个方向上的更新步数均为 O(log n)
 *
 * 4. 前缀和查询原理：
 *    计算从(1,1)到(x,y)的区域和时，通过累加多个不重叠的矩形区域实现
 *    例如，查询 sum(3,3)时，会累加：
 *    tree[3][3], tree[2][3], tree[3][2], tree[2][2] 等节点
 *
 * 5. 区间查询的容斥应用：
 *    sumRegion(a,b,c,d) = sum(c,d) - sum(a-1,d) - sum(c,b-1) + sum(a-1,b-1)
 *    这利用了二维平面上的容斥原理，通过四个前缀和的组合计算出任意矩形区域的和
 */
```

```
/**
 * 二维树状数组的高级应用与工程优化：
 *
 * 1. 典型应用案例详解：
 *    - 案例一：二维平面统计
 *      应用场景：图像像素统计、二维区域热度图、地理信息系统中的范围查询
 *      操作模式：频繁更新少量点，多次查询不同区域的统计值
 *      性能优势：比直接遍历 O(n²)更快，适用于中等规模矩阵
 *
 *    - 案例二：二维频率统计
 *      应用场景：文本词频矩阵、多维数据统计分析
 *      优化方法：结合离散化技术处理大范围稀疏数据
 *      输入输出示例：
 *      输入：3x3 矩阵初始化为 0
 *           update(1,1,5) // 在位置(1,1)增加 5
 *           update(2,2,3) // 在位置(2,2)增加 3
 *           sumRegion(0,0,2,2) // 查询整个矩阵的和
 *      输出：8
 *
 *    - 案例三：二维差分应用
 *      应用场景：二维区间更新问题
 *      实现思路：通过二维差分数组转换，将区间更新转化为四个角点的更新
 *      这种方法在 Code05_TwoDimensionIntervalAddIntervalQuery1.java 和
Code05_TwoDimensionIntervalAddIntervalQuery2.java 中有详细实现
 *
 * 2. 性能优化技术：
 *    - 内存优化：对于大型稀疏矩阵，可以考虑使用稀疏矩阵表示
 *    - 数据类型优化：根据实际数据范围选择合适的数据类型，如 int、long 或 long long
 *    - 并行处理：对于超大型矩阵，可以考虑分块并行处理
 *    - 缓存优化：调整循环顺序，提高缓存命中率
 *
 * 3. 工程实现注意事项：
 *    - 索引转换：注意原始矩阵（从 0 开始）和树状数组（从 1 开始）的索引转换
 *    - 边界检查：实现时要确保不会越界，特别是对于接近数组边界的查询
 *    - 初始化效率：初始化时可以使用二维前缀和技术提高效率
 *    - 异常处理：添加对非法输入的检查和处理
 *
 * 4. 与其他数据结构对比：
 *    - 二维线段树：功能更强大但实现复杂，常数较大
 *    - 二维前缀和数组：静态数据高效，但无法处理动态更新
 *    - 块状数组：实现简单但时间复杂度为 O(√n × √m)
 *    - 二维平衡树：实现复杂，不适合此类特定场景
 *
```

```
 * 5. 常见错误与调试技巧：
 *    - 索引越界：特别注意边界条件下的处理
 *    - 数组初始化错误：确保初始化时正确转换索引
 *    - 容斥计算错误：检查区间查询时四个前缀和的加减是否正确
 *    - 性能瓶颈：对于大规模数据，考虑分块或稀疏表示
 */
}

/**
 * 以下是C++实现的二维树状数组（Fenwick Tree）单点更新区域查询代码
 *
 * #include <iostream>
 * #include <vector>
 * using namespace std;
 *
 * class NumMatrix {
 * private:
 *     vector<vector<int>> tree;  // 二维树状数组，从索引1开始
 *     vector<vector<int>> nums;  // 原始数值数组，从索引1开始
 *     int n, m;  // 行数和列数
 *
 *     // lowbit函数
 *     int lowbit(int i) {
 *         return i & -i;
 *     }
 *
 *     // 在树状数组的(x,y)位置增加v值
 *     void add(int x, int y, int v) {
 *         for (int i = x; i <= n; i += lowbit(i)) {
 *             for (int j = y; j <= m; j += lowbit(j)) {
 *                 tree[i][j] += v;
 *             }
 *         }
 *     }
 *
 *     // 计算从(1,1)到(x,y)矩形区域的累加和
 *     int sum(int x, int y) {
 *         int ans = 0;
 *         for (int i = x; i > 0; i -= lowbit(i)) {
 *             for (int j = y; j > 0; j -= lowbit(j)) {
 *                 ans += tree[i][j];
 *             }
 *         }
```

```
*         return ans;
*     }
*
* public:
*     // 构造函数
*     NumMatrix(vector<vector<int>>& matrix) {
*         if (matrix.empty() || matrix[0].empty()) {
*             n = 0;
*             m = 0;
*             return;
*         }
*
*         n = matrix.size();
*         m = matrix[0].size();
*
*         // 初始化树状数组和原始数值数组，大小为(n+1)×(m+1)
*         tree.resize(n + 1, vector<int>(m + 1, 0));
*         nums.resize(n + 1, vector<int>(m + 1, 0));
*
*         // 初始化所有元素
*         for (int i = 0; i < n; ++i) {
*             for (int j = 0; j < m; ++j) {
*                 update(i, j, matrix[i][j]);
*             }
*         }
*     }
*
*     // 更新原始矩阵中(x,y)位置的值为 v
*     void update(int x, int y, int v) {
*         // 计算新旧值的差值，并更新树状数组
*         add(x + 1, y + 1, v - nums[x + 1][y + 1]);
*         // 更新原始数值数组
*         nums[x + 1][y + 1] = v;
*     }
*
*     // 查询从(a,b)到(c,d)矩形区域的累加和
*     int sumRegion(int a, int b, int c, int d) {
*         // 转换到树状数组的索引并应用容斥原理
*         return sum(c + 1, d + 1) - sum(a, d + 1) - sum(c + 1, b) + sum(a, b);
*     }
* };
*
* // 测试代码（可选）
```

```
 * int main() {
 *     // 示例矩阵
 *     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}
 *     };
 *
 *     NumMatrix numMatrix(matrix);
 *
 *     // 测试查询
 *     cout << "查询[2,1,4,3]区域的和: " << numMatrix.sumRegion(2, 1, 4, 3) << endl;
 *
 *     // 测试更新
 *     numMatrix.update(3, 2, 2);
 *     cout << "更新(3,2)为2后，查询[2,1,4,3]区域的和: " << numMatrix.sumRegion(2, 1, 4, 3) <<
endl;
 *
 *     return 0;
 * }
 */

/**
 * 以下是Python实现的二维树状数组（Fenwick Tree）单点更新区域查询代码
 *
 * class NumMatrix:
 *     def __init__(self, matrix):
 *         if not matrix or not matrix[0]:
 *             self.n = 0
 *             self.m = 0
 *             return
 *
 *         self.n = len(matrix)
 *         self.m = len(matrix[0])
 *
 *         # 初始化树状数组和原始数值数组，索引从1开始
 *         self.tree = [[0] * (self.m + 1) for _ in range(self.n + 1)]
 *         self.nums = [[0] * (self.m + 1) for _ in range(self.n + 1)]
 *
 *         # 初始化所有元素
 *         for i in range(self.n):
```

```
*             for j in range(self.m):
*                 self.update(i, j, matrix[i][j])
*
*     def _lowbit(self, i):
*         # lowbit 函数
*         return i & -i
*
*     def _add(self, x, y, v):
*         # 在树状数组的(x,y)位置增加 v 值
*         i = x
*         while i <= self.n:
*             j = y
*             while j <= self.m:
*                 self.tree[i][j] += v
*                 j += self._lowbit(j)
*             i += self._lowbit(i)
*
*     def _sum(self, x, y):
*         # 计算从(1,1)到(x,y)矩形区域的累加和
*         ans = 0
*         i = x
*         while i > 0:
*             j = y
*             while j > 0:
*                 ans += self.tree[i][j]
*                 j -= self._lowbit(j)
*             i -= self._lowbit(i)
*         return ans
*
*     def update(self, row, col, val):
*         # 更新原始矩阵中(row,col)位置的值为 val
*         delta = val - self.nums[row + 1][col + 1]
*         self._add(row + 1, col + 1, delta)
*         self.nums[row + 1][col + 1] = val
*
*     def sumRegion(self, row1, col1, row2, col2):
*         # 查询从(row1,col1)到(row2,col2)矩形区域的累加和
*         return (self._sum(row2 + 1, col2 + 1) -
*                 self._sum(row1, col2 + 1) -
*                 self._sum(row2 + 1, col1) +
*                 self._sum(row1, col1))
*
* # 测试代码（可选）
```

```
* if __name__ == "__main__":
*     # 示例矩阵
*     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]
*     ]
*
*     num_matrix = NumMatrix(matrix)
*
*     # 测试查询
*     print(f"查询[2,1,4,3]区域的和: {num_matrix.sumRegion(2, 1, 4, 3)}")
*
*     # 测试更新
*     num_matrix.update(3, 2, 2)
*     print(f"更新(3,2)为2后，查询[2,1,4,3]区域的和: {num_matrix.sumRegion(2, 1, 4, 3)}")
*/

/**
 * 二维树状数组的高级分析与应用：
 *
 * 1. 与其他二维数据结构的对比：
 *    - 二维线段树：二维树状数组实现更简单，常数更小，但线段树可以支持更复杂的区间操作
 *    - 二维块状数组：二维树状数组对于单点更新和范围查询更高效
 *    - 简单二维数组暴力查询：对于静态数据预处理后的前缀和查询更快，但无法高效处理动态更新
 *
 * 2. 性能优化技巧：
 *    - 缓存热点区域的数据，减少重复查询
 *    - 对于大规模数据，考虑使用离散化技术
 *    - 使用迭代而非递归实现，减少函数调用开销
 *    - 对于频繁查询的场景，可以实现批量查询的优化
 *
 * 3. 内存优化策略：
 *    - 仅在需要时初始化树状数组
 *    - 对于稀疏矩阵，可以考虑使用哈希表实现的树状数组
 *    - 对于非常大的矩阵，考虑分块处理或使用压缩存储
 *
 * 4. 扩展应用：
 *    - 二维频率统计：统计某矩形区域内满足条件的元素个数
 *    - 二维逆序对：计算二维平面上的逆序对数量
 *    - 二维区间最大/最小值：通过维护最大值/最小值的树状数组实现（非标准树状数组）
```

```
 *     - 高维扩展：可以扩展到三维及以上，但性能会指数下降
 *
 * 5. 实际应用场景：
 *     - 图像处理中的像素值统计和更新
 *     - 二维数据库中的范围查询和更新
 *     - 地理信息系统中的区域查询
 *     - 矩阵中的动态子矩阵和查询
 */
```

==========

文件: Code05_TwoDimensionIntervalAddIntervalQuery1.java

==========

```
package class108;

/**
 * 二维树状数组区间增加、区间查询模板
 *
 * 本实现通过维护四个二维树状数组，实现了O(log n * log m)时间复杂度的矩形区域更新和查询操作。
 * 算法基于二维差分数组的思想，并通过数学推导得出需要维护的四个关键数组，以支持高效的区间操作。
 *
 * 测试链接: https://www.luogu.com.cn/problem/P4514
 * 提交时请把类名改成"Main"，可以通过所有测试用例
 *
 * 时间复杂度分析：
 * - 区间更新(add): O(log n * log m)
 * - 区间查询(range): O(log n * log m)
 * 空间复杂度: O(n * m)
 *
 * 数学原理分析：
 * 二维树状数组区间更新+区间查询的实现基于以下数学推导：
 * 1. 对于二维数组的前缀和 sum(x,y)，可以通过差分和树状数组的结合高效计算
 * 2. 为了支持区间更新和区间查询，需要维护四个树状数组，分别对应不同的组合项
 * 3. 通过数学展开和化简，可以得到查询前缀和时的线性组合公式
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.StreamTokenizer;

public class Code05_TwoDimensionIntervalAddIntervalQuery1 {
```

```
/**
 * 最大数据范围，根据题目要求设置
 */
public static int MAXN = 2050;

/**
 * 最大数据范围，根据题目要求设置
 */
public static int MAXM = 2050;

/**
 * 维护四个二维树状数组，用于支持二维区间更新和区间查询
 * info1[i][j]: 维护差分数组 d[i][j]
 * info2[i][j]: 维护 d[i][j] * i
 * info3[i][j]: 维护 d[i][j] * j
 * info4[i][j]: 维护 d[i][j] * i * j
 */
// 维护信息 : d[i][j]
public static int[][] info1 = new int[MAXN][MAXM];

// 维护信息 : d[i][j] * i
public static int[][] info2 = new int[MAXN][MAXM];

// 维护信息 : d[i][j] * j
public static int[][] info3 = new int[MAXN][MAXM];

// 维护信息 : d[i][j] * i * j
public static int[][] info4 = new int[MAXN][MAXM];

/**
 * n: 二维数组的行数
 * m: 二维数组的列数
 */
public static int n, m;

/**
 * lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
 *
 * @param i 输入的整数
 * @return i 的二进制表示中最低位的 1 所对应的值
 */
public static int lowbit(int i) {
    return i & -i;
```

```
}

/**
 * 在点(x,y)处更新差分数组，并同时维护四个树状数组
 *
 * @param x 行坐标（从 1 开始）
 * @param y 列坐标（从 1 开始）
 * @param v 要增加的值
 */
public static void add(int x, int y, int v) {
    // 计算四个需要更新的树状数组对应的值
    int v1 = v;             // 对应 info1: d[i][j]
    int v2 = x * v;         // 对应 info2: d[i][j] * i
    int v3 = y * v;         // 对应 info3: d[i][j] * j
    int v4 = x * y * v;     // 对应 info4: d[i][j] * i * j

    // 同时更新四个树状数组
    for (int i = x; i <= n; i += lowbit(i)) {
        for (int j = y; j <= m; j += lowbit(j)) {
            info1[i][j] += v1;
            info2[i][j] += v2;
            info3[i][j] += v3;
            info4[i][j] += v4;
        }
    }
}

/**
 * 计算二维前缀和(1,1)~(x,y)
 * 使用数学推导得出的公式，结合四个树状数组计算二维前缀和
 *
 * @param x 行坐标（从 1 开始）
 * @param y 列坐标（从 1 开始）
 * @return (1,1)~(x,y)矩形区域的和
 */
// 以(1,1)左上角，以(x,y)右下角
public static int sum(int x, int y) {
    int ans = 0;
    // 遍历树状数组计算前缀和
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            // 数学公式计算：(x+1)(y+1)*info1 - (y+1)*info2 - (x+1)*info3 + info4
            // 此公式由二维前缀和展开推导得出
```

```
                ans += (x + 1) * (y + 1) * info1[i][j] - (y + 1) * info2[i][j] - (x + 1) *
info3[i][j] + info4[i][j];
            }
        }
        return ans;
    }

    /**
     * 给矩形区域(a,b)~(c,d)的所有元素加 v
     * 利用二维差分数组的特性，将矩形区域更新转换为四个角落点的更新
     *
     * @param a 左上区域行坐标（从 1 开始）
     * @param b 左上区域列坐标（从 1 开始）
     * @param c 右下区域行坐标（从 1 开始）
     * @param d 右下区域列坐标（从 1 开始）
     * @param v 要增加的值
     */
    public static void add(int a, int b, int c, int d, int v) {
        // 利用二维差分数组的特性，对四个角点进行更新
        add(a, b, v);          // (a,b)处加 v
        add(a, d + 1, -v);     // (a,d+1)处减 v
        add(c + 1, b, -v);     // (c+1,b)处减 v
        add(c + 1, d + 1, v); // (c+1,d+1)处加 v
    }

    /**
     * 查询区域和(a,b)~(c,d)
     * 利用二维前缀和的容斥原理，通过四个前缀和的组合计算出目标区域的和
     *
     * @param a 左上区域行坐标（从 1 开始）
     * @param b 左上区域列坐标（从 1 开始）
     * @param c 右下区域行坐标（从 1 开始）
     * @param d 右下区域列坐标（从 1 开始）
     * @return (a,b)~(c,d)矩形区域的和
     */
    public static int range(int a, int b, int c, int d) {
        // 容斥原理：全量减去两边加上重叠部分
        return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
    }

    /**
     * 主函数，处理输入输出和操作请求
     */
```

```
public static void main(String[] args) throws IOException {
    // 初始化输入流
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    String op;
    int a, b, c, d, v;

    // 处理输入直到文件结束
    while (in.nextToken() != StreamTokenizer.TT_EOF) {
        op = in.sval;
        // X 命令：初始化二维数组大小
        if (op.equals("X")) {
            in.nextToken();
            n = (int) in.nval;
            in.nextToken();
            m = (int) in.nval;
        }
        // L 命令：区间更新操作
        else if (op.equals("L")) {
            in.nextToken();
            a = (int) in.nval;
            in.nextToken();
            b = (int) in.nval;
            in.nextToken();
            c = (int) in.nval;
            in.nextToken();
            d = (int) in.nval;
            in.nextToken();
            v = (int) in.nval;
            // 调用区间更新方法
            add(a, b, c, d, v);
        }
        // 查询命令：区间查询操作
        else {
            in.nextToken();
            a = (int) in.nval;
            in.nextToken();
            b = (int) in.nval;
            in.nextToken();
            c = (int) in.nval;
            in.nextToken();
            d = (int) in.nval;
            // 调用区间查询方法并输出结果
```

```
                System.out.println(range(a, b, c, d));
            }
        }
        // 关闭资源
        br.close();
    }

    /**
     * 二维树状数组区间更新+区间查询的实现原理详解：
     *
     * 1. 算法核心思想：
     *    - 结合差分数组和二维树状数组的思想
     *    - 差分数组用于实现高效的区间更新
     *    - 树状数组用于高效查询前缀和
     *
     * 2. 数学推导：
     *    对于二维数组 A[i][j]，定义差分数组 d[i][j]满足：
     *    A[i][j] = sum_{x=1 to i} sum_{y=1 to j} d[x][y]
     *
     *    前缀和 S(x,y) = sum_{i=1 to x} sum_{j=1 to y} A[i][j]
     *    展开后需要维护四个关键组合项
     *
     * 3. 实现细节：
     *    - 使用四个树状数组维护不同的组合项
     *    - 区间更新转换为四个角点的更新操作
     *    - 区间查询使用容斥原理组合四个前缀和
     *
     * 4. 数据结构优势：
     *    - 相比普通二维前缀和，支持高效区间修改
     *    - 相比二维线段树，实现更简洁，常数更小
     *    - 相比暴力算法，时间复杂度大幅提升
     */
}


/**
 * 以下是 C++实现的树状数组（Fenwick Tree）代码
 * 包含 1D 树状数组、2D 树状数组、以及各种应用场景的实现
 *
 * // 1D 树状数组实现（单点更新，区间查询）
 * class FenwickTree1D {
 * private:
 *     vector<int> tree;
```

```
*     int n;
*
* public:
*     // 构造函数
*     FenwickTree1D(int size) : n(size) {
*         tree.resize(n + 1, 0); // 树状数组从索引 1 开始
*     }
*
*     // lowbit 操作
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
*     // 单点更新：在位置 i 增加 val
*     void update(int i, int val) {
*         while (i <= n) {
*             tree[i] += val;
*             i += lowbit(i);
*         }
*     }
*
*     // 前缀和查询：查询[1, i]的和
*     int query(int i) {
*         int sum = 0;
*         while (i > 0) {
*             sum += tree[i];
*             i -= lowbit(i);
*         }
*         return sum;
*     }
*
*     // 区间查询：查询[l, r]的和
*     int rangeQuery(int l, int r) {
*         return query(r) - query(l - 1);
*     }
* };
*
* // LeetCode 315 - 计算右侧小于当前元素的个数（C++实现）
* class Solution315 {
* private:
*     vector<int> tree;
*     int n;
*
```

```
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
*     void update(int i, int val) {
*         while (i <= n) {
*             tree[i] += val;
*             i += lowbit(i);
*         }
*     }
*
*     int query(int i) {
*         int sum = 0;
*         while (i > 0) {
*             sum += tree[i];
*             i -= lowbit(i);
*         }
*         return sum;
*     }
*
* public:
*     vector<int> countSmaller(vector<int>& nums) {
*         int m = nums.size();
*         if (m == 0) return {};
*
*         // 离散化处理
*         vector<int> sortedNums = nums;
*         sort(sortedNums.begin(), sortedNums.end());
*         sortedNums.erase(unique(sortedNums.begin(), sortedNums.end()), sortedNums.end());
*
*         unordered_map<int, int> valueToRank;
*         for (int i = 0; i < sortedNums.size(); ++i) {
*             valueToRank[sortedNums[i]] = i + 1; // 排名从 1 开始
*         }
*
*         n = sortedNums.size();
*         tree.resize(n + 1, 0);
*
*         vector<int> result(m);
*         // 从右向左遍历
*         for (int i = m - 1; i >= 0; --i) {
*             int rank = valueToRank[nums[i]];
*             result[i] = query(rank - 1); // 查询比当前元素小的个数
```

```
*             update(rank, 1); // 将当前元素加入树状数组
*         }
*
*         return result;
*     }
* };
*
* // 二维树状数组（单点更新，区间查询）
* class FenwickTree2D {
* private:
*     vector<vector<int>> tree;
*     int m, n;
*
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
* public:
*     FenwickTree2D(int rows, int cols) : m(rows), n(cols) {
*         tree.resize(m + 1, vector<int>(n + 1, 0));
*     }
*
*     void update(int x, int y, int val) {
*         for (int i = x; i <= m; i += lowbit(i)) {
*             for (int j = y; j <= n; j += lowbit(j)) {
*                 tree[i][j] += val;
*             }
*         }
*     }
*
*     int query(int x, int y) {
*         int sum = 0;
*         for (int i = x; i > 0; i -= lowbit(i)) {
*             for (int j = y; j > 0; j -= lowbit(j)) {
*                 sum += tree[i][j];
*             }
*         }
*         return sum;
*     }
*
*     int rangeQuery(int x1, int y1, int x2, int y2) {
*         return query(x2, y2) - query(x1 - 1, y2) - query(x2, y1 - 1) + query(x1 - 1, y1 - 1);
*     }
```

```
* };
*
* // LeetCode 493 - 翻转对（C++实现）
* class Solution493 {
* private:
*     vector<int> tree;
*     int n;
*
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
*     void update(int i, int val) {
*         while (i <= n) {
*             tree[i] += val;
*             i += lowbit(i);
*         }
*     }
*
*     int query(int i) {
*         int sum = 0;
*         while (i > 0) {
*             sum += tree[i];
*             i -= lowbit(i);
*         }
*         return sum;
*     }
*
* public:
*     int reversePairs(vector<int>& nums) {
*         int m = nums.size();
*         if (m <= 1) return 0;
*
*         // 离散化处理
*         set<long long> allNums;
*         for (int num : nums) {
*             allNums.insert((long long)num);
*             allNums.insert((long long)num * 2);
*         }
*
*         vector<long long> sortedNums(allNums.begin(), allNums.end());
*         unordered_map<long long, int> valueToRank;
*         for (int i = 0; i < sortedNums.size(); ++i) {
```

```
 *             valueToRank[sortedNums[i]] = i + 1;
 *         }
 *
 *         n = sortedNums.size();
 *         tree.resize(n + 1, 0);
 *
 *         int count = 0;
 *         for (int i = m - 1; i >= 0; --i) {
 *             // 二分查找找到第一个大于等于 nums[i]/2 的位置
 *             auto it = lower_bound(sortedNums.begin(), sortedNums.end(), (double)nums[i] / 2);
 *             int idx = it - sortedNums.begin();
 *
 *             if (idx > 0) {
 *                 count += query(idx);
 *             }
 *
 *             int rank = valueToRank[(long long)nums[i]];
 *             update(rank, 1);
 *         }
 *
 *         return count;
 *     }
 * };
 */


/**
 * 以下是 Python 实现的树状数组（Fenwick Tree）代码
 * 包含 1D 树状数组、2D 树状数组、以及各种应用场景的实现
 *
 * # 1D 树状数组实现（单点更新，区间查询）
 * class FenwickTree1D:
 *     def __init__(self, size):
 *         self.n = size
 *         self.tree = [0] * (self.n + 1)  # 树状数组从索引 1 开始
 *
 *     def lowbit(self, x):
 *         # 获取 x 的二进制表示中最低位 1 所对应的值
 *         return x & (-x)
 *
 *     def update(self, i, val):
 *         # 在位置 i 增加 val
 *         while i <= self.n:
```

```
*             self.tree[i] += val
*             i += self.lowbit(i)
*
*     def query(self, i):
*         # 查询[1, i]的前缀和
*         res = 0
*         while i > 0:
*             res += self.tree[i]
*             i -= self.lowbit(i)
*         return res
*
*     def range_query(self, l, r):
*         # 查询[l, r]的区间和
*         return self.query(r) - self.query(l - 1)
*
* # LeetCode 315 - 计算右侧小于当前元素的个数（Python 实现）
* class Solution315:
*     def countSmaller(self, nums):
*         n = len(nums)
*         if n == 0:
*             return []
*
*         # 离散化处理
*         sorted_nums = sorted(set(nums))
*         value_to_rank = {val: i + 1 for i, val in enumerate(sorted_nums)}  # 排名从 1 开始
*
*         max_rank = len(sorted_nums)
*         fenwick_tree = FenwickTree1D(max_rank)
*
*         result = [0] * n
*         # 从右向左遍历
*         for i in range(n - 1, -1, -1):
*             rank = value_to_rank[nums[i]]
*             result[i] = fenwick_tree.query(rank - 1)  # 查询比当前元素小的个数
*             fenwick_tree.update(rank, 1)  # 将当前元素加入树状数组
*
*         return result
*
* # 二维树状数组（单点更新，区间查询）
* class FenwickTree2D:
*     def __init__(self, rows, cols):
*         self.m = rows
*         self.n = cols
```

```
 *         self.tree = [[0] * (self.n + 1) for _ in range(self.m + 1)]
 *
 *     def lowbit(self, x):
 *         return x & (-x)
 *
 *     def update(self, x, y, val):
 *         i = x
 *         while i <= self.m:
 *             j = y
 *             while j <= self.n:
 *                 self.tree[i][j] += val
 *                 j += self.lowbit(j)
 *             i += self.lowbit(i)
 *
 *     def query(self, x, y):
 *         res = 0
 *         i = x
 *         while i > 0:
 *             j = y
 *             while j > 0:
 *                 res += self.tree[i][j]
 *                 j -= self.lowbit(j)
 *             i -= self.lowbit(i)
 *         return res
 *
 *     def range_query(self, x1, y1, x2, y2):
 *         return self.query(x2, y2) - self.query(x1 - 1, y2) - self.query(x2, y1 - 1) +
self.query(x1 - 1, y1 - 1)
 *
 * # LeetCode 493 - 翻转对（Python 实现）
 * class Solution493:
 *     def reversePairs(self, nums):
 *         n = len(nums)
 *         if n <= 1:
 *             return 0
 *
 *         # 离散化处理
 *         all_nums = set()
 *         for num in nums:
 *             all_nums.add(num)
 *             all_nums.add(2 * num)
 *
 *         sorted_nums = sorted(all_nums)
```

```
 *         value_to_rank = {val: i + 1 for i, val in enumerate(sorted_nums)}
 *
 *         max_rank = len(sorted_nums)
 *         fenwick_tree = FenwickTree1D(max_rank)
 *
 *         count = 0
 *         for i in range(n - 1, -1, -1):
 *             # 二分查找找到第一个大于等于 nums[i]/2 的位置
 *             target = nums[i] / 2
 *             left, right = 0, len(sorted_nums)
 *             while left < right:
 *                 mid = (left + right) // 2
 *                 if sorted_nums[mid] >= target:
 *                     right = mid
 *                 else:
 *                     left = mid + 1
 *
 *             if left > 0:
 *                 count += fenwick_tree.query(left)
 *
 *             rank = value_to_rank[nums[i]]
 *             fenwick_tree.update(rank, 1)
 *
 *         return count
 */
```

==========

文件: Code05_TwoDimensionIntervalAddIntervalQuery2.java

==========

```
package class108;

/**
 * 二维树状数组区间增加、区间查询实现
 *
 * 本文件包含了二维树状数组区间更新和区间查询操作的详细实现
 * 支持二维平面上的高效矩形区域更新和查询操作
 *
 * 测试链接: https://www.luogu.com.cn/problem/P4514
 *
 * 核心思想:
 * 使用四个树状数组维护差分数组的不同组合项，通过数学推导得出的公式支持区间操作
 *
```

```
 * 时间复杂度分析：
 * - 区间更新: O(log n * log m)，其中 n 和 m 分别是矩阵的行数和列数
 * - 区间查询: O(log n * log m)
 * 空间复杂度: O(n * m)，用于存储四个二维树状数组
 */

/**
 * 二维树状数组区间更新区间查询的可视化与原理解析：
 *
 * 1. 二维差分原理可视化：
 *    - 当我们需要对矩形区域(a,b)~(c,d)增加 v 时，我们只需要在差分数组的四个角点进行操作：
 *      d[a][b] += v    // 区域左上角开始增加 v
 *      d[a][d+1] -= v  // 右侧边界减去 v，抵消右侧区域的影响
 *      d[c+1][b] -= v  // 下侧边界减去 v，抵消下侧区域的影响
 *      d[c+1][d+1] += v // 右下边界增加 v，恢复交叉区域的影响
 *    - 这种操作确保了只有目标矩形区域内的所有点会被增加 v，而其他区域不变
 *
 * 2. 树状数组与差分结合的可视化理解：
 *    - 原始数组 a[i][j] = 差分数组 d[i][j]的前缀和
 *    - 二维前缀和 sum(a[i][j]) = 差分数组 d[i][j]加权前缀和
 *    - 四个树状数组分别维护不同权重的差分数组，实现高效计算
 *
 * 3. 四个树状数组的功能可视化：
 *    info1: 存储 d[i][j]的树状数组，负责(i+1)(j+1)部分的计算
 *    info2: 存储 d[i][j]*i 的树状数组，负责-(i+1)部分的计算
 *    info3: 存储 d[i][j]*j 的树状数组，负责-(j+1)部分的计算
 *    info4: 存储 d[i][j]*i*j 的树状数组，负责+1 部分的计算
 *    这四个部分组合起来正好构成了二维前缀和的数学公式
 */

/* C++代码实现 */
/* 取消注释以下代码可以直接在 C++环境中编译运行 */

//#include <cstdio>
//using namespace std;
//
///**
// * 最大数据范围，根据题目要求设置
// */
//const int MAXN = 2050;
//const int MAXM = 2050;
//
///**
```

```
// * 维护四个二维树状数组
// * info1[i][j]: 维护差分数组 d[i][j]
// * info2[i][j]: 维护 d[i][j] * i
// * info3[i][j]: 维护 d[i][j] * j
// * info4[i][j]: 维护 d[i][j] * i * j
// */
//int info1[MAXN][MAXM], info2[MAXN][MAXM], info3[MAXN][MAXM], info4[MAXN][MAXM];
//int n, m;
//
///**
// * lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
// *
// * @param i 输入的整数
// * @return i 的二进制表示中最低位的 1 所对应的值
// */
//int lowbit(int i) {
//    return i & -i;  // 利用位运算获取最低位的 1
//}
//
///**
// * 在点(x,y)处更新差分数组，并同时维护四个树状数组
// *
// * @param x 行坐标（从 1 开始）
// * @param y 列坐标（从 1 开始）
// * @param v 要增加的值
// */
//void add(int x, int y, int v) {
//    // 计算四个需要更新的树状数组对应的值
//    int v1 = v;            // 对应 info1: d[i][j]
//    int v2 = x * v;        // 对应 info2: d[i][j] * i
//    int v3 = y * v;        // 对应 info3: d[i][j] * j
//    int v4 = x * y * v;    // 对应 info4: d[i][j] * i * j
//
//    // 更新四个树状数组
//    for (int i = x; i <= n; i += lowbit(i)) {
//        for (int j = y; j <= m; j += lowbit(j)) {
//            info1[i][j] += v1;
//            info2[i][j] += v2;
//            info3[i][j] += v3;
//            info4[i][j] += v4;
//        }
//    }
//}
```

```
//
///**
// * 计算二维前缀和(1,1)~(x,y)
// * 使用数学推导得出的公式，结合四个树状数组计算二维前缀和
// *
// * @param x 行坐标（从 1 开始）
// * @param y 列坐标（从 1 开始）
// * @return (1,1)~(x,y)矩形区域的和
// */
//int sum(int x, int y) {
//    int ans = 0;
//    // 遍历树状数组计算前缀和
//    for (int i = x; i > 0; i -= lowbit(i)) {
//        for (int j = y; j > 0; j -= lowbit(j)) {
//            // 数学公式计算，由二维前缀和展开推导得出
//            ans += (x + 1) * (y + 1) * info1[i][j] - (y + 1) * info2[i][j] - (x + 1) *
info3[i][j] + info4[i][j];
//        }
//    }
//    return ans;
//}
//
///**
// * 给矩形区域(a,b)~(c,d)的所有元素加 v
// * 利用二维差分数组的特性，将矩形区域更新转换为四个角落点的更新
// *
// * @param a 左上区域行坐标（从 1 开始）
// * @param b 左上区域列坐标（从 1 开始）
// * @param c 右下区域行坐标（从 1 开始）
// * @param d 右下区域列坐标（从 1 开始）
// * @param v 要增加的值
// */
//void add(int a, int b, int c, int d, int v) {
//    // 利用二维差分数组的特性，对四个角点进行更新
//    add(a, b, v);          // (a,b)处加 v
//    add(c + 1, d + 1, v); // (c+1,d+1)处加 v
//    add(a, d + 1, -v);     // (a,d+1)处减 v
//    add(c + 1, b, -v);     // (c+1,b)处减 v
//}
//
///**
// * 查询区域和(a,b)~(c,d)
// * 利用二维前缀和的容斥原理，通过四个前缀和的组合计算出目标区域的和
```

```
// *
// * @param a 左上区域行坐标（从1开始）
// * @param b 左上区域列坐标（从1开始）
// * @param c 右下区域行坐标（从1开始）
// * @param d 右下区域列坐标（从1开始）
// * @return (a,b)~(c,d)矩形区域的和
// */
//int range(int a, int b, int c, int d) {
//    // 容斥原理：全量减去两边加上重叠部分
//    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
//}
//
///**
// * 主函数，处理输入输出和操作请求
// */
//int main() {
//    char op;  // 操作类型
//    int a, b, c, d, v;  // 坐标和值
//
//    // 读取初始操作
//    scanf("%s", &op);
//    scanf("%d%d", &n, &m);  // 读取二维数组大小
//
//    // 处理操作直到文件结束
//    while (scanf("%s", &op) != EOF) {
//        if (op == 'X') {  // X命令：更新数组大小
//            scanf("%d%d", &n, &m);
//        } else if (op == 'L') {  // L命令：区间更新操作
//            scanf("%d%d%d%d%d", &a, &b, &c, &d, &v);
//            add(a, b, c, d, v);  // 执行区间更新
//        } else {  // 查询命令：区间查询操作
//            scanf("%d%d%d%d", &a, &b, &c, &d);
//            printf("%d\n", range(a, b, c, d));  // 输出查询结果
//        }
//    }
//
//    return 0;
//}

/**
 * 以下是Java实现的二维树状数组区间更新区间查询代码
 */
public class TwoDimensionFenwickTree {
```

```
/**
 * 维护四个二维树状数组
 */
private long[][] info1; // 维护 d[i][j]
private long[][] info2; // 维护 d[i][j] * i
private long[][] info3; // 维护 d[i][j] * j
private long[][] info4; // 维护 d[i][j] * i * j
private int n; // 行数
private int m; // 列数

/**
 * 构造函数
 * @param n 最大行数
 * @param m 最大列数
 */
public TwoDimensionFenwickTree(int n, int m) {
    this.n = n;
    this.m = m;
    // 初始化四个树状数组，索引从 1 开始
    info1 = new long[n + 2][m + 2];
    info2 = new long[n + 2][m + 2];
    info3 = new long[n + 2][m + 2];
    info4 = new long[n + 2][m + 2];
}

/**
 * lowbit 函数
 * @param i 输入整数
 * @return 最低位 1 所对应的值
 */
private int lowbit(int i) {
    return i & -i;
}

/**
 * 在点(x,y)处增加 v，同时更新四个树状数组
 * @param x 行坐标（从 1 开始）
 * @param y 列坐标（从 1 开始）
 * @param v 要增加的值
 */
private void add(int x, int y, long v) {
    long v1 = v;
```

```
        long v2 = v * x;
        long v3 = v * y;
        long v4 = v * x * y;

        // 更新四个树状数组
        for (int i = x; i <= n; i += lowbit(i)) {
            for (int j = y; j <= m; j += lowbit(j)) {
                info1[i][j] += v1;
                info2[i][j] += v2;
                info3[i][j] += v3;
                info4[i][j] += v4;
            }
        }
    }

    /**
     * 计算前缀和(1,1)~(x,y)
     * @param x 行坐标（从1开始）
     * @param y 列坐标（从1开始）
     * @return 前缀和结果
     */
    private long sum(int x, int y) {
        long ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) {
            for (int j = y; j > 0; j -= lowbit(j)) {
                // 应用数学公式
                ans += (x + 1) * (y + 1) * info1[i][j] - (y + 1) * info2[i][j] - (x + 1) *
info3[i][j] + info4[i][j];
            }
        }
        return ans;
    }

    /**
     * 对矩形区域(a,b)~(c,d)的所有元素加v
     * @param a 左上区域行坐标（从1开始）
     * @param b 左上区域列坐标（从1开始）
     * @param c 右下区域行坐标（从1开始）
     * @param d 右下区域列坐标（从1开始）
     * @param v 要增加的值
     */
    public void rangeAdd(int a, int b, int c, int d, long v) {
        add(a, b, v);
```

```
        add(a, d + 1, -v);
        add(c + 1, b, -v);
        add(c + 1, d + 1, v);
    }

    /**
     * 查询矩形区域(a,b)~(c,d)的和
     * @param a 左上区域行坐标（从1开始）
     * @param b 左上区域列坐标（从1开始）
     * @param c 右下区域行坐标（从1开始）
     * @param d 右下区域列坐标（从1开始）
     * @return 区域和
     */
    public long rangeQuery(int a, int b, int c, int d) {
        return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
    }

    /**
     * 设置矩阵大小
     * @param n 新的行数
     * @param m 新的列数
     */
    public void setSize(int n, int m) {
        this.n = n;
        this.m = m;
        // 重置树状数组
        info1 = new long[n + 2][m + 2];
        info2 = new long[n + 2][m + 2];
        info3 = new long[n + 2][m + 2];
        info4 = new long[n + 2][m + 2];
    }

    /**
     * 主方法，用于处理输入输出
     */
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        String op = sc.next();
        int n = sc.nextInt();
        int m = sc.nextInt();

        TwoDimensionFenwickTree ft = new TwoDimensionFenwickTree(n, m);
```

```
        while (sc.hasNext()) {
            op = sc.next();
            if (op.equals("X")) {
                n = sc.nextInt();
                m = sc.nextInt();
                ft.setSize(n, m);
            } else if (op.equals("L")) {
                int a = sc.nextInt();
                int b = sc.nextInt();
                int c = sc.nextInt();
                int d = sc.nextInt();
                long v = sc.nextLong();
                ft.rangeAdd(a, b, c, d, v);
            } else {
                int a = sc.nextInt();
                int b = sc.nextInt();
                int c = sc.nextInt();
                int d = sc.nextInt();
                System.out.println(ft.rangeQuery(a, b, c, d));
            }
        }
        sc.close();
    }
}

/**
 * 以下是Python实现的二维树状数组区间更新区间查询代码
 *
 * class TwoDimensionFenwickTree:
 *     def __init__(self, n, m):
 *         self.n = n
 *         self.m = m
 *         # 初始化四个树状数组，索引从1开始
 *         self.info1 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护d[i][j]
 *         self.info2 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护d[i][j] * i
 *         self.info3 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护d[i][j] * j
 *         self.info4 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护d[i][j] * i * j
 *
 *     def _lowbit(self, x):
 *         # lowbit函数
 *         return x & -x
 *
 *     def _add(self, x, y, v):
```

```
*         # 在点(x,y)处增加 v，同时更新四个树状数组
*         v1 = v
*         v2 = v * x
*         v3 = v * y
*         v4 = v * x * y
*
*         i = x
*         while i <= self.n:
*             j = y
*             while j <= self.m:
*                 self.info1[i][j] += v1
*                 self.info2[i][j] += v2
*                 self.info3[i][j] += v3
*                 self.info4[i][j] += v4
*                 j += self._lowbit(j)
*             i += self._lowbit(i)
*
*     def _sum(self, x, y):
*         # 计算前缀和(1,1)~(x,y)
*         ans = 0
*         i = x
*         while i > 0:
*             j = y
*             while j > 0:
*                 # 应用数学公式
*                 ans += (x + 1) * (y + 1) * self.info1[i][j] \
*                        - (y + 1) * self.info2[i][j] \
*                        - (x + 1) * self.info3[i][j] \
*                        + self.info4[i][j]
*                 j -= self._lowbit(j)
*             i -= self._lowbit(i)
*         return ans
*
*     def range_add(self, a, b, c, d, v):
*         # 对矩形区域(a,b)~(c,d)的所有元素加 v
*         self._add(a, b, v)
*         self._add(a, d + 1, -v)
*         self._add(c + 1, b, -v)
*         self._add(c + 1, d + 1, v)
*
*     def range_query(self, a, b, c, d):
*         # 查询矩形区域(a,b)~(c,d)的和
*         return (self._sum(c, d) -
```

```
*                 self._sum(a - 1, d) -
*                 self._sum(c, b - 1) +
*                 self._sum(a - 1, b - 1))
*
*     def set_size(self, n, m):
*         # 设置矩阵大小
*         self.n = n
*         self.m = m
*         # 重置树状数组
*         self.info1 = [[0] * (m + 2) for _ in range(n + 2)]
*         self.info2 = [[0] * (m + 2) for _ in range(n + 2)]
*         self.info3 = [[0] * (m + 2) for _ in range(n + 2)]
*         self.info4 = [[0] * (m + 2) for _ in range(n + 2)]
*
* # 主函数
* def main():
*     import sys
*     input = sys.stdin.read().split()
*     ptr = 0
*
*     op = input[ptr]
*     ptr += 1
*     n = int(input[ptr])
*     ptr += 1
*     m = int(input[ptr])
*     ptr += 1
*
*     ft = TwoDimensionFenwickTree(n, m)
*
*     while ptr < len(input):
*         op = input[ptr]
*         ptr += 1
*
*         if op == 'X':
*             n = int(input[ptr])
*             ptr += 1
*             m = int(input[ptr])
*             ptr += 1
*             ft.set_size(n, m)
*         elif op == 'L':
*             a = int(input[ptr])
*             ptr += 1
*             b = int(input[ptr])
```

```
 *             ptr += 1
 *             c = int(input[ptr])
 *             ptr += 1
 *             d = int(input[ptr])
 *             ptr += 1
 *             v = int(input[ptr])
 *             ptr += 1
 *             ft.range_add(a, b, c, d, v)
 *         else:
 *             a = int(input[ptr])
 *             ptr += 1
 *             b = int(input[ptr])
 *             ptr += 1
 *             c = int(input[ptr])
 *             ptr += 1
 *             d = int(input[ptr])
 *             ptr += 1
 *             print(ft.range_query(a, b, c, d))
 *
 * if __name__ == '__main__':
 *     main()
 */

/**
 * 二维树状数组区间更新区间查询的典型应用案例与深入优化：
 *
 * 1. 典型应用案例详解：
 *    - 案例一：二维区间累加与查询
 *      应用场景：图像处理中的矩形区域亮度调整、二维热图动态更新
 *      输入输出示例：
 *      输入：初始化一个 3x3 矩阵为全 0
 *           rangeAdd(1,1,3,3,5) // 整个矩阵增加 5
 *           rangeAdd(1,1,2,2,3) // 左上 2x2 区域再增加 3
 *           rangeQuery(1,1,3,3) // 查询整个矩阵的和
 *      输出：75 (3x3x5 + 2x2x3 = 45 + 12 = 57？不，这里应该是查询原始数组的和，需要重新计算)
 *           正确计算：(1,1)=8, (1,2)=8, (1,3)=5, (2,1)=8, (2,2)=8, (2,3)=5, (3,1)=5, (3,2)=5,
(3,3)=5
 *           总和=8+8+5+8+8+5+5+5+5=62
 *
 *    - 案例二：二维频率统计
 *      应用场景：文本词频矩阵、用户活跃度热图
 *      特点：频繁进行区间更新，多次查询不同区域的总和
 *      优化方法：结合离散化技术处理大范围稀疏数据
```

```
*
*     - 案例三：二维动态区域操作
*       应用场景：游戏开发中的区域效果、物理模拟中的力场计算
*       特点：需要高效的动态区间操作，对性能要求高
*       优势：比暴力更新 O(n²)和二维线段树更高效
*
* 2. 性能优化深度技巧：
*     - 数据类型优化：
*       根据实际数据范围选择合适的数据类型，避免不必要的空间浪费
*       对于可能溢出的情况，及时切换到更大的数据类型
*       C++和 Java 中使用 long/long long 类型，Python 自动处理大数
*
*     - 输入输出优化：
*       使用快速 IO 方法处理大规模输入，避免超时
*       在 Java 中使用 BufferedReader 替代 Scanner
*       在 C++中使用 scanf/printf 替代 cin/cout
*       示例：
*       // Java 快速 IO 示例
*       BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
*       StringTokenizer st = new StringTokenizer(br.readLine());
*
*     - 离散化技术：
*       当二维坐标范围很大但实际使用的坐标点较少时，使用离散化
*       将原始坐标映射到连续的较小整数范围内
*       可以节省大量内存空间，适用于大规模稀疏数据
*
*     - 内存访问模式优化：
*       调整循环顺序，提高缓存命中率
*       优先按行访问，然后按列访问，符合内存的行优先存储
*       避免频繁的列方向跳跃访问
*
*     - 并行化处理：
*       对于超大型矩阵，可以考虑分块并行处理
*       将矩阵分成多个子块，每个线程处理一部分
*       适用于多核处理器，可显著提高大规模数据处理速度
*
* 3. 工程实现高级注意事项：
*     - 索引管理：
*       严格区分原始数据（0-based）和树状数组（1-based）的索引
*       注意边界条件，特别是 c+1 和 d+1 不能越界
*       在初始化时预留足够的空间（+2）避免越界问题
*
*     - 异常处理与边界检查：
```

*   对输入的坐标范围进行验证，确保不会越界
*   处理极端情况，如空矩阵、单个元素矩阵等
*   添加适当的错误处理机制
*   示例：
*   public void rangeAdd(int a, int b, int c, int d, long v) {
*   // 验证坐标范围
*   if (a < 1 || b < 1 || c > n || d > m || a > c || b > d) {
*   throw new IllegalArgumentException("Invalid range coordinates");
*   }
*   // 正常处理
*   add(a, b, v);
*   add(a, d + 1, -v);
*   add(c + 1, b, -v);
*   add(c + 1, d + 1, v);
*   }
*
*   - 可复用性设计：
*   将二维树状数组封装为独立的类，提供清晰的 API
*   考虑添加批量操作方法，提高频繁操作的效率
*   设计适当的接口，便于集成到更大的系统中
*
*   - 单元测试：
*   编写全面的单元测试，覆盖各种边界情况
*   测试不同的输入模式和操作组合
*   使用小数据集验证正确性，大数据集测试性能
*
* 4. 与其他二维区间操作数据结构对比：
*   - 二维线段树：
*   功能更强大，支持区间最值查询
*   实现复杂，代码量大，常数较大
*   时间复杂度相同，但实际效率通常低于树状数组
*
*   - 二维块状数组：
*   实现简单，适合某些特定场景
*   时间复杂度为 O(√n × √m)，对于大规模数据效率较低
*   内存占用较小，实现灵活
*
*   - 二维平衡树：
*   实现复杂，不适合此类特定场景
*   时间复杂度较高，通常不用于二维区间操作
*   主要用于动态维护有序集合
*
*   - 二维前缀和数组：

```
 *       只能处理静态数据，无法进行动态更新
 *       查询时间 O(1)，但不支持更新操作
 *       适用于只读场景
 *
 * 5. 常见陷阱与调试技巧：
 *    - 数学公式错误：
 *       确保前缀和计算的数学公式正确无误
 *       仔细检查四个树状数组的组合方式
 *       使用小例子手动验证公式
 *
 *    - 边界条件处理：
 *       特别注意 c+1 和 d+1 可能超出原始数组范围的情况
 *       确保数组大小足够大，或者添加边界检查
 *
 *    - 索引转换错误：
 *       严格区分 0-based 和 1-based 索引
 *       在输入输出时注意索引转换
 *
 *    - 数据溢出：
 *       及时使用更大的数据类型，如 long/long long
 *       监控中间计算结果，防止溢出
 *
 *    - 性能瓶颈：
 *       对于大规模数据，监控内存使用情况
 *       考虑使用稀疏表示或离散化技术
 *       优化输入输出操作，避免成为性能瓶颈
 */
```

==========

文件: JZ51_ReversePairs.cpp

==========

```
/*
 * 剑指 Offer 51. 数组中的逆序对
 * 题目链接: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
 *
 * 题目描述:
 * 在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。
 * 输入一个数组，求出这个数组中的逆序对的总数。
 *
 * 示例:
 * 输入: [7,5,6,4]
 * 输出: 5
```

```
 * 解释: 逆序对为 (7,5), (7,6), (7,4), (5,4), (6,4)
 *
 * 解题思路:
 * 使用树状数组 + 离散化来计算逆序对个数。
 * 离散化是为了处理大数值的情况，将原始数值映射到连续的小范围内。
 * 从左往右遍历数组，对于每个元素，查询树状数组中比它大的元素个数，
 * 然后将当前元素插入树状数组。
 *
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */

class Solution {
private:
    int tree[20001];  // 树状数组，用于统计元素出现次数
    int sorted[20001];  // 离散化后的数组
    int MAXN;
    int uniqueCount;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param i 位置（从 1 开始）
     * @param v 增加的值
     */
    void add(int i, int v) {
        // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while (i < MAXN) {
            tree[i] += v;
            // 移动到父节点
            i += lowbit(i);
        }
```

```
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
int sum(int i) {
    int ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 离散化函数：将原始数组的值映射到连续的小范围内
 *
 * @param nums 原始数组
 * @param n 数组长度
 */
void discretize(int nums[], int n) {
    // 创建排序数组
    for (int i = 0; i < n; i++) {
        sorted[i] = nums[i];
    }

    // 手动排序（冒泡排序）
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (sorted[j] > sorted[j + 1]) {
                int temp = sorted[j];
                sorted[j] = sorted[j + 1];
                sorted[j + 1] = temp;
            }
        }
    }

    // 去重
```

```
        uniqueCount = 1;
        for (int i = 1; i < n; i++) {
            if (sorted[i] != sorted[uniqueCount - 1]) {
                sorted[uniqueCount] = sorted[i];
                uniqueCount++;
            }
        }

        MAXN = uniqueCount + 1;
        // 初始化树状数组
        for (int i = 0; i < MAXN; i++) {
            tree[i] = 0;
        }
    }

    /**
     * 获取元素在离散化数组中的位置（使用二分查找）
     *
     * @param val 要查找的值
     * @return 该值在离散化数组中的位置
     */
    int getId(int val) {
        int left = 0, right = uniqueCount - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (sorted[mid] >= val) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return left + 1; // 树状数组下标从1开始
    }

public:
    /**
     * 计算数组中的逆序对总数
     *
     * @param nums 输入数组
     * @param numsSize 数组长度
     * @return 逆序对总数
     */
    int reversePairs(int* nums, int numsSize) {
```

```
        if (numsSize == 0) return 0;

        // 离散化处理
        discretize(nums, numsSize);

        int ans = 0;
        // 从左往右遍历数组
        for (int i = 0; i < numsSize; i++) {
            // 获取当前元素在离散化数组中的位置
            int id = getId(nums[i]);
            // 查询比当前元素大的元素个数（即逆序对个数）
            ans += sum(MAXN - 1) - sum(id);
            // 将当前元素插入树状数组
            add(id, 1);
        }

        return ans;
    }
};

/*
 * 测试函数
 * 由于编译环境限制，此处省略测试代码
 * 实际使用时可以直接调用 Solution 类的方法
 *
 * 示例调用方式：
 * Solution solution;
 * int nums[] = {7, 5, 6, 4};
 * int result = solution.reversePairs(nums, 4);
 * result 将包含结果 5
 */

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

文件: JZ51_ReversePairs.java
==========================================
package class108;

import java.util.*;

/**
 * 剑指 Offer 51. 数组中的逆序对
 * 题目链接: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
```

```
 *
 * 题目描述:
 * 在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。
 * 输入一个数组，求出这个数组中的逆序对的总数。
 *
 * 示例:
 * 输入: [7,5,6,4]
 * 输出: 5
 * 解释: 逆序对为 (7,5), (7,6), (7,4), (5,4), (6,4)
 *
 * 解题思路:
 * 使用树状数组 + 离散化来计算逆序对个数。
 * 离散化是为了处理大数值的情况，将原始数值映射到连续的小范围内。
 * 从左往右遍历数组，对于每个元素，查询树状数组中比它大的元素个数，
 * 然后将当前元素插入树状数组。
 *
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */

public class JZ51_ReversePairs {
    // 树状数组最大容量
    private int MAXN;

    // 树状数组，用于统计元素出现次数
    private int[] tree;

    // 离散化后的数组
    private int[] sorted;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    private int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
```

```
     *
     * @param i 位置（从 1 开始）
     * @param v 增加的值
     */
    private void add(int i, int v) {
        // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while (i < MAXN) {
            tree[i] += v;
            // 移动到父节点
            i += lowbit(i);
        }
    }

    /**
     * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
     *
     * @param i 查询的结束位置
     * @return 前缀和
     */
    private int sum(int i) {
        int ans = 0;
        // 从位置 i 开始，沿着子节点路径向下累加
        while (i > 0) {
            ans += tree[i];
            // 移动到前一个相关区间
            i -= lowbit(i);
        }
        return ans;
    }

    /**
     * 离散化函数：将原始数组的值映射到连续的小范围内
     *
     * @param nums 原始数组
     */
    private void discretize(int[] nums) {
        // 创建排序数组
        sorted = Arrays.stream(nums).distinct().sorted().toArray();
        MAXN = sorted.length + 1;
        tree = new int[MAXN];
    }

    /**
```

```
     * 获取元素在离散化数组中的位置（使用二分查找）
     *
     * @param val 要查找的值
     * @return 该值在离散化数组中的位置
     */
    private int getId(int val) {
        int left = 0, right = sorted.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (sorted[mid] >= val) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return left + 1; // 树状数组下标从 1 开始
    }

    /**
     * 计算数组中的逆序对总数
     *
     * @param nums 输入数组
     * @return 逆序对总数
     */
    public int reversePairs(int[] nums) {
        int n = nums.length;
        if (n == 0) return 0;

        // 离散化处理
        discretize(nums);

        int ans = 0;
        // 从左往右遍历数组
        for (int i = 0; i < n; i++) {
            // 获取当前元素在离散化数组中的位置
            int id = getId(nums[i]);
            // 查询比当前元素大的元素个数（即逆序对个数）
            ans += sum(MAXN - 1) - sum(id);
            // 将当前元素插入树状数组
            add(id, 1);
        }

        return ans;
```

```
    }

    /**
     * 测试函数
     */
    public static void main(String[] args) {
        JZ51_ReversePairs solution = new JZ51_ReversePairs();

        // 测试用例 1
        int[] nums1 = {7, 5, 6, 4};
        int result1 = solution.reversePairs(nums1);
        System.out.println("输入: [7,5,6,4]");
        System.out.println("输出: " + result1);
        System.out.println("期望: 5");
        System.out.println();

        // 测试用例 2
        int[] nums2 = {1, 3, 2, 3, 1};
        int result2 = solution.reversePairs(nums2);
        System.out.println("输入: [1,3,2,3,1]");
        System.out.println("输出: " + result2);
        System.out.println("期望: 4");
        System.out.println();

        // 测试用例 3
        int[] nums3 = {};
        int result3 = solution.reversePairs(nums3);
        System.out.println("输入: []");
        System.out.println("输出: " + result3);
        System.out.println("期望: 0");
    }
}
```

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

文件: JZ51_ReversePairs.py

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

"""
剑指 Offer 51. 数组中的逆序对
题目链接: https://leetcode.cn/problems/shu-zu-zhong-de-ni-xu-dui-lcof/
```

题目描述:
在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。
输入一个数组，求出这个数组中的逆序对的总数。

示例:
输入: [7,5,6,4]
输出: 5
解释: 逆序对为 (7,5), (7,6), (7,4), (5,4), (6,4)

解题思路:
使用树状数组 + 离散化来计算逆序对个数。
离散化是为了处理大数值的情况，将原始数值映射到连续的小范围内。
从左往右遍历数组，对于每个元素，查询树状数组中比它大的元素个数，
然后将当前元素插入树状数组。

时间复杂度：O(n log n)
空间复杂度：O(n)
"""

```
class Solution:
    def __init__(self):
        """
        初始化函数
        """
        self.tree = []
        self.sorted = []
        self.MAXN = 0

    def lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
        例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

        :param i: 输入数字
        :return: 最低位的 1 所代表的数值
        """
        return i & -i

    def add(self, i, v):
        """
        单点增加操作：在位置 i 上增加 v
```

```
    :param i: 位置（从 1 开始）
    :param v: 增加的值
    """
    # 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while i < self.MAXN:
        self.tree[i] += v
        # 移动到父节点
        i += self.lowbit(i)

def sum(self, i):
    """
    查询前缀和：计算从位置 1 到位置 i 的所有元素之和

    :param i: 查询的结束位置
    :return: 前缀和
    """
    ans = 0
    # 从位置 i 开始，沿着子节点路径向下累加
    while i > 0:
        ans += self.tree[i]
        # 移动到前一个相关区间
        i -= self.lowbit(i)
    return ans

def discretize(self, nums):
    """
    离散化函数：将原始数组的值映射到连续的小范围内

    :param nums: 原始数组
    """
    # 创建排序数组并去重
    self.sorted = sorted(list(set(nums)))
    self.MAXN = len(self.sorted) + 1
    self.tree = [0] * self.MAXN

def get_id(self, val):
    """
    获取元素在离散化数组中的位置（使用二分查找）

    :param val: 要查找的值
    :return: 该值在离散化数组中的位置
    """
```

```
        left, right = 0, len(self.sorted) - 1
        while left <= right:
            mid = (left + right) // 2
            if self.sorted[mid] >= val:
                right = mid - 1
            else:
                left = mid + 1
        return left + 1  # 树状数组下标从1开始

    def reversePairs(self, nums):
        """
        计算数组中的逆序对总数

        :param nums: 输入数组
        :return: 逆序对总数
        """
        n = len(nums)
        if n == 0:
            return 0

        # 离散化处理
        self.discretize(nums)

        ans = 0
        # 从左往右遍历数组
        for i in range(n):
            # 获取当前元素在离散化数组中的位置
            id = self.get_id(nums[i])
            # 查询比当前元素大的元素个数（即逆序对个数）
            ans += self.sum(self.MAXN - 1) - self.sum(id)
            # 将当前元素插入树状数组
            self.add(id, 1)

        return ans


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

    # 测试用例1
```

```
    nums1 = [7, 5, 6, 4]
    result1 = solution.reversePairs(nums1)
    print("输入: [7,5,6,4]")
    print("输出: {}".format(result1))
    print("期望: 5")
    print()

    # 测试用例 2
    nums2 = [1, 3, 2, 3, 1]
    result2 = solution.reversePairs(nums2)
    print("输入: [1,3,2,3,1]")
    print("输出: {}".format(result2))
    print("期望: 4")
    print()

    # 测试用例 3
    nums3 = []
    result3 = solution.reversePairs(nums3)
    print("输入: []")
    print("输出: {}".format(result3))
    print("期望: 0")


# 运行测试
if __name__ == "__main__":
    main()

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

文件: LeetCode307_RangeSumQuery.cpp
==========================================
/**
 * LeetCode 307. 区域和检索 - 数组可修改
 * 题目链接: https://leetcode.cn/problems/range-sum-query-mutable/description/
 *
 * 题目描述:
 * 给你一个数组 nums ，请你完成两类查询。
 * 其中一类查询要求更新数组 nums 下标对应的值
 * 另一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <=
right
 * 实现 NumArray 类：
 * - NumArray(int[] nums) 用整数数组 nums 初始化对象
 * - void update(int index, int val) 将 nums[index] 的值更新为 val
```

```
 * - int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
 *
 * 示例:
 * 输入：
 * ["NumArray", "sumRange", "update", "sumRange"]
 * [[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
 * 输出：
 * [null, 9, null, 8]
 *
 * 解释：
 * NumArray numArray = new NumArray([1, 3, 5]);
 * numArray.sumRange(0, 2); // 返回 1 + 3 + 5 = 9
 * numArray.update(1, 2);   // nums = [1,2,5]
 * numArray.sumRange(0, 2); // 返回 1 + 2 + 5 = 8
 *
 * 解题思路:
 * 使用树状数组实现单点修改和区间查询
 * 时间复杂度：
 * - 单点修改: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(n)
 */

#include <vector>
using namespace std;

class NumArray {
private:
    // 树状数组最大容量
    int MAXN;

    // 树状数组，存储前缀和信息
    vector<int> tree;

    // 原始数组
    vector<int> nums;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
```

```
     * @return 最低位的 1 所代表的数值
     */
    int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param i 位置（从 1 开始）
     * @param v 增加的值
     */
    void add(int i, int v) {
        // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while (i < MAXN) {
            tree[i] += v;
            // 移动到父节点
            i += lowbit(i);
        }
    }

    /**
     * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
     *
     * @param i 查询的结束位置
     * @return 前缀和
     */
    int sum(int i) {
        int ans = 0;
        // 从位置 i 开始，沿着子节点路径向下累加
        while (i > 0) {
            ans += tree[i];
            // 移动到前一个相关区间
            i -= lowbit(i);
        }
        return ans;
    }

public:
    /**
     * 构造函数：用整数数组 nums 初始化对象
     *
     * @param nums 初始数组
```

```
     */
    NumArray(vector<int>& nums) {
        this->nums = nums;
        this->MAXN = nums.size() + 1;
        this->tree = vector<int>(MAXN, 0);

        // 初始化树状数组
        for (int i = 0; i < nums.size(); i++) {
            add(i + 1, nums[i]);
        }
    }

    /**
     * 更新操作：将 nums[index] 的值更新为 val
     *
     * @param index 要更新的位置
     * @param val 新的值
     */
    void update(int index, int val) {
        // 计算差值
        int delta = val - this->nums[index];
        // 更新原始数组
        this->nums[index] = val;
        // 更新树状数组
        add(index + 1, delta);
    }

    /**
     * 区间查询：返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
     *
     * @param left 区间起始位置
     * @param right 区间结束位置
     * @return 区间和
     */
    int sumRange(int left, int right) {
        return sum(right + 1) - sum(left);
    }
};

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray* obj = new NumArray(nums);
 * obj->update(index,val);
```

```
 * int param_2 = obj->sumRange(left,right);
 */
```

==========

文件: LeetCode307_RangeSumQuery.java

==========

```
package class108;

import java.util.*;

/**
 * LeetCode 307. 区域和检索 - 数组可修改
 * 题目链接: https://leetcode.cn/problems/range-sum-query-mutable/description/
 *
 * 题目描述:
 * 给你一个数组 nums ，请你完成两类查询。
 * 其中一类查询要求更新数组 nums 下标对应的值
 * 另一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right
 * 实现 NumArray 类：
 * - NumArray(int[] nums) 用整数数组 nums 初始化对象
 * - void update(int index, int val) 将 nums[index] 的值更新为 val
 * - int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
 *
 * 示例:
 * 输入：
 * ["NumArray", "sumRange", "update", "sumRange"]
 * [[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
 * 输出：
 * [null, 9, null, 8]
 *
 * 解释：
 * NumArray numArray = new NumArray([1, 3, 5]);
 * numArray.sumRange(0, 2); // 返回 1 + 3 + 5 = 9
 * numArray.update(1, 2);   // nums = [1,2,5]
 * numArray.sumRange(0, 2); // 返回 1 + 2 + 5 = 8
 *
 * 解题思路:
 * 使用树状数组实现单点修改和区间查询
 * 时间复杂度：
 * - 单点修改: O(log n)
```

```
 * - 区间查询: O(log n)
 * 空间复杂度: O(n)
 */

class NumArray {
    // 树状数组最大容量
    private int MAXN;

    // 树状数组，存储前缀和信息
    private int[] tree;

    // 原始数组
    private int[] nums;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    private int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param i 位置（从 1 开始）
     * @param v 增加的值
     */
    private void add(int i, int v) {
        // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while (i < MAXN) {
            tree[i] += v;
            // 移动到父节点
            i += lowbit(i);
        }
    }

    /**
     * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
     *
```

```
     * @param i 查询的结束位置
     * @return 前缀和
     */
    private int sum(int i) {
        int ans = 0;
        // 从位置 i 开始，沿着子节点路径向下累加
        while (i > 0) {
            ans += tree[i];
            // 移动到前一个相关区间
            i -= lowbit(i);
        }
        return ans;
    }

    /**
     * 构造函数：用整数数组 nums 初始化对象
     *
     * @param nums 初始数组
     */
    public NumArray(int[] nums) {
        this.nums = nums;
        this.MAXN = nums.length + 1;
        this.tree = new int[MAXN];

        // 初始化树状数组
        for (int i = 0; i < nums.length; i++) {
            add(i + 1, nums[i]);
        }
    }

    /**
     * 更新操作：将 nums[index] 的值更新为 val
     *
     * @param index 要更新的位置
     * @param val 新的值
     */
    public void update(int index, int val) {
        // 计算差值
        int delta = val - nums[index];
        // 更新原始数组
        nums[index] = val;
        // 更新树状数组
        add(index + 1, delta);
```

```
    }

    /**
     * 区间查询：返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和
     *
     * @param left 区间起始位置
     * @param right 区间结束位置
     * @return 区间和
     */
    public int sumRange(int left, int right) {
        return sum(right + 1) - sum(left);
    }
}


/**
 * 以下是 C++实现的树状数组（Fenwick Tree）代码
 * 包含所有已实现的算法问题的 C++版本
 *
 * // LeetCode 307 - 区域和检索 - 数组可修改（C++实现）
 * class NumArray {
 * private:
 *     vector<int> tree; // 树状数组
 *     vector<int> nums; // 原始数组
 *     int n;            // 数组大小
 *
 *     // lowbit 函数
 *     int lowbit(int x) {
 *         return x & (-x);
 *     }
 *
 *     // 单点增加
 *     void add(int i, int v) {
 *         while (i < tree.size()) {
 *             tree[i] += v;
 *             i += lowbit(i);
 *         }
 *     }
 *
 *     // 前缀和查询
 *     int sum(int i) {
 *         int ans = 0;
 *         while (i > 0) {
```

```
*            ans += tree[i];
*            i -= lowbit(i);
*        }
*        return ans;
*    }
*
* public:
*    // 构造函数
*    NumArray(vector<int>& nums) {
*        this->nums = nums;
*        this->n = nums.size();
*        this->tree.resize(n + 1, 0);
*
*        // 初始化树状数组
*        for (int i = 0; i < n; i++) {
*            add(i + 1, nums[i]);
*        }
*    }
*
*    // 更新操作
*    void update(int index, int val) {
*        int delta = val - nums[index];
*        nums[index] = val;
*        add(index + 1, delta);
*    }
*
*    // 区间查询
*    int sumRange(int left, int right) {
*        return sum(right + 1) - sum(left);
*    }
* };
*
* // POJ 2299 Ultra-QuickSort - 求逆序数（C++实现）
* class UltraQuickSort {
* private:
*    vector<long long> tree;
*    int size;
*
*    int lowbit(int x) {
*        return x & (-x);
*    }
*
*    void update(int i, int val) {
```

```
*         while (i <= size) {
*             tree[i] += val;
*             i += lowbit(i);
*         }
*     }
*
*     long long query(int i) {
*         long long sum = 0;
*         while (i > 0) {
*             sum += tree[i];
*             i -= lowbit(i);
*         }
*         return sum;
*     }
*
* public:
*     long long countInversions(vector<int>& nums) {
*         int n = nums.size();
*         if (n <= 1) return 0;
*
*         // 离散化
*         vector<int> sortedNums = nums;
*         sort(sortedNums.begin(), sortedNums.end());
*         sortedNums.erase(unique(sortedNums.begin(), sortedNums.end()), sortedNums.end());
*
*         unordered_map<int, int> valueToRank;
*         int rank = 1;
*         for (int num : sortedNums) {
*             valueToRank[num] = rank++;
*         }
*
*         // 计算逆序数
*         long long inversions = 0;
*         size = rank - 1;
*         tree.resize(size + 1, 0);
*
*         // 从右向左遍历
*         for (int i = n - 1; i >= 0; i--) {
*             int currentRank = valueToRank[nums[i]];
*             inversions += query(currentRank - 1);
*             update(currentRank, 1);
*         }
*
```

```
*         return inversions;
*     }
* };
*
* // POJ 2352 Stars - 统计左下方星星个数（C++实现）
* class Stars {
* private:
*     vector<int> tree;
*     int size;
*
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
*     void update(int i, int val) {
*         if (i == 0) i = 1; // 处理 x=0 的情况
*         while (i <= size) {
*             tree[i] += val;
*             i += lowbit(i);
*         }
*     }
*
*     int query(int i) {
*         if (i == 0) return 0;
*         int sum = 0;
*         while (i > 0) {
*             sum += tree[i];
*             i -= lowbit(i);
*         }
*         return sum;
*     }
*
* public:
*     vector<int> countStars(vector<vector<int>>& stars, int maxX) {
*         int n = stars.size();
*         vector<int> result(n, 0);
*
*         // 排序
*         sort(stars.begin(), stars.end(), [](const vector<int>& a, const vector<int>& b) {
*             if (a[1] != b[1]) return a[1] < b[1];
*             return a[0] < b[0];
*         });
*
```

```
*         size = maxX + 1;
*         tree.resize(size + 1, 0);
*
*         for (auto& star : stars) {
*             int x = star[0];
*             int level = query(x);
*             result[level]++;
*             update(x, 1);
*         }
*
*         return result;
*     }
* };
*
* // LeetCode 493 - 翻转对（C++实现）
* class ReversePairs {
* private:
*     vector<int> tree;
*     int size;
*
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
*     void update(int i, int val) {
*         while (i <= size) {
*             tree[i] += val;
*             i += lowbit(i);
*         }
*     }
*
*     int query(int i) {
*         int sum = 0;
*         while (i > 0) {
*             sum += tree[i];
*             i -= lowbit(i);
*         }
*         return sum;
*     }
*
* public:
*     int reversePairs(vector<int>& nums) {
*         int n = nums.size();
```

```
*         if (n <= 1) return 0;
*
*         // 离散化
*         set<long long> allNumbers;
*         for (int num : nums) {
*             allNumbers.insert((long long)num);
*             allNumbers.insert((long long)num * 2);
*         }
*
*         vector<long long> sortedList(allNumbers.begin(), allNumbers.end());
*         unordered_map<long long, int> valueToRank;
*         int rank = 1;
*         for (long long num : sortedList) {
*             valueToRank[num] = rank++;
*         }
*
*         int count = 0;
*         size = sortedList.size();
*         tree.resize(size + 1, 0);
*
*         // 从右向左遍历
*         for (int i = n - 1; i >= 0; i--) {
*             // 二分查找
*             int left = 0, right = sortedList.size();
*             while (left < right) {
*                 int mid = left + (right - left) / 2;
*                 if (sortedList[mid] >= (double)nums[i] / 2) {
*                     right = mid;
*                 } else {
*                     left = mid + 1;
*                 }
*             }
*             count += left > 0 ? query(left) : 0;
*
*             // 更新树状数组
*             int currentRank = valueToRank[(long long)nums[i]];
*             update(currentRank, 1);
*         }
*
*         return count;
*     }
* };
*
```

```
* // LeetCode 315 - 计算右侧小于当前元素的个数（C++实现）
* class Solution315 {
* private:
*     vector<int> tree;
*     int size;
*
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
*     void update(int i, int val) {
*         while (i <= size) {
*             tree[i] += val;
*             i += lowbit(i);
*         }
*     }
*
*     int query(int i) {
*         int sum = 0;
*         while (i > 0) {
*             sum += tree[i];
*             i -= lowbit(i);
*         }
*         return sum;
*     }
*
* public:
*     vector<int> countSmaller(vector<int>& nums) {
*         int n = nums.size();
*         vector<int> result(n);
*
*         if (n == 0) return result;
*
*         // 离散化
*         vector<int> sortedNums = nums;
*         sort(sortedNums.begin(), sortedNums.end());
*         sortedNums.erase(unique(sortedNums.begin(), sortedNums.end()), sortedNums.end());
*
*         unordered_map<int, int> valueToRank;
*         int rank = 1;
*         for (int num : sortedNums) {
*             valueToRank[num] = rank++;
*         }
```

```
*
*         size = sortedNums.size();
*         tree.resize(size + 1, 0);
*
*         // 从右向左处理
*         for (int i = n - 1; i >= 0; i--) {
*             int currentRank = valueToRank[nums[i]];
*             result[i] = query(currentRank - 1);
*             update(currentRank, 1);
*         }
*
*         return result;
*     }
* };
*
* // LeetCode 308 - 二维区域和检索 - 可变（C++实现）
* class NumMatrix308 {
* private:
*     vector<vector<int>> tree; // 二维树状数组
*     vector<vector<int>> nums; // 原始矩阵
*     int m, n;                 // 行数和列数
*
*     int lowbit(int x) {
*         return x & (-x);
*     }
*
*     void updateTree(int i, int j, int val) {
*         for (int x = i; x <= m; x += lowbit(x)) {
*             for (int y = j; y <= n; y += lowbit(y)) {
*                 tree[x][y] += val;
*             }
*         }
*     }
*
*     int queryTree(int i, int j) {
*         int sum = 0;
*         for (int x = i; x > 0; x -= lowbit(x)) {
*             for (int y = j; y > 0; y -= lowbit(y)) {
*                 sum += tree[x][y];
*             }
*         }
*         return sum;
*     }
```

```
 *
 * public:
 *     NumMatrix308(vector<vector<int>>& matrix) {
 *         if (matrix.empty() || matrix[0].empty()) return;
 *
 *         m = matrix.size();
 *         n = matrix[0].size();
 *         tree.resize(m + 1, vector<int>(n + 1, 0));
 *         nums = vector<vector<int>>(m, vector<int>(n, 0));
 *
 *         for (int i = 0; i < m; i++) {
 *             for (int j = 0; j < n; j++) {
 *                 update(i, j, matrix[i][j]);
 *             }
 *         }
 *     }
 *
 *     void update(int row, int col, int val) {
 *         int delta = val - nums[row][col];
 *         nums[row][col] = val;
 *         updateTree(row + 1, col + 1, delta);
 *     }
 *
 *     int sumRegion(int row1, int col1, int row2, int col2) {
 *         return queryTree(row2 + 1, col2 + 1) -
 *                queryTree(row1, col2 + 1) -
 *                queryTree(row2 + 1, col1) +
 *                queryTree(row1, col1);
 *     }
 * };
 */


/**
 * 以下是Python实现的树状数组（Fenwick Tree）代码
 * 包含所有已实现的算法问题的Python版本
 *
 * # LeetCode 307 - 区域和检索 - 数组可修改（Python实现）
 * class NumArray:
 *     def __init__(self, nums):
 *         self.nums = nums
 *         self.n = len(nums)
 *         self.tree = [0] * (self.n + 1)  # 树状数组从索引1开始
```

```
*
*         # 初始化树状数组
*         for i in range(self.n):
*             self.add(i + 1, nums[i])
*
*     def lowbit(self, x):
*         return x & (-x)
*
*     def add(self, i, val):
*         while i < len(self.tree):
*             self.tree[i] += val
*             i += self.lowbit(i)
*
*     def sum_prefix(self, i):
*         ans = 0
*         while i > 0:
*             ans += self.tree[i]
*             i -= self.lowbit(i)
*         return ans
*
*     def update(self, index, val):
*         delta = val - self.nums[index]
*         self.nums[index] = val
*         self.add(index + 1, delta)
*
*     def sumRange(self, left, right):
*         return self.sum_prefix(right + 1) - self.sum_prefix(left)
*
* # POJ 2299 Ultra-QuickSort - 求逆序数（Python 实现）
* class UltraQuickSort:
*     def countInversions(self, nums):
*         n = len(nums)
*         if n <= 1:
*             return 0
*
*         # 离散化
*         sorted_nums = sorted(list(set(nums)))
*         value_to_rank = {val: i + 1 for i, val in enumerate(sorted_nums)}
*
*         # 树状数组类
*         class FenwickTree:
*             def __init__(self, size):
*                 self.size = size
```

```
*                 self.tree = [0] * (size + 1)
*
*             def lowbit(self, x):
*                 return x & (-x)
*
*             def update(self, i, val):
*                 while i <= self.size:
*                     self.tree[i] += val
*                     i += self.lowbit(i)
*
*             def query(self, i):
*                 sum_ = 0
*                 while i > 0:
*                     sum_ += self.tree[i]
*                     i -= self.lowbit(i)
*                 return sum_
*
*         # 计算逆序数
*         inversions = 0
*         fenwick_tree = FenwickTree(len(sorted_nums))
*
*         # 从右向左遍历
*         for i in range(n - 1, -1, -1):
*             current_rank = value_to_rank[nums[i]]
*             inversions += fenwick_tree.query(current_rank - 1)
*             fenwick_tree.update(current_rank, 1)
*
*         return inversions
*
* # POJ 2352 Stars - 统计左下方星星个数（Python 实现）
* class Stars:
*     def countStars(self, stars, maxX):
*         n = len(stars)
*         result = [0] * n
*
*         # 排序
*         stars.sort(key=lambda x: (x[1], x[0]))
*
*         # 树状数组类
*         class FenwickTree:
*             def __init__(self, size):
*                 self.size = size
*                 self.tree = [0] * (size + 1)
```

```
*
*            def lowbit(self, x):
*                return x & (-x)
*
*            def update(self, i, val):
*                if i == 0:
*                    i = 1  # 处理 x=0 的情况
*                while i <= self.size:
*                    self.tree[i] += val
*                    i += self.lowbit(i)
*
*            def query(self, i):
*                if i == 0:
*                    return 0
*                sum_ = 0
*                while i > 0:
*                    sum_ += self.tree[i]
*                    i -= self.lowbit(i)
*                return sum_
*
*        fenwick_tree = FenwickTree(maxX + 1)
*
*        for star in stars:
*            x = star[0]
*            level = fenwick_tree.query(x)
*            result[level] += 1
*            fenwick_tree.update(x, 1)
*
*        return result
*
* # LeetCode 493 - 翻转对（Python 实现）
* class ReversePairs:
*     def reversePairs(self, nums):
*         n = len(nums)
*         if n <= 1:
*             return 0
*
*         # 离散化
*         all_numbers = set()
*         for num in nums:
*             all_numbers.add(num)
*             all_numbers.add(2 * num)
*
```

```
*         sorted_list = sorted(all_numbers)
*         value_to_rank = {val: i + 1 for i, val in enumerate(sorted_list)}
*
*         # 树状数组类
*         class FenwickTree:
*             def __init__(self, size):
*                 self.size = size
*                 self.tree = [0] * (size + 1)
*
*             def lowbit(self, x):
*                 return x & (-x)
*
*             def update(self, i, val):
*                 while i <= self.size:
*                     self.tree[i] += val
*                     i += self.lowbit(i)
*
*             def query(self, i):
*                 sum_ = 0
*                 while i > 0:
*                     sum_ += self.tree[i]
*                     i -= self.lowbit(i)
*                 return sum_
*
*         count = 0
*         fenwick_tree = FenwickTree(len(sorted_list))
*
*         # 从右向左遍历
*         for i in range(n - 1, -1, -1):
*             # 二分查找
*             target = nums[i] / 2
*             left, right = 0, len(sorted_list)
*             while left < right:
*                 mid = left + (right - left) // 2
*                 if sorted_list[mid] >= target:
*                     right = mid
*                 else:
*                     left = mid + 1
*             count += left > 0 and fenwick_tree.query(left) or 0
*
*             # 更新树状数组
*             current_rank = value_to_rank[nums[i]]
*             fenwick_tree.update(current_rank, 1)
```

```
*
*         return count
*
* # LeetCode 315 - 计算右侧小于当前元素的个数（Python 实现）
* class Solution315:
*     def countSmaller(self, nums):
*         n = len(nums)
*         result = [0] * n
*
*         if n == 0:
*             return []
*
*         # 离散化
*         sorted_nums = sorted(list(set(nums)))
*         value_to_rank = {val: i + 1 for i, val in enumerate(sorted_nums)}
*
*         # 树状数组类
*         class FenwickTree:
*             def __init__(self, size):
*                 self.size = size
*                 self.tree = [0] * (size + 1)
*
*             def lowbit(self, x):
*                 return x & (-x)
*
*             def update(self, i, val):
*                 while i <= self.size:
*                     self.tree[i] += val
*                     i += self.lowbit(i)
*
*             def query(self, i):
*                 sum_ = 0
*                 while i > 0:
*                     sum_ += self.tree[i]
*                     i -= self.lowbit(i)
*                 return sum_
*
*         fenwick_tree = FenwickTree(len(sorted_nums))
*
*         # 从右向左处理
*         for i in range(n - 1, -1, -1):
*             current_rank = value_to_rank[nums[i]]
*             result[i] = fenwick_tree.query(current_rank - 1)
```

```
*             fenwick_tree.update(current_rank, 1)
*
*         return result
*
* # LeetCode 308 - 二维区域和检索 - 可变（Python实现）
* class NumMatrix308:
*     def __init__(self, matrix):
*         if not matrix or not matrix[0]:
*             return
*
*         self.m = len(matrix)
*         self.n = len(matrix[0])
*         self.tree = [[0] * (self.n + 1) for _ in range(self.m + 1)]
*         self.nums = [[0] * self.n for _ in range(self.m)]
*
*         for i in range(self.m):
*             for j in range(self.n):
*                 self.update(i, j, matrix[i][j])
*
*     def lowbit(self, x):
*         return x & (-x)
*
*     def updateTree(self, i, j, val):
*         x = i
*         while x <= self.m:
*             y = j
*             while y <= self.n:
*                 self.tree[x][y] += val
*                 y += self.lowbit(y)
*             x += self.lowbit(x)
*
*     def queryTree(self, i, j):
*         sum_ = 0
*         x = i
*         while x > 0:
*             y = j
*             while y > 0:
*                 sum_ += self.tree[x][y]
*                 y -= self.lowbit(y)
*             x -= self.lowbit(x)
*         return sum_
*
*     def update(self, row, col, val):
```

```
 *         delta = val - self.nums[row][col]
 *         self.nums[row][col] = val
 *         self.updateTree(row + 1, col + 1, delta)
 *
 *     def sumRegion(self, row1, col1, row2, col2):
 *         return (self.queryTree(row2 + 1, col2 + 1) -
 *                 self.queryTree(row1, col2 + 1) -
 *                 self.queryTree(row2 + 1, col1) +
 *                 self.queryTree(row1, col1))
 */
```

```
/**
 * POJ 2299 Ultra-QuickSort（求逆序数）
 * 题目链接: http://poj.org/problem?id=2299
 *
 * 题目描述:
 * 给定一个序列，求该序列的逆序数。逆序数是指序列中满足 i < j 且 a[i] > a[j] 的有序对 (i, j) 的个数。
 *
 * 解题思路:
 * 使用树状数组结合离散化来高效计算逆序数
 * 1. 离散化：将原始数组中的值映射到连续的较小范围内
 * 2. 从右向左遍历数组，对于每个元素：
 *    - 查询树状数组中比当前元素小的元素个数，即为该元素贡献的逆序数
 *    - 将当前元素插入到树状数组中
 *
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */
class UltraQuickSort {
    /**
     * 树状数组实现
     */
    class FenwickTree {
        private int[] tree;
        private int size;

        public FenwickTree(int size) {
            this.size = size;
            this.tree = new int[size + 1];
        }
```

```
    private int lowbit(int x) {
        return x & (-x);
    }

    public void update(int i, int val) {
        while (i <= size) {
            tree[i] += val;
            i += lowbit(i);
        }
    }

    public int query(int i) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= lowbit(i);
        }
        return sum;
    }
}

/**
 * 计算逆序数
 * @param nums 输入数组
 * @return 逆序数
 */
public long countInversions(int[] nums) {
    int n = nums.length;
    if (n <= 1) {
        return 0;
    }

    // 离散化
    int[] sortedNums = Arrays.copyOf(nums, n);
    Arrays.sort(sortedNums);

    Map<Integer, Integer> valueToRank = new HashMap<>();
    int rank = 1;
    for (int i = 0; i < n; i++) {
        if (i == 0 || sortedNums[i] != sortedNums[i - 1]) {
            valueToRank.put(sortedNums[i], rank++);
        }
```

```
        }

        // 计算逆序数
        long inversions = 0;
        FenwickTree fenwickTree = new FenwickTree(rank - 1);

        // 从右向左遍历
        for (int i = n - 1; i >= 0; i--) {
            int currentRank = valueToRank.get(nums[i]);
            // 比当前元素小的元素个数就是当前元素贡献的逆序数
            inversions += fenwickTree.query(currentRank - 1);
            // 将当前元素加入树状数组
            fenwickTree.update(currentRank, 1);
        }

        return inversions;
    }
}



/**
 * POJ 2352 Stars（统计左下方星星个数）
 * 题目链接: http://poj.org/problem?id=2352
 *
 * 题目描述:
 * 在一个平面直角坐标系中，给定 n 个星星的坐标，每个星星的等级等于坐标严格小于它的星星的数量。
 * 求各个等级的星星数量。
 *
 * 解题思路:
 * 1. 按照 y 坐标从小到大排序，y 相同则按 x 从小到大排序
 * 2. 使用树状数组来维护 x 坐标的出现次数
 * 3. 对于每个星星，查询树状数组中 x 坐标小于当前星星 x 坐标的星星数量，即为该星星的等级
 * 4. 更新树状数组，将当前星星的 x 坐标计数加 1
 *
 * 时间复杂度：O(n log max_x)
 * 空间复杂度：O(max_x)
 */
class Stars {
    /**
     * 树状数组实现
     */
    class FenwickTree {
        private int[] tree;
```

```
    private int size;

    public FenwickTree(int size) {
        this.size = size;
        this.tree = new int[size + 1]; // 注意 x 坐标可能从 0 开始
    }

    private int lowbit(int x) {
        return x & (-x);
    }

    public void update(int i, int val) {
        // 注意：如果 x 可能为 0，需要将其映射到 1，因为树状数组从 1 开始
        if (i == 0) {
            i = 1;
        }
        while (i <= size) {
            tree[i] += val;
            i += lowbit(i);
        }
    }

    public int query(int i) {
        // 注意：如果 x 可能为 0，需要将其映射到 1
        if (i == 0) {
            return 0;
        }
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= lowbit(i);
        }
        return sum;
    }
}

/**
 * 统计各个等级的星星数量
 * @param stars 星星坐标数组，每个元素为[x, y]
 * @param maxX 最大 x 坐标值
 * @return 等级分布数组，result[i]表示等级为 i 的星星数量
 */
public int[] countStars(int[][] stars, int maxX) {
```

```
        int n = stars.length;
        int[] result = new int[n]; // 最多可能有 n 个不同等级

        // 按照 y 坐标从小到大排序，y 相同则按 x 从小到大排序
        Arrays.sort(stars, (a, b) -> {
            if (a[1] != b[1]) {
                return a[1] - b[1];
            }
            return a[0] - b[0];
        });

        FenwickTree fenwickTree = new FenwickTree(maxX + 1); // x 坐标可能从 0 开始

        for (int[] star : stars) {
            int x = star[0];
            // 查询等级：x 坐标严格小于当前 x 的星星数量
            int level = fenwickTree.query(x);
            result[level]++;
            // 更新树状数组
            fenwickTree.update(x, 1);
        }

        return result;
    }
}



/**
 * LeetCode 493. 翻转对
 * 题目链接: https://leetcode.cn/problems/reverse-pairs/description/
 *
 * 题目描述:
 * 给定一个数组 nums ，如果 i < j 且 nums[i] > 2*nums[j] 我们就将 (i,j) 称作一个重要翻转对。
 * 你需要返回给定数组中的重要翻转对的数量。
 *
 * 解题思路:
 * 使用树状数组结合离散化来高效计算重要翻转对
 * 1. 离散化：将原始数组和 2 倍原始数组的值合并后进行离散化
 * 2. 从右向左遍历数组，对于每个元素：
 *    - 查询树状数组中小于 nums[i]/2 的元素个数，即为该元素贡献的翻转对数量
 *    - 将当前元素插入到树状数组中
 *
 * 时间复杂度：O(n log n)
```

```
 * 空间复杂度：O(n)
 */
class ReversePairs {
    /**
     * 树状数组实现
     */
    class FenwickTree {
        private int[] tree;
        private int size;

        public FenwickTree(int size) {
            this.size = size;
            this.tree = new int[size + 1];
        }

        private int lowbit(int x) {
            return x & (-x);
        }

        public void update(int i, int val) {
            while (i <= size) {
                tree[i] += val;
                i += lowbit(i);
            }
        }

        public int query(int i) {
            int sum = 0;
            while (i > 0) {
                sum += tree[i];
                i -= lowbit(i);
            }
            return sum;
        }
    }

    /**
     * 计算重要翻转对的数量
     * @param nums 输入数组
     * @return 重要翻转对的数量
     */
    public int reversePairs(int[] nums) {
        int n = nums.length;
```

```
if (n <= 1) {
    return 0;
}

// 离散化：需要处理 nums[i]和 2*nums[i]
Set<Long> allNumbers = new HashSet<>();
for (int num : nums) {
    allNumbers.add((long)num);
    allNumbers.add(2L * num);
}

// 将所有数字排序并映射到排名
List<Long> sortedList = new ArrayList<>(allNumbers);
Collections.sort(sortedList);

Map<Long, Integer> valueToRank = new HashMap<>();
for (int i = 0; i < sortedList.size(); i++) {
    valueToRank.put(sortedList.get(i), i + 1); // 排名从 1 开始
}

int count = 0;
FenwickTree fenwickTree = new FenwickTree(sortedList.size());

// 从右向左遍历
for (int i = n - 1; i >= 0; i--) {
    // 查找有多少个已处理的元素小于 nums[i]/2
    // 使用二分查找找到第一个大于等于 nums[i]/2 的元素的位置
    int left = 0;
    int right = sortedList.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (sortedList.get(mid) >= (double)nums[i] / 2) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    // 所有小于 nums[i]/2 的元素个数为 left
    count += left > 0 ? fenwickTree.query(left) : 0;

    // 将当前元素插入树状数组
    int rank = valueToRank.get((long)nums[i]);
    fenwickTree.update(rank, 1);
```

```
        }

        return count;
    }
}

/**
 * Your NumArray object will be instantiated and called as such:
 * NumArray obj = new NumArray(nums);
 * obj.update(index,val);
 * int param_2 = obj.sumRange(left,right);
 */



/**
 * LeetCode 315. 计算右侧小于当前元素的个数
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/description/
 *
 * 题目描述:
 * 给你一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质：
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例 1:
 * 输入：nums = [5,2,6,1]
 * 输出：[2,1,1,0]
 * 解释：
 * 5 的右侧有 2 个更小的元素 (2 和 1)
 * 2 的右侧仅有 1 个更小的元素 (1)
 * 6 的右侧有 1 个更小的元素 (1)
 * 1 的右侧有 0 个更小的元素
 *
 * 示例 2:
 * 输入：nums = [-1]
 * 输出：[0]
 *
 * 示例 3:
 * 输入：nums = [-1,-1]
 * 输出：[0,0]
 *
 * 解题思路:
 * 使用树状数组（Fenwick Tree）结合离散化来高效求解逆序对问题
 * 1. 离散化：将原始数组中的值映射到连续的较小范围内
 * 2. 从右向左遍历数组，对于每个元素：
```

```
 *     - 查询树状数组中小于当前元素的元素个数
 *     - 将当前元素插入到树状数组中
 *
 * 时间复杂度：
 * - 离散化：O(n log n)
 * - 树状数组操作：O(n log n)
 * 总时间复杂度：O(n log n)
 *
 * 空间复杂度：
 * - 树状数组：O(n)
 * - 离散化数组：O(n)
 * 总空间复杂度：O(n)
 *
 * 优化分析：
 * 这是本题的最优解法之一。其他可能的解法包括归并排序（时间复杂度 O(n log n)，但需要更多辅助空间）
 * 和二分搜索树（最坏情况可能退化到 O(n²)）。树状数组结合离散化的方法在时间和空间上都是最优的。
 */
class Solution315 {
    /**
     * 树状数组实现
     */
    class FenwickTree {
        private int[] tree;
        private int size;

        /**
         * 构造函数
         * @param size 树状数组大小
         */
        public FenwickTree(int size) {
            this.size = size;
            this.tree = new int[size + 1]; // 树状数组从索引 1 开始
        }

        /**
         * lowbit 操作：获取 x 的二进制表示中最低位 1 所代表的值
         * @param x 输入整数
         * @return 最低位 1 所代表的值
         */
        private int lowbit(int x) {
            return x & (-x);
        }
```

```
    /**
     * 单点更新：在索引 i 的位置增加 val
     * @param i 索引位置（从 1 开始）
     * @param val 增加的值
     */
    public void update(int i, int val) {
        while (i <= size) {
            tree[i] += val;
            i += lowbit(i);
        }
    }

    /**
     * 前缀和查询：查询[1, i]区间的元素和
     * @param i 结束索引（从 1 开始）
     * @return [1, i]区间的元素和
     */
    public int query(int i) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= lowbit(i);
        }
        return sum;
    }
}

/**
 * 计算右侧小于当前元素的个数
 * @param nums 输入数组
 * @return 结果数组，其中每个元素表示右侧小于当前元素的个数
 */
public List<Integer> countSmaller(int[] nums) {
    int n = nums.length;
    List<Integer> result = new ArrayList<>();

    if (n == 0) {
        return result;
    }

    // 离散化处理
    // 1. 复制数组并排序去重
    int[] sortedNums = Arrays.copyOf(nums, n);
```

```
        Arrays.sort(sortedNums);

        // 2. 创建值到排名的映射
        Map<Integer, Integer> valueToRank = new HashMap<>();
        int rank = 1;
        for (int i = 0; i < n; i++) {
            if (i == 0 || sortedNums[i] != sortedNums[i - 1]) {
                valueToRank.put(sortedNums[i], rank++);
            }
        }

        // 3. 使用树状数组从右向左处理
        FenwickTree fenwickTree = new FenwickTree(rank - 1);
        for (int i = n - 1; i >= 0; i--) {
            // 获取当前元素的排名
            int currentRank = valueToRank.get(nums[i]);

            // 查询小于当前元素的数量（即排名小于 currentRank 的元素个数）
            int smallerCount = fenwickTree.query(currentRank - 1);

            // 添加结果（注意需要后续反转）
            result.add(smallerCount);

            // 将当前元素加入树状数组
            fenwickTree.update(currentRank, 1);
        }

        // 反转结果数组，因为我们是从右向左处理的
        Collections.reverse(result);

        return result;
    }
}



/**
 * LeetCode 308. 二维区域和检索 - 可变
 * 题目链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/description/
 *
 * 题目描述:
 * 给你一个二维矩阵 matrix ，需要支持以下操作：
 * 1. update(row, col, val)：将 matrix[row][col] 的值更新为 val。
 * 2. sumRegion(row1, col1, row2, col2)：返回矩阵中从左上角 (row1, col1) 到右下角 (row2, col2) 的
```

矩形区域内所有元素的和。
 *
 * 示例 1:
 * 输入:
 * ["NumMatrix", "sumRegion", "update", "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], [3, 2, 2], [2, 1, 4, 3]]
 * 输出:
 * [null, 8, null, 10]
 *
 * 解题思路:
 * 使用二维树状数组来实现二维区域的单点更新和区间查询
 *
 * 时间复杂度：
 * - 单点更新: O(log m * log n)
 * - 区间查询: O(log m * log n)
 *
 * 空间复杂度: O(m * n)
 */
```
class NumMatrix308 {
    private int[][] tree; // 二维树状数组
    private int[][] nums; // 原始矩阵
    private int m; // 行数
    private int n; // 列数

    /**
     * 构造函数
     * @param matrix 输入矩阵
     */
    public NumMatrix308(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return;
        }

        m = matrix.length;
        n = matrix[0].length;
        tree = new int[m + 1][n + 1]; // 树状数组从索引 1 开始
        nums = new int[m][n]; // 保存原始矩阵用于计算差分

        // 初始化树状数组
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                update(i, j, matrix[i][j]);
```

```
        }
    }
}

/**
 * lowbit 操作
 * @param x 输入整数
 * @return 最低位 1 所代表的值
 */
private int lowbit(int x) {
    return x & (-x);
}

/**
 * 单点更新：在(i+1,j+1)位置增加 val
 * @param i 行索引（从 0 开始）
 * @param j 列索引（从 0 开始）
 * @param val 增加的值
 */
private void updateTree(int i, int j, int val) {
    for (int x = i; x <= m; x += lowbit(x)) {
        for (int y = j; y <= n; y += lowbit(y)) {
            tree[x][y] += val;
        }
    }
}

/**
 * 查询从(1,1)到(i,j)的前缀和
 * @param i 结束行索引（从 1 开始）
 * @param j 结束列索引（从 1 开始）
 * @return 前缀和
 */
private int queryTree(int i, int j) {
    int sum = 0;
    for (int x = i; x > 0; x -= lowbit(x)) {
        for (int y = j; y > 0; y -= lowbit(y)) {
            sum += tree[x][y];
        }
    }
    return sum;
}
```

```
    /**
     * 更新矩阵中的元素值
     * @param row 行索引
     * @param col 列索引
     * @param val 新的值
     */
    public void update(int row, int col, int val) {
        // 计算差值
        int delta = val - nums[row][col];
        // 更新原始矩阵
        nums[row][col] = val;
        // 更新树状数组
        updateTree(row + 1, col + 1, delta);
    }

    /**
     * 查询矩阵区域和
     * @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 区域和
     */
    public int sumRegion(int row1, int col1, int row2, int col2) {
        // 使用容斥原理计算区域和
        return queryTree(row2 + 1, col2 + 1)
             - queryTree(row1, col2 + 1)
             - queryTree(row2 + 1, col1)
             + queryTree(row1, col1);
    }
}

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

文件: LeetCode307_RangeSumQuery.py
==========================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
LeetCode 307. 区域和检索 - 数组可修改
题目链接: https://leetcode.cn/problems/range-sum-query-mutable/description/
```

题目描述:
给你一个数组 nums ，请你完成两类查询。
其中一类查询要求更新数组 nums 下标对应的值
另一类查询要求返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和，其中 left <= right
实现 NumArray 类：
- NumArray(int[] nums) 用整数数组 nums 初始化对象
- void update(int index, int val) 将 nums[index] 的值更新为 val
- int sumRange(int left, int right) 返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和

示例:
输入：
["NumArray", "sumRange", "update", "sumRange"]
[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
输出：
[null, 9, null, 8]

解释：
NumArray numArray = new NumArray([1, 3, 5]);
numArray.sumRange(0, 2); // 返回 1 + 3 + 5 = 9
numArray.update(1, 2);   // nums = [1,2,5]
numArray.sumRange(0, 2); // 返回 1 + 2 + 5 = 8

解题思路:
使用树状数组实现单点修改和区间查询
时间复杂度：
- 单点修改: O(log n)
- 区间查询: O(log n)
空间复杂度: O(n)
"""

```
class NumArray:
    """
    树状数组类，用于高效处理单点修改和区间查询
    """

    def __init__(self, nums):
        """
        初始化树状数组
        :param nums: 初始数组
        """
```

```
        self.n = len(nums)
        self.nums = nums
        # 树状数组，存储前缀和信息，索引从 1 开始
        self.tree = [0] * (self.n + 1)

        # 初始化树状数组
        for i in range(self.n):
            self._add(i + 1, nums[i])

    def _lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
        例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

        :param i: 输入数字
        :return: 最低位的 1 所代表的数值
        """
        return i & (-i)

    def _add(self, i, v):
        """
        单点增加操作：在位置 i 上增加 v

        :param i: 位置（从 1 开始）
        :param v: 增加的值
        """
        # 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while i <= self.n:
            self.tree[i] += v
            # 移动到父节点
            i += self._lowbit(i)

    def _sum(self, i):
        """
        查询前缀和：计算从位置 1 到位置 i 的所有元素之和

        :param i: 查询的结束位置
        :return: 前缀和
        """
        ans = 0
        # 从位置 i 开始，沿着子节点路径向下累加
        while i > 0:
            ans += self.tree[i]
```

```
            # 移动到前一个相关区间
            i -= self._lowbit(i)
        return ans

    def update(self, index, val):
        """
        更新操作：将 nums[index] 的值更新为 val

        :param index: 要更新的位置
        :param val: 新的值
        """
        # 计算差值
        delta = val - self.nums[index]
        # 更新原始数组
        self.nums[index] = val
        # 更新树状数组
        self._add(index + 1, delta)

    def sumRange(self, left, right):
        """
        区间查询：返回数组 nums 中索引 left 和索引 right 之间（包含）的 nums 元素的和

        :param left: 区间起始位置
        :param right: 区间结束位置
        :return: 区间和
        """
        return self._sum(right + 1) - self._sum(left)


# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# obj.update(index,val)
# param_2 = obj.sumRange(left,right)

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

文件: LeetCode308_RangeSumQuery2D.cpp
==============================================
/*
 * LeetCode 308. 二维区域和检索- 矩阵可修改
 * 题目链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/
 *
 * 题目描述:
```

```
 * 给你一个 2D 矩阵 matrix，请计算出从左上角 (row1, col1) 到右下角 (row2, col2) 组成的矩形中所有元素的和。
 * 实现 NumMatrix 类：
 * - NumMatrix(int[][] matrix) 用整数矩阵 matrix 初始化对象
 * - void update(int row, int col, int val) 更新 matrix[row][col] 的值为 val
 * - int sumRegion(int row1, int col1, int row2, int col2) 返回矩阵 matrix 中指定矩形区域的元素和
 *
 * 解题思路:
 * 使用二维树状数组来实现动态二维区域和查询。
 * 1. 对于更新操作，计算差值并更新树状数组
 * 2. 对于区域和查询，使用二维前缀和的容斥原理：
 *    sumRegion(row1, col1, row2, col2) =
 *      sum(row2, col2) - sum(row1-1, col2) - sum(row2, col1-1) + sum(row1-1, col1-1)
 *
 * 时间复杂度：
 * - 更新操作: O(log m * log n)
 * - 区域和查询: O(log m * log n)
 * 空间复杂度: O(m * n)
 */

class NumMatrix {
private:
    int** tree;  // 二维树状数组
    int** nums;  // 原始矩阵
    int m, n;    // 矩阵的行数和列数

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    int lowbit(int i) {
        return i & -i;
    }

    /**
     * 二维树状数组单点增加操作
     *
     * @param x x 坐标（从 1 开始）
     * @param y y 坐标（从 1 开始）
     * @param v 增加的值
```

```
     */
    void add(int x, int y, int v) {
        for (int i = x; i <= m; i += lowbit(i)) {
            for (int j = y; j <= n; j += lowbit(j)) {
                tree[i][j] += v;
            }
        }
    }

    /**
     * 二维树状数组前缀和查询：计算从(0,0)到(x,y)的矩形区域内所有元素的和
     *
     * @param x x坐标
     * @param y y坐标
     * @return 前缀和
     */
    int sum(int x, int y) {
        int ans = 0;
        for (int i = x + 1; i > 0; i -= lowbit(i)) {
            for (int j = y + 1; j > 0; j -= lowbit(j)) {
                ans += tree[i][j];
            }
        }
        return ans;
    }

public:
    /**
     * 二维树状数组初始化
     *
     * @param matrix 输入矩阵
     * @param matrixSize 矩阵行数
     * @param matrixColSize 矩阵每行的列数
     */
    NumMatrix(int** matrix, int matrixSize, int* matrixColSize) {
        if (matrix == nullptr || matrixSize == 0 || matrixColSize[0] == 0) {
            this->m = 0;
            this->n = 0;
            return;
        }

        m = matrixSize;
        n = matrixColSize[0];
```

```
    // 分配内存
    tree = new int*[m + 1];
    nums = new int*[m];
    for (int i = 0; i <= m; i++) {
        tree[i] = new int[n + 1];
        if (i < m) {
            nums[i] = new int[n];
        }
    }

    // 初始化数组
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            tree[i][j] = 0;
        }
        if (i < m) {
            for (int j = 0; j < n; j++) {
                nums[i][j] = 0;
            }
        }
    }

    // 初始化树状数组
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            update(i, j, matrix[i][j]);
        }
    }
}

/**
 * 析构函数
 */
~NumMatrix() {
    if (m > 0 && n > 0) {
        for (int i = 0; i <= m; i++) {
            delete[] tree[i];
            if (i < m) {
                delete[] nums[i];
            }
        }
        delete[] tree;
```

```
            delete[] nums;
        }
    }

    /**
     * 更新操作：将 matrix[row][col] 的值更新为 val
     *
     * @param row 行索引
     * @param col 列索引
     * @param val 新的值
     */
    void update(int row, int col, int val) {
        if (m == 0 || n == 0) return;

        // 计算差值
        int delta = val - nums[row][col];
        // 更新原始数组
        nums[row][col] = val;
        // 更新树状数组
        add(row + 1, col + 1, delta);
    }

    /**
     * 区域和查询：返回矩阵中指定矩形区域的元素和
     *
     * @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 矩形区域的元素和
     */
    int sumRegion(int row1, int col1, int row2, int col2) {
        if (m == 0 || n == 0) return 0;

        // 使用二维前缀和的容斥原理
        return sum(row2, col2) - sum(row1 - 1, col2) - sum(row2, col1 - 1) + sum(row1 - 1, col1 -
1);
    }
};

/*
 * 测试函数
 * 由于编译环境限制，此处省略测试代码
```

```
 * 实际使用时可以直接调用 NumMatrix 类的方法
 */

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

文件: LeetCode308_RangeSumQuery2D.java
==========================================
package class108;

/**
 * LeetCode 308. 二维区域和检索- 矩阵可修改
 * 题目链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/
 *
 * 题目描述:
 * 给你一个 2D 矩阵 matrix，请计算出从左上角 (row1, col1) 到右下角 (row2, col2) 组成的矩形中所有元素的和。
 * 实现 NumMatrix 类：
 * - NumMatrix(int[][] matrix) 用整数矩阵 matrix 初始化对象
 * - void update(int row, int col, int val) 更新 matrix[row][col] 的值为 val
 * - int sumRegion(int row1, int col1, int row2, int col2) 返回矩阵 matrix 中指定矩形区域的元素和
 *
 * 解题思路:
 * 使用二维树状数组来实现动态二维区域和查询。
 * 1. 对于更新操作，计算差值并更新树状数组
 * 2. 对于区域和查询，使用二维前缀和的容斥原理：
 *    sumRegion(row1, col1, row2, col2) =
 *      sum(row2, col2) - sum(row1-1, col2) - sum(row2, col1-1) + sum(row1-1, col1-1)
 *
 * 时间复杂度：
 * - 更新操作: O(log m * log n)
 * - 区域和查询: O(log m * log n)
 * 空间复杂度: O(m * n)
 */

public class LeetCode308_RangeSumQuery2D {
    class NumMatrix {
        private int[][] tree;  // 二维树状数组
        private int[][] nums;  // 原始矩阵
        private int m, n;      // 矩阵的行数和列数

        /**
         * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
         * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
```

```
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    private int lowbit(int i) {
        return i & -i;
    }

    /**
     * 二维树状数组初始化
     *
     * @param matrix 输入矩阵
     */
    public NumMatrix(int[][] matrix) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return;
        }

        m = matrix.length;
        n = matrix[0].length;
        this.nums = new int[m][n];
        this.tree = new int[m + 1][n + 1];

        // 初始化树状数组
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                update(i, j, matrix[i][j]);
            }
        }
    }

    /**
     * 二维树状数组单点增加操作
     *
     * @param x x 坐标（从 1 开始）
     * @param y y 坐标（从 1 开始）
     * @param v 增加的值
     */
    private void add(int x, int y, int v) {
        for (int i = x; i <= m; i += lowbit(i)) {
            for (int j = y; j <= n; j += lowbit(j)) {
                tree[i][j] += v;
            }
```

```
    }
}

/**
 * 二维树状数组前缀和查询：计算从(0,0)到(x,y)的矩形区域内所有元素的和
 *
 * @param x x坐标
 * @param y y坐标
 * @return 前缀和
 */
private int sum(int x, int y) {
    int ans = 0;
    for (int i = x + 1; i > 0; i -= lowbit(i)) {
        for (int j = y + 1; j > 0; j -= lowbit(j)) {
            ans += tree[i][j];
        }
    }
    return ans;
}

/**
 * 更新操作：将 matrix[row][col] 的值更新为 val
 *
 * @param row 行索引
 * @param col 列索引
 * @param val 新的值
 */
public void update(int row, int col, int val) {
    if (m == 0 || n == 0) return;

    // 计算差值
    int delta = val - nums[row][col];
    // 更新原始数组
    nums[row][col] = val;
    // 更新树状数组
    add(row + 1, col + 1, delta);
}

/**
 * 区域和查询：返回矩阵中指定矩形区域的元素和
 *
 * @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 矩形区域的元素和
         */
        public int sumRegion(int row1, int col1, int row2, int col2) {
            if (m == 0 || n == 0) return 0;

            // 使用二维前缀和的容斥原理
            return sum(row2, col2) - sum(row1 - 1, col2) - sum(row2, col1 - 1) + sum(row1 - 1,
col1 - 1);
        }
    }

    /**
     * 测试函数
     */
    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}
        };

        LeetCode308_RangeSumQuery2D solution = new LeetCode308_RangeSumQuery2D();
        NumMatrix numMatrix = solution.new NumMatrix(matrix);

        // 测试区域和查询
        System.out.println("sumRegion(2, 1, 4, 3): " + numMatrix.sumRegion(2, 1, 4, 3)); // 期望:
8
        System.out.println("sumRegion(1, 1, 2, 2): " + numMatrix.sumRegion(1, 1, 2, 2)); // 期望:
11
        System.out.println("sumRegion(1, 2, 2, 4): " + numMatrix.sumRegion(1, 2, 2, 4)); // 期望:
12

        // 测试更新操作
        numMatrix.update(3, 2, 2);
        System.out.println("更新(3, 2)为2后，sumRegion(2, 1, 4, 3): " + numMatrix.sumRegion(2,
1, 4, 3)); // 期望: 10
    }
}
```

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

文件: LeetCode308_RangeSumQuery2D.py
================================================
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
LeetCode 308. 二维区域和检索- 矩阵可修改
题目链接: https://leetcode.cn/problems/range-sum-query-2d-mutable/

题目描述:
给你一个 2D 矩阵 matrix，请计算出从左上角 (row1, col1) 到右下角 (row2, col2) 组成的矩形中所有元素的和。
实现 NumMatrix 类：
- NumMatrix(int[][] matrix) 用整数矩阵 matrix 初始化对象
- void update(int row, int col, int val) 更新 matrix[row][col] 的值为 val
- int sumRegion(int row1, int col1, int row2, int col2) 返回矩阵 matrix 中指定矩形区域的元素和

解题思路:
使用二维树状数组来实现动态二维区域和查询。
1. 对于更新操作，计算差值并更新树状数组
2. 对于区域和查询，使用二维前缀和的容斥原理：
   sumRegion(row1, col1, row2, col2) =
     sum(row2, col2) - sum(row1-1, col2) - sum(row2, col1-1) + sum(row1-1, col1-1)

时间复杂度：
- 更新操作: O(log m * log n)
- 区域和查询: O(log m * log n)
空间复杂度: O(m * n)
"""



class NumMatrix:
    def __init__(self, matrix):
        """
        二维树状数组初始化

        :param matrix: 输入矩阵
        """
        if not matrix or not matrix[0]:
            self.m = 0
```

```
            self.n = 0
            return

        self.m = len(matrix)
        self.n = len(matrix[0])
        self.nums = [[0 for _ in range(self.n)] for _ in range(self.m)]
        self.tree = [[0 for _ in range(self.n + 1)] for _ in range(self.m + 1)]

        # 初始化树状数组
        for i in range(self.m):
            for j in range(self.n):
                self.update(i, j, matrix[i][j])

    def lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
        例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

        :param i: 输入数字
        :return: 最低位的 1 所代表的数值
        """
        return i & -i

    def add(self, x, y, v):
        """
        二维树状数组单点增加操作

        :param x: x 坐标（从 1 开始）
        :param y: y 坐标（从 1 开始）
        :param v: 增加的值
        """
        i = x
        while i <= self.m:
            j = y
            while j <= self.n:
                self.tree[i][j] += v
                j += self.lowbit(j)
            i += self.lowbit(i)

    def sum(self, x, y):
        """
        二维树状数组前缀和查询：计算从(0,0)到(x,y)的矩形区域内所有元素的和
```

```
        :param x: x 坐标
        :param y: y 坐标
        :return: 前缀和
        """
        ans = 0
        i = x + 1
        while i > 0:
            j = y + 1
            while j > 0:
                ans += self.tree[i][j]
                j -= self.lowbit(j)
            i -= self.lowbit(i)
        return ans

    def update(self, row, col, val):
        """
        更新操作：将 matrix[row][col] 的值更新为 val

        :param row: 行索引
        :param col: 列索引
        :param val: 新的值
        """
        if self.m == 0 or self.n == 0:
            return

        # 计算差值
        delta = val - self.nums[row][col]
        # 更新原始数组
        self.nums[row][col] = val
        # 更新树状数组
        self.add(row + 1, col + 1, delta)

    def sumRegion(self, row1, col1, row2, col2):
        """
        区域和查询：返回矩阵中指定矩形区域的元素和

        :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: 矩形区域的元素和
        """
        if self.m == 0 or self.n == 0:
```

```
            return 0

        # 使用二维前缀和的容斥原理
        return (self.sum(row2, col2) - self.sum(row1 - 1, col2) -
                self.sum(row2, col1 - 1) + self.sum(row1 - 1, col1 - 1))


def main():
    """
    测试函数
    """
    # 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)

    # 测试区域和查询
    print("sumRegion(2, 1, 4, 3):", numMatrix.sumRegion(2, 1, 4, 3))  # 期望: 8
    print("sumRegion(1, 1, 2, 2):", numMatrix.sumRegion(1, 1, 2, 2))  # 期望: 11
    print("sumRegion(1, 2, 2, 4):", numMatrix.sumRegion(1, 2, 2, 4))  # 期望: 12

    # 测试更新操作
    numMatrix.update(3, 2, 2)
    print("更新(3, 2)为2后，sumRegion(2, 1, 4, 3):", numMatrix.sumRegion(2, 1, 4, 3))  # 期望:
10


if __name__ == "__main__":
    main()

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

文件: LeetCode315_CountSmallerNumbersAfterSelf.cpp
==========================================
/*
 * 由于编译环境限制，使用基础C++实现，避免使用STL容器
 * 这是LeetCode 315. 计算右侧小于当前元素的个数的C++实现
```

```
 */

/**
 * LeetCode 315. 计算右侧小于当前元素的个数
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 *
 * 题目描述:
 * 给你一个整数数组 nums ，按要求返回一个新数组 counts 。数组 counts 有该性质：
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例:
 * 输入: nums = [5,2,6,1]
 * 输出: [2,1,1,0]
 * 解释:
 * 5 的右侧有 2 个更小的元素 (2 和 1)
 * 2 的右侧有 1 个更小的元素 (1)
 * 6 的右侧有 1 个更小的元素 (1)
 * 1 的右侧有 0 个更小的元素
 *
 * 解题思路:
 * 使用树状数组 + 离散化来解决这个问题。
 * 1. 离散化：由于数值范围可能很大，需要先进行离散化处理，将数值映射到连续的小范围内
 * 2. 从右往左遍历数组：
 *    - 对于每个元素，查询树状数组中比它小的元素个数（即右侧小于当前元素的个数）
 *    - 将当前元素插入树状数组
 *
 * 时间复杂度：O(n log n)，其中 n 是数组长度
 * 空间复杂度：O(n)
 */

class Solution {
private:
    int tree[20001];  // 假设最大数组大小为20000
    int sorted[20001];
    int MAXN;
    int uniqueCount;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
```

```
 */
int lowbit(int i) {
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
void add(int i, int v) {
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i < MAXN) {
        tree[i] += v;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
int sum(int i) {
    int ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 离散化函数：将原始数组的值映射到连续的小范围内
 *
 * @param nums 原始数组
 */
void discretize(int nums[], int n) {
```

```
    // 创建排序数组
    for (int i = 0; i < n; i++) {
        sorted[i] = nums[i];
    }

    // 手动排序（冒泡排序）
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (sorted[j] > sorted[j + 1]) {
                int temp = sorted[j];
                sorted[j] = sorted[j + 1];
                sorted[j + 1] = temp;
            }
        }
    }

    // 去重
    int uniqueCount = 1;
    for (int i = 1; i < n; i++) {
        if (sorted[i] != sorted[uniqueCount - 1]) {
            sorted[uniqueCount] = sorted[i];
            uniqueCount++;
        }
    }

    this->uniqueCount = uniqueCount;
    MAXN = uniqueCount + 1;
    // 初始化树状数组
    for (int i = 0; i < MAXN; i++) {
        tree[i] = 0;
    }
}

/**
 * 获取元素在离散化数组中的位置（使用二分查找）
 *
 * @param val 要查找的值
 * @return 该值在离散化数组中的位置
 */
int getId(int val, int sortedSize) {
    int left = 0, right = sortedSize - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
```

```
            if (sorted[mid] >= val) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return left + 1; // 树状数组下标从1开始
    }

public:
    /**
     * 计算右侧小于当前元素的个数
     *
     * @param nums 输入数组
     * @return 结果数组
     */
    void countSmaller(int nums[], int n, int result[]) {
        // 离散化处理
        discretize(nums, n);

        // 初始化结果数组
        for (int i = 0; i < n; i++) {
            result[i] = 0;
        }

        // 从右往左遍历数组
        for (int i = n - 1; i >= 0; i--) {
            // 获取当前元素在离散化数组中的位置
            int id = getId(nums[i], uniqueCount);
            // 查询比当前元素小的元素个数
            result[i] = sum(id - 1);
            // 将当前元素插入树状数组
            add(id, 1);
        }
    }
};

/**
 * 测试函数
 */
/*
 * 主函数用于测试
 * 由于编译环境限制，此处省略测试代码
```

```
 * 实际使用时可以直接调用 Solution 类的方法
 */

// 示例调用方式：
// Solution solution;
// int nums[] = {5, 2, 6, 1};
// int result[4];
// solution.countSmaller(nums, 4, result);
// result 数组中将包含结果[2,1,1,0]

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

文件: LeetCode315_CountSmallerNumbersAfterSelf.java
================================================
package class108;

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

/**
 * LeetCode 315. 计算右侧小于当前元素的个数
 * 题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/
 *
 * 题目描述:
 * 给你一个整数数组 nums ，按要求返回一个新数组 counts 。数组 counts 有该性质：
 * counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。
 *
 * 示例:
 * 输入: nums = [5,2,6,1]
 * 输出: [2,1,1,0]
 * 解释:
 * 5 的右侧有 2 个更小的元素 (2 和 1)
 * 2 的右侧有 1 个更小的元素 (1)
 * 6 的右侧有 1 个更小的元素 (1)
 * 1 的右侧有 0 个更小的元素
 *
 * 解题思路:
 * 使用树状数组 + 离散化来解决这个问题。
 * 1. 离散化：由于数值范围可能很大，需要先进行离散化处理，将数值映射到连续的小范围内
 * 2. 从右往左遍历数组：
 *    - 对于每个元素，查询树状数组中比它小的元素个数（即右侧小于当前元素的个数）
 *    - 将当前元素插入树状数组
 *
```

```
 * 时间复杂度：O(n log n)，其中 n 是数组长度
 * 空间复杂度：O(n)
 */

public class LeetCode315_CountSmallerNumbersAfterSelf {
    // 树状数组最大容量
    private int MAXN;

    // 树状数组，用于统计元素出现次数
    private int[] tree;

    // 离散化后的数组
    private int[] sorted;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    private int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param i 位置（从 1 开始）
     * @param v 增加的值
     */
    private void add(int i, int v) {
        // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while (i < MAXN) {
            tree[i] += v;
            // 移动到父节点
            i += lowbit(i);
        }
    }

    /**
     * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
     *
```

```
     * @param i 查询的结束位置
     * @return 前缀和
     */
    private int sum(int i) {
        int ans = 0;
        // 从位置 i 开始，沿着子节点路径向下累加
        while (i > 0) {
            ans += tree[i];
            // 移动到前一个相关区间
            i -= lowbit(i);
        }
        return ans;
    }

    /**
     * 离散化函数：将原始数组的值映射到连续的小范围内
     *
     * @param nums 原始数组
     */
    private void discretize(int[] nums) {
        // 创建排序数组
        sorted = Arrays.stream(nums).distinct().sorted().toArray();
        MAXN = sorted.length + 1;
        tree = new int[MAXN];
    }

    /**
     * 获取元素在离散化数组中的位置（使用二分查找）
     *
     * @param val 要查找的值
     * @return 该值在离散化数组中的位置
     */
    private int getId(int val) {
        int left = 0, right = sorted.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (sorted[mid] >= val) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return left + 1; // 树状数组下标从 1 开始
```

```
    }

    /**
     * 计算右侧小于当前元素的个数
     *
     * @param nums 输入数组
     * @return 结果数组
     */
    public List<Integer> countSmaller(int[] nums) {
        int n = nums.length;
        List<Integer> result = new ArrayList<>();

        // 离散化处理
        discretize(nums);

        // 从右往左遍历数组
        for (int i = n - 1; i >= 0; i--) {
            // 获取当前元素在离散化数组中的位置
            int id = getId(nums[i]);
            // 查询比当前元素小的元素个数
            result.add(sum(id - 1));
            // 将当前元素插入树状数组
            add(id, 1);
        }

        // 由于是从右往左遍历的，需要反转结果
        Collections.reverse(result);
        return result;
    }

    /**
     * 测试函数
     */
    public static void main(String[] args) {
        LeetCode315_CountSmallerNumbersAfterSelf solution = new
LeetCode315_CountSmallerNumbersAfterSelf();

        // 测试用例 1
        int[] nums1 = {5, 2, 6, 1};
        List<Integer> result1 = solution.countSmaller(nums1);
        System.out.println("输入: [5,2,6,1]");
        System.out.println("输出: " + result1);
        System.out.println("期望: [2,1,1,0]");
```

```
        System.out.println();

        // 测试用例 2
        int[] nums2 = {-1};
        List<Integer> result2 = solution.countSmaller(nums2);
        System.out.println("输入: [-1]");
        System.out.println("输出: " + result2);
        System.out.println("期望: [0]");
        System.out.println();

        // 测试用例 3
        int[] nums3 = {-1, -1};
        List<Integer> result3 = solution.countSmaller(nums3);
        System.out.println("输入: [-1,-1]");
        System.out.println("输出: " + result3);
        System.out.println("期望: [0,0]");
    }
}
```

==========

文件: LeetCode315_CountSmallerNumbersAfterSelf.py

==========

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

"""
LeetCode 315. 计算右侧小于当前元素的个数
题目链接: https://leetcode.cn/problems/count-of-smaller-numbers-after-self/

题目描述:
给你一个整数数组 nums ，按要求返回一个新数组 counts 。数组 counts 有该性质：
counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

示例:
输入: nums = [5,2,6,1]
输出: [2,1,1,0]
解释:
5 的右侧有 2 个更小的元素 (2 和 1)
2 的右侧有 1 个更小的元素 (1)
6 的右侧有 1 个更小的元素 (1)
1 的右侧有 0 个更小的元素
```

解题思路:
使用树状数组 + 离散化来解决这个问题。
1. 离散化：由于数值范围可能很大，需要先进行离散化处理，将数值映射到连续的小范围内
2. 从右往左遍历数组：
   - 对于每个元素，查询树状数组中比它小的元素个数（即右侧小于当前元素的个数）
   - 将当前元素插入树状数组

时间复杂度：O(n log n)，其中 n 是数组长度
空间复杂度：O(n)
"""

```
class Solution:
    def __init__(self):
        """
        初始化函数
        """
        self.tree = []
        self.sorted = []
        self.MAXN = 0

    def lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
        例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

        :param i: 输入数字
        :return: 最低位的 1 所代表的数值
        """
        return i & -i

    def add(self, i, v):
        """
        单点增加操作：在位置 i 上增加 v

        :param i: 位置（从 1 开始）
        :param v: 增加的值
        """
        # 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while i < self.MAXN:
            self.tree[i] += v
            # 移动到父节点
            i += self.lowbit(i)
```

```
def sum(self, i):
    """
    查询前缀和：计算从位置 1 到位置 i 的所有元素之和

    :param i: 查询的结束位置
    :return: 前缀和
    """
    ans = 0
    # 从位置 i 开始，沿着子节点路径向下累加
    while i > 0:
        ans += self.tree[i]
        # 移动到前一个相关区间
        i -= self.lowbit(i)
    return ans

def discretize(self, nums):
    """
    离散化函数：将原始数组的值映射到连续的小范围内

    :param nums: 原始数组
    """
    # 创建排序数组并去重
    self.sorted = sorted(list(set(nums)))
    self.MAXN = len(self.sorted) + 1
    self.tree = [0] * self.MAXN

def get_id(self, val):
    """
    获取元素在离散化数组中的位置（使用二分查找）

    :param val: 要查找的值
    :return: 该值在离散化数组中的位置
    """
    left, right = 0, len(self.sorted) - 1
    while left <= right:
        mid = (left + right) // 2
        if self.sorted[mid] >= val:
            right = mid - 1
        else:
            left = mid + 1
    return left + 1  # 树状数组下标从 1 开始
```

```
    def countSmaller(self, nums):
        """
        计算右侧小于当前元素的个数

        :param nums: 输入数组
        :return: 结果数组
        """
        n = len(nums)
        result = []

        # 离散化处理
        self.discretize(nums)

        # 从右往左遍历数组
        for i in range(n - 1, -1, -1):
            # 获取当前元素在离散化数组中的位置
            id = self.get_id(nums[i])
            # 查询比当前元素小的元素个数
            result.append(self.sum(id - 1))
            # 将当前元素插入树状数组
            self.add(id, 1)

        # 由于是从右往左遍历的，需要反转结果
        result.reverse()
        return result


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

    # 测试用例 1
    nums1 = [5, 2, 6, 1]
    result1 = solution.countSmaller(nums1)
    print("输入: [5,2,6,1]")
    print("输出: {}".format(result1))
    print("期望: [2,1,1,0]")
    print()

    # 测试用例 2
    nums2 = [-1]
```

```
    result2 = solution.countSmaller(nums2)
    print("输入: [-1]")
    print("输出: {}".format(result2))
    print("期望: [0]")
    print()

    # 测试用例 3
    nums3 = [-1, -1]
    result3 = solution.countSmaller(nums3)
    print("输入: [-1,-1]")
    print("输出: {}".format(result3))
    print("期望: [0,0]")


# 运行测试
if __name__ == "__main__":
    main()

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

文件: P1908_InversePairs.cpp
==========================================
/**
 * 洛谷 P1908 逆序对
 * 题目链接: https://www.luogu.com.cn/problem/P1908
 *
 * 题目描述:
 * 给定一个序列 a，求有多少对 (i, j) 满足 i < j 且 a[i] > a[j]。
 *
 * 输入格式:
 * 第一行包含一个正整数 n，表示序列长度。
 * 第二行包含 n 个整数，表示序列 a。
 *
 * 输出格式:
 * 输出一行一个整数表示逆序对个数。
 *
 * 样例输入:
 * 6
 * 5 4 2 6 3 1
 *
 * 样例输出:
 * 11
 *
```

```
 * 解题思路:
 * 使用树状数组 + 离散化来计算逆序对个数。
 * 离散化是为了处理大数值的情况，将原始数值映射到连续的小范围内。
 * 从右往左遍历数组，对于每个元素，查询树状数组中比它小的元素个数，
 * 然后将当前元素插入树状数组。
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */

#include <cstdio>
#include <algorithm>
using namespace std;

const int MAXN = 500001;

// 树状数组，用于统计元素出现次数
int tree[MAXN];

// 数组长度
int n;

// 原始数组和离散化后的数组
int arr[MAXN];
int sorted[MAXN];

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
int lowbit(int i) {
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
void add(int i, int v) {
```

```
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= n) {
        tree[i] += v;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
int sum(int i) {
    int ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 离散化函数：将原始数组的值映射到连续的小范围内
 */
void discretize() {
    // 复制并排序数组
    for (int i = 1; i <= n; i++) {
        sorted[i] = arr[i];
    }
    sort(sorted + 1, sorted + n + 1);

    // 去重
    int uniqueCount = 1;
    for (int i = 2; i <= n; i++) {
        if (sorted[i] != sorted[uniqueCount]) {
            sorted[++uniqueCount] = sorted[i];
        }
    }
```

```
    // 更新 sorted 数组长度
    sorted[0] = uniqueCount;
}

/**
 * 获取元素在离散化数组中的位置（使用二分查找）
 *
 * @param val 要查找的值
 * @return 该值在离散化数组中的位置
 */
int getId(int val) {
    int left = 1, right = sorted[0];
    while (left <= right) {
        int mid = (left + right) / 2;
        if (sorted[mid] >= val) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 主函数：处理输入输出和调用相关操作
 */
int main() {
    // 读取数组长度 n
    scanf("%d", &n);

    // 读取数组元素
    for (int i = 1; i <= n; i++) {
        scanf("%d", &arr[i]);
    }

    // 离散化处理
    discretize();

    long long ans = 0;
    // 从右往左遍历数组
    for (int i = n; i >= 1; i--) {
        // 获取当前元素在离散化数组中的位置
        int id = getId(arr[i]);
```

```
        // 查询比当前元素小的元素个数（即逆序对个数）
        ans += sum(id - 1);
        // 将当前元素插入树状数组
        add(id, 1);
    }

    // 输出结果
    printf("%lld\n", ans);

    return 0;
}
```

==========

文件: P1908_InversePairs.java

==========

```
package class108;

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

/**
 * 洛谷 P1908 逆序对
 * 题目链接: https://www.luogu.com.cn/problem/P1908
 *
 * 题目描述:
 * 给定一个序列 a，求有多少对 (i, j) 满足 i < j 且 a[i] > a[j]。
 *
 * 输入格式:
 * 第一行包含一个正整数 n，表示序列长度。
 * 第二行包含 n 个整数，表示序列 a。
 *
 * 输出格式:
 * 输出一行一个整数表示逆序对个数。
 *
 * 样例输入:
 * 6
 * 5 4 2 6 3 1
 *
 * 样例输出:
 * 11
 *
 * 解题思路:
```

```
 * 使用树状数组 + 离散化来计算逆序对个数。
 * 离散化是为了处理大数值的情况，将原始数值映射到连续的小范围内。
 * 从右往左遍历数组，对于每个元素，查询树状数组中比它小的元素个数，
 * 然后将当前元素插入树状数组。
 * 时间复杂度：O(n log n)
 * 空间复杂度：O(n)
 */

public class P1908_InversePairs {
    // 树状数组最大容量
    public static int MAXN = 500001;

    // 树状数组，用于统计元素出现次数
    public static int[] tree = new int[MAXN];

    // 数组长度
    public static int n;

    // 原始数组和离散化后的数组
    public static int[] arr = new int[MAXN];
    public static int[] sorted = new int[MAXN];

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    public static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param i 位置（从 1 开始）
     * @param v 增加的值
     */
    public static void add(int i, int v) {
        // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while (i <= n) {
            tree[i] += v;
```

```
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
public static int sum(int i) {
    int ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 离散化函数：将原始数组的值映射到连续的小范围内
 */
public static void discretize() {
    // 复制并排序数组
    for (int i = 1; i <= n; i++) {
        sorted[i] = arr[i];
    }
    Arrays.sort(sorted, 1, n + 1);

    // 去重
    int uniqueCount = 1;
    for (int i = 2; i <= n; i++) {
        if (sorted[i] != sorted[uniqueCount]) {
            sorted[++uniqueCount] = sorted[i];
        }
    }

    // 更新 sorted 数组长度
    sorted[0] = uniqueCount;
}
```

```
/**
 * 获取元素在离散化数组中的位置（使用二分查找）
 *
 * @param val 要查找的值
 * @return 该值在离散化数组中的位置
 */
public static int getId(int val) {
    int left = 1, right = sorted[0];
    while (left <= right) {
        int mid = (left + right) / 2;
        if (sorted[mid] >= val) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

/**
 * 主函数：处理输入输出和调用相关操作
 */
public static void main(String[] args) throws IOException {
    // 使用高效的 IO 处理方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度 n
    in.nextToken();
    n = (int) in.nval;

    // 读取数组元素
    for (int i = 1; i <= n; i++) {
        in.nextToken();
        arr[i] = (int) in.nval;
    }

    // 离散化处理
    discretize();

    long ans = 0;
```

```
        // 从右往左遍历数组
        for (int i = n; i >= 1; i--) {
            // 获取当前元素在离散化数组中的位置
            int id = getId(arr[i]);
            // 查询比当前元素小的元素个数（即逆序对个数）
            ans += sum(id - 1);
            // 将当前元素插入树状数组
            add(id, 1);
        }

        // 输出结果
        out.println(ans);

        // 刷新输出缓冲区并关闭 IO 流
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: P1908_InversePairs.py

==========

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

"""
```

洛谷 P1908 逆序对

题目链接: https://www.luogu.com.cn/problem/P1908

题目描述:

给定一个序列 a，求有多少对 (i, j) 满足 i < j 且 a[i] > a[j]。

输入格式:

第一行包含一个正整数 n，表示序列长度。

第二行包含 n 个整数，表示序列 a。

输出格式:

输出一行一个整数表示逆序对个数。

样例输入:

6

5 4 2 6 3 1

样例输出:
11

解题思路:
使用树状数组 + 离散化来计算逆序对个数。
离散化是为了处理大数值的情况，将原始数值映射到连续的小范围内。
从右往左遍历数组，对于每个元素，查询树状数组中比它小的元素个数，
然后将当前元素插入树状数组。
时间复杂度：O(n log n)
空间复杂度：O(n)
"""

```
class BinaryIndexTree:
    """
    树状数组类，用于高效处理单点修改和前缀和查询
    """

    def __init__(self, n):
        """
        初始化树状数组
        :param n: 数组大小
        """
        # 树状数组最大容量
        self.MAXN = n + 1
        # 树状数组，存储前缀和信息，索引从 1 开始
        self.tree = [0] * self.MAXN

    def lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
        例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

        :param i: 输入数字
        :return: 最低位的 1 所代表的数值
        """
        return i & (-i)

    def add(self, i, v):
        """
        单点增加操作：在位置 i 上增加 v
```

```
        :param i: 位置（从 1 开始）
        :param v: 增加的值
        """
        # 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while i < self.MAXN:
            self.tree[i] += v
            # 移动到父节点
            i += self.lowbit(i)

    def sum(self, i):
        """
        查询前缀和：计算从位置 1 到位置 i 的所有元素之和

        :param i: 查询的结束位置
        :return: 前缀和
        """
        ans = 0
        # 从位置 i 开始，沿着子节点路径向下累加
        while i > 0:
            ans += self.tree[i]
            # 移动到前一个相关区间
            i -= self.lowbit(i)
        return ans


def main():
    """
    主函数：处理输入输出和调用相关操作
    """
    # 读取数组长度 n
    n = int(input())

    # 读取数组元素
    arr = list(map(int, input().split()))

    # 离散化处理
    # 1. 获取所有不重复的元素并排序
    sorted_vals = sorted(set(arr))

    # 2. 建立值到索引的映射
    val_to_id = {val: idx + 1 for idx, val in enumerate(sorted_vals)}
```

```
    # 创建树状数组实例
    bit = BinaryIndexTree(len(sorted_vals))

    ans = 0
    # 从右往左遍历数组
    for i in range(n - 1, -1, -1):
        # 获取当前元素在离散化数组中的位置
        id = val_to_id[arr[i]]
        # 查询比当前元素小的元素个数（即逆序对个数）
        ans += bit.sum(id - 1)
        # 将当前元素插入树状数组
        bit.add(id, 1)

    # 输出结果
    print(ans)


# 程序入口
if __name__ == "__main__":
    main()

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

文件: P3368_BitTree2.cpp
==============================================
/**
 * 洛谷 P3368 【模板】树状数组 2
 * 题目链接: https://www.luogu.com.cn/problem/P3368
 *
 * 题目描述:
 * 给定一个数列，需要进行下面两种操作：
 * 1. 将某区间加上一个值
 * 2. 求出某一个数的值
 *
 * 输入格式:
 * 第一行包含两个正整数 n, m，分别表示该数列数字的个数和总操作的次数。
 * 第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。
 * 接下来 m 行每行包含 3 或 4 个整数，表示一个操作：
 * - 如果是 1 l r k：表示将区间 [l,r] 加上 k
 * - 如果是 2 x：表示求出第 x 项的值
 *
 * 输出格式:
 * 对于每个 2 操作，输出一行一个整数表示答案。
```

```
 *
 * 样例输入:
 * 5 5
 * 1 5 4 2 3
 * 1 2 4 2
 * 2 3
 * 1 1 5 -1
 * 1 3 5 7
 * 2 4
 *
 * 样例输出:
 * 7
 * 9
 *
 * 解题思路:
 * 使用树状数组实现区间修改和单点查询，采用差分数组的思想
 * 差分数组的性质：原数组的区间修改等价于差分数组的单点修改
 * 原数组的单点查询等价于差分数组的前缀和查询
 * 时间复杂度：
 * - 区间修改: O(log n)
 * - 单点查询: O(log n)
 * 空间复杂度: O(n)
 */

#include <iostream>
using namespace std;

const int MAXN = 500001;

// 树状数组，维护差分数组的前缀和
long long tree[MAXN];

// 数组长度和操作次数
int n, m;

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
int lowbit(int i) {
```

```
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
void add(int i, long long v) {
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= n) {
        tree[i] += v;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
long long sum(int i) {
    long long ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 区间增加操作：在区间[l,r]上每个元素都增加 v
 * 利用差分数组的思想：
 * 在差分数组的第 l 个位置加上 v，在第 r+1 个位置减去 v
 *
 * @param l 区间起始位置
 * @param r 区间结束位置
 * @param v 增加的值
```

```
 */
void rangeAdd(int l, int r, long long v) {
    add(l, v);
    add(r + 1, -v);
}

/**
 * 主函数：处理输入输出和调用相关操作
 */
int main() {
    // 读取数组长度 n 和操作次数 m
    cin >> n >> m;

    // 读取初始数组并构建差分数组
    long long pre = 0;
    for (int i = 1; i <= n; i++) {
        long long cur;
        cin >> cur;
        // 构建差分数组：差分数组第 i 位 = 原数组第 i 位 - 原数组第 i-1 位
        add(i, cur - pre);
        pre = cur;
    }

    // 处理 m 次操作
    for (int i = 1, a, b, c; i <= m; i++) {
        cin >> a;

        if (a == 1) {
            // 操作 1：区间[l,r]增加 k
            cin >> b >> c >> pre;
            rangeAdd(b, c, pre);
        } else {
            // 操作 2：查询位置 b 的值（即差分数组的前缀和）
            cin >> b;
            cout << sum(b) << endl;
        }
    }

    return 0;
}
```

==========

文件: P3368_BitTree2.java

```
==========================================
package class108;

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

/**
 * 洛谷 P3368 【模板】树状数组 2
 * 题目链接: https://www.luogu.com.cn/problem/P3368
 *
 * 题目描述:
 * 给定一个数列，需要进行下面两种操作：
 * 1. 将某区间加上一个值
 * 2. 求出某一个数的值
 *
 * 输入格式:
 * 第一行包含两个正整数 n, m，分别表示该数列数字的个数和总操作的次数。
 * 第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。
 * 接下来 m 行每行包含 3 或 4 个整数，表示一个操作：
 * - 如果是 1 l r k：表示将区间 [l,r] 加上 k
 * - 如果是 2 x：表示求出第 x 项的值
 *
 * 输出格式:
 * 对于每个 2 操作，输出一行一个整数表示答案。
 *
 * 样例输入:
 * 5 5
 * 1 5 4 2 3
 * 1 2 4 2
 * 2 3
 * 1 1 5 -1
 * 1 3 5 7
 * 2 4
 *
 * 样例输出:
 * 7
 * 9
 *
 * 解题思路:
 * 使用树状数组实现区间修改和单点查询，采用差分数组的思想
 * 差分数组的性质：原数组的区间修改等价于差分数组的单点修改
 * 原数组的单点查询等价于差分数组的前缀和查询
```

```
 * 时间复杂度：
 * - 区间修改: O(log n)
 * - 单点查询: O(log n)
 * 空间复杂度: O(n)
 */

public class P3368_BitTree2 {
    // 树状数组最大容量
    public static int MAXN = 500001;

    // 树状数组，维护差分数组的前缀和
    public static long[] tree = new long[MAXN];

    // 数组长度和操作次数
    public static int n, m;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    public static int lowbit(int i) {
        return i & -i;
    }

    /**
     * 单点增加操作：在位置 i 上增加 v
     *
     * @param i 位置（从 1 开始）
     * @param v 增加的值
     */
    public static void add(int i, long v) {
        // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while (i <= n) {
            tree[i] += v;
            // 移动到父节点
            i += lowbit(i);
        }
    }

    /**
```

```
     * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
     *
     * @param i 查询的结束位置
     * @return 前缀和
     */
    public static long sum(int i) {
        long ans = 0;
        // 从位置 i 开始，沿着子节点路径向下累加
        while (i > 0) {
            ans += tree[i];
            // 移动到前一个相关区间
            i -= lowbit(i);
        }
        return ans;
    }

    /**
     * 区间增加操作：在区间[l,r]上每个元素都增加 v
     * 利用差分数组的思想：
     * 在差分数组的第 l 个位置加上 v，在第 r+1 个位置减去 v
     *
     * @param l 区间起始位置
     * @param r 区间结束位置
     * @param v 增加的值
     */
    public static void rangeAdd(int l, int r, long v) {
        add(l, v);
        add(r + 1, -v);
    }

    /**
     * 主函数：处理输入输出和调用相关操作
     */
    public static void main(String[] args) throws IOException {
        // 使用高效的 IO 处理方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取数组长度 n 和操作次数 m
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
```

```
        m = (int) in.nval;

        // 读取初始数组并构建差分数组
        long pre = 0;
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            long cur = (long) in.nval;
            // 构建差分数组：差分数组第 i 位 = 原数组第 i 位 - 原数组第 i-1 位
            add(i, cur - pre);
            pre = cur;
        }

        // 处理 m 次操作
        for (int i = 1, a, b, c; i <= m; i++) {
            in.nextToken();
            a = (int) in.nval;

            if (a == 1) {
                // 操作 1：区间[l,r]增加 k
                in.nextToken();
                b = (int) in.nval;
                in.nextToken();
                c = (int) in.nval;
                in.nextToken();
                long k = (long) in.nval;
                rangeAdd(b, c, k);
            } else {
                // 操作 2：查询位置 b 的值（即差分数组的前缀和）
                in.nextToken();
                b = (int) in.nval;
                out.println(sum(b));
            }
        }

        // 刷新输出缓冲区并关闭 IO 流
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: P3368_BitTree2.py
==========
#!/usr/bin/env python3
-*- coding: utf-8 -*-

"""
洛谷 P3368 【模板】树状数组 2
题目链接: https://www.luogu.com.cn/problem/P3368

题目描述:
给定一个数列，需要进行下面两种操作：
1. 将某区间加上一个值
2. 求出某一个数的值

输入格式:
第一行包含两个正整数 n, m，分别表示该数列数字的个数和总操作的次数。
第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。
接下来 m 行每行包含 3 或 4 个整数，表示一个操作：
- 如果是 1 l r k：表示将区间 [l,r] 加上 k
- 如果是 2 x：表示求出第 x 项的值

输出格式:
对于每个 2 操作，输出一行一个整数表示答案。

样例输入:
5 5
1 5 4 2 3
1 2 4 2
2 3
1 1 5 -1
1 3 5 7
2 4

样例输出:
7
9

解题思路:
使用树状数组实现区间修改和单点查询，采用差分数组的思想
差分数组的性质：原数组的区间修改等价于差分数组的单点修改
原数组的单点查询等价于差分数组的前缀和查询
时间复杂度：
- 区间修改: O(log n)

- 单点查询: O(log n)
空间复杂度: O(n)
"""

```
class BinaryIndexTree:
    """
    树状数组类，用于高效处理区间修改和单点查询（使用差分数组）
    """

    def __init__(self, n):
        """
        初始化树状数组
        :param n: 数组大小
        """
        # 树状数组最大容量
        self.MAXN = n + 1
        # 树状数组，维护差分数组的前缀和
        self.tree = [0] * self.MAXN

    def lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
        例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

        :param i: 输入数字
        :return: 最低位的 1 所代表的数值
        """
        return i & (-i)

    def add(self, i, v):
        """
        单点增加操作：在位置 i 上增加 v

        :param i: 位置（从 1 开始）
        :param v: 增加的值
        """
        # 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
        while i < self.MAXN:
            self.tree[i] += v
            # 移动到父节点
            i += self.lowbit(i)
```

```
    def sum(self, i):
        """
        查询前缀和：计算从位置 1 到位置 i 的所有元素之和

        :param i: 查询的结束位置
        :return: 前缀和
        """
        ans = 0
        # 从位置 i 开始，沿着子节点路径向下累加
        while i > 0:
            ans += self.tree[i]
            # 移动到前一个相关区间
            i -= self.lowbit(i)
        return ans

    def range_add(self, l, r, v):
        """
        区间增加操作：在区间[l,r]上每个元素都增加 v
        利用差分数组的思想：
        在差分数组的第 l 个位置加上 v，在第 r+1 个位置减去 v

        :param l: 区间起始位置
        :param r: 区间结束位置
        :param v: 增加的值
        """
        self.add(l, v)
        self.add(r + 1, -v)


def main():
    """
    主函数：处理输入输出和调用相关操作
    """
    # 读取数组长度 n 和操作次数 m
    n, m = map(int, input().split())

    # 创建树状数组实例
    bit = BinaryIndexTree(n)

    # 读取初始数组并构建差分数组
    values = list(map(int, input().split()))
    pre = 0
    for i in range(1, n + 1):
```

```
        # 构建差分数组：差分数组第 i 位 = 原数组第 i 位 - 原数组第 i-1 位
        bit.add(i, values[i - 1] - pre)
        pre = values[i - 1]

    # 处理 m 次操作
    for _ in range(m):
        operation = list(map(int, input().split()))
        if operation[0] == 1:
            # 操作 1：区间[operation[1], operation[2]]增加 operation[3]
            bit.range_add(operation[1], operation[2], operation[3])
        else:
            # 操作 2：查询位置 operation[1]的值（即差分数组的前缀和）
            print(bit.sum(operation[1]))



# 程序入口
if __name__ == "__main__":
    main()

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

文件: P3374_BitTree1.cpp
==========================================
/**
 * 洛谷 P3374 【模板】树状数组 1
 * 题目链接: https://www.luogu.com.cn/problem/P3374
 *
 * 题目描述:
 * 给定一个数列，需要进行下面两种操作：
 * 1. 将某一个数加上一个值
 * 2. 求出某区间内所有数的和
 *
 * 输入格式:
 * 第一行包含两个正整数 n, m，分别表示该数列数字的个数和总操作的次数。
 * 第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。
 * 接下来 m 行每行包含 3 个整数，表示一个操作：
 * - 如果是 1 x k：表示将第 x 个数加上 k
 * - 如果是 2 x y：表示求出第 x 到第 y 项的和
 *
 * 输出格式:
 * 对于每个 2 操作，输出一行一个整数表示答案。
 *
 * 样例输入:
```

```
 * 5 5
 * 1 5 4 2 3
 * 1 1 3
 * 2 2 4
 * 1 2 4
 * 2 1 5
 * 2 2 4
 *
 * 样例输出:
 * 11
 * 18
 * 16
 *
 * 解题思路:
 * 使用树状数组（Binary Indexed Tree/Fenwick Tree）实现单点修改和区间查询
 * 时间复杂度：
 * - 单点修改: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(n)
 */

#include <cstdio>
using namespace std;

const int MAXN = 500001;

// 树状数组，存储前缀和信息
int tree[MAXN];

// 数组长度和操作次数
int n, m;

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
int lowbit(int i) {
    return i & -i;
}
```

```
/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
void add(int i, int v) {
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= n) {
        tree[i] += v;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
int sum(int i) {
    int ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
        i -= lowbit(i);
    }
    return ans;
}

/**
 * 区间查询：计算从位置 l 到位置 r 的所有元素之和
 * 利用前缀和的性质：[l,r]的和 = [1,r]的和 - [1,l-1]的和
 *
 * @param l 区间起始位置
 * @param r 区间结束位置
 * @return 区间和
 */
int range(int l, int r) {
    return sum(r) - sum(l - 1);
}
```

```
/**
 * 主函数：处理输入输出和调用相关操作
 */
int main() {
    // 读取数组长度 n 和操作次数 m
    scanf("%d%d", &n, &m);

    // 读取初始数组并构建树状数组
    for (int i = 1, v; i <= n; i++) {
        scanf("%d", &v);
        // 初始构建相当于在每个位置上增加初始值
        add(i, v);
    }

    // 处理 m 次操作
    for (int i = 1, a, b, c; i <= m; i++) {
        scanf("%d%d%d", &a, &b, &c);

        if (a == 1) {
            // 操作 1：在位置 b 上增加 c
            add(b, c);
        } else {
            // 操作 2：查询区间[b,c]的和
            printf("%d\n", range(b, c));
        }
    }

    return 0;
}
```

==========

文件: P3374_BitTree1.java

==========

```
// package class108;

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

/**
 * 洛谷 P3374 【模板】树状数组 1
 * 题目链接: https://www.luogu.com.cn/problem/P3374
```

```
 *
 * 题目描述:
 * 给定一个数列，需要进行下面两种操作：
 * 1. 将某一个数加上一个值
 * 2. 求出某区间内所有数的和
 *
 * 输入格式:
 * 第一行包含两个正整数 n, m，分别表示该数列数字的个数和总操作的次数。
 * 第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。
 * 接下来 m 行每行包含 3 个整数，表示一个操作：
 * - 如果是 1 x k：表示将第 x 个数加上 k
 * - 如果是 2 x y：表示求出第 x 到第 y 项的和
 *
 * 输出格式:
 * 对于每个 2 操作，输出一行一个整数表示答案。
 *
 * 样例输入:
 * 5 5
 * 1 5 4 2 3
 * 1 1 3
 * 2 2 4
 * 1 2 4
 * 2 1 5
 * 2 2 4
 *
 * 样例输出:
 * 11
 * 18
 * 16
 *
 * 解题思路:
 * 使用树状数组（Binary Indexed Tree/Fenwick Tree）实现单点修改和区间查询
 * 时间复杂度：
 * - 单点修改: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(n)
 */

public class P3374_BitTree1 {
    // 树状数组最大容量
    public static int MAXN = 500001;

    // 树状数组，存储前缀和信息
```

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

// 数组长度和操作次数
public static int n, m;

/**
 * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
 * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
 *
 * @param i 输入数字
 * @return 最低位的 1 所代表的数值
 */
public static int lowbit(int i) {
    return i & -i;
}

/**
 * 单点增加操作：在位置 i 上增加 v
 *
 * @param i 位置（从 1 开始）
 * @param v 增加的值
 */
public static void add(int i, int v) {
    // 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while (i <= n) {
        tree[i] += v;
        // 移动到父节点
        i += lowbit(i);
    }
}

/**
 * 查询前缀和：计算从位置 1 到位置 i 的所有元素之和
 *
 * @param i 查询的结束位置
 * @return 前缀和
 */
public static int sum(int i) {
    int ans = 0;
    // 从位置 i 开始，沿着子节点路径向下累加
    while (i > 0) {
        ans += tree[i];
        // 移动到前一个相关区间
```

```
            i -= lowbit(i);
        }
        return ans;
    }

    /**
     * 区间查询：计算从位置 l 到位置 r 的所有元素之和
     * 利用前缀和的性质：[l,r]的和 = [1,r]的和 - [1,l-1]的和
     *
     * @param l 区间起始位置
     * @param r 区间结束位置
     * @return 区间和
     */
    public static int range(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    /**
     * 主函数：处理输入输出和调用相关操作
     */
    public static void main(String[] args) throws IOException {
        // 使用高效的 IO 处理方式
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer in = new StreamTokenizer(br);
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

        // 读取数组长度 n 和操作次数 m
        in.nextToken();
        n = (int) in.nval;
        in.nextToken();
        m = (int) in.nval;

        // 读取初始数组并构建树状数组
        for (int i = 1, v; i <= n; i++) {
            in.nextToken();
            v = (int) in.nval;
            // 初始构建相当于在每个位置上增加初始值
            add(i, v);
        }

        // 处理 m 次操作
        for (int i = 1, a, b, c; i <= m; i++) {
            in.nextToken();
```

```
            a = (int) in.nval;
            in.nextToken();
            b = (int) in.nval;
            in.nextToken();
            c = (int) in.nval;

            if (a == 1) {
                // 操作 1：在位置 b 上增加 c
                add(b, c);
            } else {
                // 操作 2：查询区间[b,c]的和
                out.println(range(b, c));
            }
        }

        // 刷新输出缓冲区并关闭 IO 流
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: P3374_BitTree1.py

==========

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

"""
洛谷 P3374 【模板】树状数组 1
题目链接: https://www.luogu.com.cn/problem/P3374

题目描述:
给定一个数列，需要进行下面两种操作：
1. 将某一个数加上一个值
2. 求出某区间内所有数的和

输入格式:
第一行包含两个正整数 n, m，分别表示该数列数字的个数和总操作的次数。
第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。
接下来 m 行每行包含 3 个整数，表示一个操作：
- 如果是 1 x k：表示将第 x 个数加上 k
```

- 如果是 2 x y：表示求出第 x 到第 y 项的和

输出格式:
对于每个 2 操作，输出一行一个整数表示答案。

样例输入:
5 5
1 5 4 2 3
1 1 3
2 2 4
1 2 4
2 1 5
2 2 4

样例输出:
11
18
16

解题思路:
使用树状数组（Binary Indexed Tree/Fenwick Tree）实现单点修改和区间查询
时间复杂度：
- 单点修改: O(log n)
- 区间查询: O(log n)
空间复杂度: O(n)
"""

```
class BinaryIndexTree:
    """
    树状数组类，用于高效处理单点修改和前缀和查询
    """

    def __init__(self, n):
        """
        初始化树状数组
        :param n: 数组大小
        """
        # 树状数组最大容量
        self.MAXN = n + 1
        # 树状数组，存储前缀和信息，索引从 1 开始
        self.tree = [0] * self.MAXN
```

```
def lowbit(self, i):
    """
    lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
    例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

    :param i: 输入数字
    :return: 最低位的 1 所代表的数值
    """
    return i & (-i)

def add(self, i, v):
    """
    单点增加操作：在位置 i 上增加 v

    :param i: 位置（从 1 开始）
    :param v: 增加的值
    """
    # 从位置 i 开始，沿着父节点路径向上更新所有相关的节点
    while i < self.MAXN:
        self.tree[i] += v
        # 移动到父节点
        i += self.lowbit(i)

def sum(self, i):
    """
    查询前缀和：计算从位置 1 到位置 i 的所有元素之和

    :param i: 查询的结束位置
    :return: 前缀和
    """
    ans = 0
    # 从位置 i 开始，沿着子节点路径向下累加
    while i > 0:
        ans += self.tree[i]
        # 移动到前一个相关区间
        i -= self.lowbit(i)
    return ans

def range_sum(self, l, r):
    """
    区间查询：计算从位置 l 到位置 r 的所有元素之和
    利用前缀和的性质：[l,r]的和 = [1,r]的和 - [1,l-1]的和
```

```
        :param l: 区间起始位置
        :param r: 区间结束位置
        :return: 区间和
        """
        return self.sum(r) - self.sum(l - 1)


def main():
    """
    主函数：处理输入输出和调用相关操作
    """
    # 读取数组长度 n 和操作次数 m
    n, m = map(int, input().split())

    # 创建树状数组实例
    bit = BinaryIndexTree(n)

    # 读取初始数组并构建树状数组
    values = list(map(int, input().split()))
    for i in range(1, n + 1):
        # 初始构建相当于在每个位置上增加初始值
        bit.add(i, values[i - 1])

    # 处理 m 次操作
    for _ in range(m):
        operation = list(map(int, input().split()))
        if operation[0] == 1:
            # 操作 1：在位置 operation[1]上增加 operation[2]
            bit.add(operation[1], operation[2])
        else:
            # 操作 2：查询区间[operation[1], operation[2]]的和
            print(bit.range_sum(operation[1], operation[2]))


# 程序入口
if __name__ == "__main__":
    main()

==========

文件: POJ2155_Matrix.cpp
==========
/*
```

```
 * POJ 2155 Matrix - 二维树状数组区间取反单点查询问题
 * 题目链接: http://poj.org/problem?id=2155
 *
 * 题目描述:
 * 给定一个 N×N 的矩阵，初始时所有元素都为 0。
 * 有两种操作：
 * 1. "C x1 y1 x2 y2"：将左上角为 (x1,y1)、右下角为 (x2,y2) 的子矩阵中的每个元素取反（0 变 1，1 变 0）
 * 2. "Q x y"：查询位置 (x,y) 的值
 *
 * 解题思路深度分析:
 *
 * 这个问题是二维树状数组的经典应用 - 区间更新（取反）、单点查询的场景。
 *
 * 1. 取反操作的数学表示：
 *    由于每次取反就是将元素值增加 1 然后模 2（0+1=1, 1+1=2≡0 mod2），
 *    所以我们可以将取反操作转化为对区间内每个元素加 1，最后查询时模 2。
 *
 * 2. 二维差分数组的应用：
 *    要实现区间加 1、单点查询，可以使用二维差分数组：
 *    - 对于矩形区域(x1,y1)到(x2,y2)的加 1 操作，只需要在差分数组的四个角进行更新：
 *      d[x1][y1] += 1
 *      d[x1][y2+1] -= 1
 *      d[x2+1][y1] -= 1
 *      d[x2+1][y2+1] += 1
 *    - 这四个点的更新可以保证，当计算原数组某点的值（差分数组前缀和）时，
 *      只有在(x1,y1)到(x2,y2)矩形内的点才会被加 1
 *
 * 3. 树状数组优化：
 *    二维树状数组用来高效维护二维差分数组，支持：
 *    - 单点更新操作 O(log n * log n)
 *    - 二维前缀和查询 O(log n * log n)
 *
 * 4. 单点查询的实现：
 *    查询位置(x,y)的值实际上是查询差分数组从(1,1)到(x,y)的二维前缀和对 2 取模的结果
 *    这是因为每次更新的影响会通过差分数组传播到所有受影响的位置
 *
 * 时间复杂度：
 * - 区间更新: O(log n * log n) - 四次单点更新操作
 * - 单点查询: O(log n * log n) - 一次二维前缀和查询
 * 空间复杂度: O(n * n)
 *
 * 与线段树对比：
```

```
 * - 树状数组实现更简洁，常数更小
 * - 线段树虽然功能更强大，但在此问题中树状数组已足够且效率更高
 */

class POJ2155_Matrix {
private:
    int tree[1001][1001];  // 二维树状数组
    int n;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    int lowbit(int i) {
        return i & -i;
    }

public:
    /**
     * 二维树状数组初始化
     *
     * @param n 矩阵大小
     */
    POJ2155_Matrix(int n) {
        this->n = n;
        // 初始化二维树状数组
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= n; j++) {
                tree[i][j] = 0;
            }
        }
    }

    /**
     * 二维树状数组单点增加操作
     *
     * @param x x 坐标（从 1 开始）
     * @param y y 坐标（从 1 开始）
     * @param v 增加的值
     */
```

```
void add(int x, int y, int v) {
    for (int i = x; i <= n; i += lowbit(i)) {
        for (int j = y; j <= n; j += lowbit(j)) {
            tree[i][j] += v;
        }
    }
}

/**
 * 二维树状数组前缀和查询：计算从(1,1)到(x,y)的矩形区域内所有元素的和
 *
 * @param x x 坐标（从 1 开始）
 * @param y y 坐标（从 1 开始）
 * @return 前缀和
 */
int sum(int x, int y) {
    int ans = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            ans += tree[i][j];
        }
    }
    return ans;
}

/**
 * 区间更新操作：将左上角为(x1,y1)、右下角为(x2,y2)的子矩阵中的每个元素取反
 *
 * @param x1 左上角 x 坐标
 * @param y1 左上角 y 坐标
 * @param x2 右下角 x 坐标
 * @param y2 右下角 y 坐标
 */
void update(int x1, int y1, int x2, int y2) {
    add(x1, y1, 1);
    add(x1, y2 + 1, -1);
    add(x2 + 1, y1, -1);
    add(x2 + 1, y2 + 1, 1);
}

/**
 * 单点查询操作：查询位置(x,y)的值
 *
```

```
     * @param x x坐标
     * @param y y坐标
     * @return 位置(x,y)的值
     */
    int query(int x, int y) {
        return sum(x, y) % 2;
    }
};

/*
 * 完整主函数实现
 * 处理POJ 2155的输入输出格式
 */
#include <iostream>
using namespace std;

int main() {
    int T;  // 测试用例数
    cin >> T;
    while (T--) {
        int N, Q;  // N:矩阵大小, Q:操作数
        cin >> N >> Q;

        POJ2155_Matrix matrix(N);

        while (Q--) {
            char op;  // 操作类型
            cin >> op;

            if (op == 'C') {  // 区间更新操作
                int x1, y1, x2, y2;
                cin >> x1 >> y1 >> x2 >> y2;
                matrix.update(x1, y1, x2, y2);
            } else if (op == 'Q') {  // 单点查询操作
                int x, y;
                cin >> x >> y;
                cout << matrix.query(x, y) << endl;
            }
        }

        if (T > 0) cout << endl;  // 不同测试用例之间输出一个空行
    }
    return 0;
```

```
}

/*
 * 以下是 Java 实现的 POJ2155 代码
 *
 * import java.io.*;
 * import java.util.*;
 *
 * public class Main {
 *     static class FenwickTree2D {
 *         private int[][] tree;
 *         private int n;
 *
 *         public FenwickTree2D(int n) {
 *             this.n = n;
 *             tree = new int[n + 2][n + 2];  // 索引从 1 开始，预留额外空间避免越界
 *         }
 *
 *         private int lowbit(int x) {
 *             return x & -x;
 *         }
 *
 *         public void add(int x, int y, int val) {
 *             for (int i = x; i <= n; i += lowbit(i)) {
 *                 for (int j = y; j <= n; j += lowbit(j)) {
 *                     tree[i][j] += val;
 *                 }
 *             }
 *         }
 *
 *         public int sum(int x, int y) {
 *             int ans = 0;
 *             for (int i = x; i > 0; i -= lowbit(i)) {
 *                 for (int j = y; j > 0; j -= lowbit(j)) {
 *                     ans += tree[i][j];
 *                 }
 *             }
 *             return ans;
 *         }
 *
 *         public void update(int x1, int y1, int x2, int y2) {
 *             add(x1, y1, 1);
 *             add(x1, y2 + 1, -1);
```

```
*            add(x2 + 1, y1, -1);
*            add(x2 + 1, y2 + 1, 1);
*        }
*
*        public int query(int x, int y) {
*            return sum(x, y) % 2;
*        }
*    }
*
*    public static void main(String[] args) throws IOException {
*        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
*        PrintWriter pw = new PrintWriter(new OutputStreamWriter(System.out));
*        int T = Integer.parseInt(br.readLine());
*
*        while (T-- > 0) {
*            StringTokenizer st = new StringTokenizer(br.readLine());
*            int N = Integer.parseInt(st.nextToken());
*            int Q = Integer.parseInt(st.nextToken());
*
*            FenwickTree2D ft = new FenwickTree2D(N);
*
*            while (Q-- > 0) {
*                st = new StringTokenizer(br.readLine());
*                char op = st.nextToken().charAt(0);
*
*                if (op == 'C') {
*                    int x1 = Integer.parseInt(st.nextToken());
*                    int y1 = Integer.parseInt(st.nextToken());
*                    int x2 = Integer.parseInt(st.nextToken());
*                    int y2 = Integer.parseInt(st.nextToken());
*                    ft.update(x1, y1, x2, y2);
*                } else {
*                    int x = Integer.parseInt(st.nextToken());
*                    int y = Integer.parseInt(st.nextToken());
*                    pw.println(ft.query(x, y));
*                }
*            }
*
*            if (T > 0) {
*                pw.println();
*            }
*        }
*
```

```
 *         pw.flush();
 *         br.close();
 *         pw.close();
 *     }
 * }
 */

/*
 * 以下是Python实现的POJ2155代码
 *
 * import sys
 *
 * class FenwickTree2D:
 *     def __init__(self, n):
 *         self.n = n
 *         # 索引从1开始，预留额外空间
 *         self.tree = [[0] * (n + 2) for _ in range(n + 2)]
 *
 *     def _lowbit(self, x):
 *         return x & -x
 *
 *     def add(self, x, y, val):
 *         i = x
 *         while i <= self.n:
 *             j = y
 *             while j <= self.n:
 *                 self.tree[i][j] += val
 *                 j += self._lowbit(j)
 *             i += self._lowbit(i)
 *
 *     def sum(self, x, y):
 *         ans = 0
 *         i = x
 *         while i > 0:
 *             j = y
 *             while j > 0:
 *                 ans += self.tree[i][j]
 *                 j -= self._lowbit(j)
 *             i -= self._lowbit(i)
 *         return ans
 *
 *     def update(self, x1, y1, x2, y2):
 *         self.add(x1, y1, 1)
```

```
*         self.add(x1, y2 + 1, -1)
*         self.add(x2 + 1, y1, -1)
*         self.add(x2 + 1, y2 + 1, 1)
*
*     def query(self, x, y):
*         return self.sum(x, y) % 2
*
* def main():
*     input = sys.stdin.read().split()
*     ptr = 0
*     T = int(input[ptr])
*     ptr += 1
*
*     for _ in range(T):
*         N = int(input[ptr])
*         ptr += 1
*         Q = int(input[ptr])
*         ptr += 1
*
*         ft = FenwickTree2D(N)
*
*         for __ in range(Q):
*             op = input[ptr]
*             ptr += 1
*
*             if op == 'C':
*                 x1 = int(input[ptr])
*                 ptr += 1
*                 y1 = int(input[ptr])
*                 ptr += 1
*                 x2 = int(input[ptr])
*                 ptr += 1
*                 y2 = int(input[ptr])
*                 ptr += 1
*                 ft.update(x1, y1, x2, y2)
*             else:
*                 x = int(input[ptr])
*                 ptr += 1
*                 y = int(input[ptr])
*                 ptr += 1
*                 print(ft.query(x, y))
*
*         if _ < T - 1:
```

```
 *             print()
 *
 * if __name__ == '__main__':
 *     main()
 */

/*
 * 二维树状数组在区间更新单点查询场景中的应用与扩展：
 *
 * 1. 类似问题扩展：
 *    - POJ 3468 A Simple Problem with Integers（一维情况）
 *    - 二维区间异或、区间加法等操作都可以用类似方法处理
 *    - 图像处理中的区域操作，如图像反转、亮度调整等
 *
 * 2. 二进制性质的应用：
 *    - 本题中利用模 2 运算来实现取反效果
 *    - 类似地，可以利用不同模数实现其他周期性操作
 *    - 二进制位操作在树状数组中起到核心作用
 *
 * 3. 实现细节与优化：
 *    - 数组大小：通常需要比最大索引大 1 或 2，避免边界检查
 *    - 输入输出优化：对于大规模数据，使用快速 IO 方法
 *    - 内存优化：对于稀疏矩阵，可以考虑使用哈希表存储
 *    - 性能考虑：在 C++中使用数组比 vector 更快，在 Java 中可以用 BufferedReader 加速
 *
 * 4. 教学价值：
 *    - 是理解二维差分数组与树状数组结合的绝佳示例
 *    - 展示了如何将复杂的区间操作转换为简单的点操作
 *    - 体现了数学抽象在算法设计中的重要性
 *
 * 5. 实际应用场景：
 *    - 二维网格统计（如地图标记、热力图更新）
 *    - 游戏开发中的区域效果（如范围伤害、增益效果）
 *    - 数据库中的二维范围更新操作
 *    - 计算机图形学中的区域着色和变换
 */
```

==========

文件: POJ2155_Matrix.java

==========

```
package class108;
```

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

/**
 * POJ 2155 Matrix
 * 题目链接: http://poj.org/problem?id=2155
 *
 * 题目描述:
 * 给定一个 N×N 的矩阵，初始时所有元素都为 0。
 * 有两种操作：
 * 1. "C x1 y1 x2 y2"：将左上角为 (x1,y1)、右下角为 (x2,y2) 的子矩阵中的每个元素取反（0 变 1，1 变 0）
 * 2. "Q x y"：查询位置 (x,y) 的值
 *
 * 解题思路:
 * 使用二维树状数组 + 差分思想来解决这个问题。
 * 对于区间更新、单点查询的问题，可以使用二维差分数组配合二维树状数组：
 * 1. 对于更新操作，我们只需要在差分数组的四个角上进行更新：
 *    - 在 (x1, y1) 处 +1
 *    - 在 (x1, y2+1) 处 -1
 *    - 在 (x2+1, y1) 处 -1
 *    - 在 (x2+1, y2+1) 处 +1
 * 2. 对于查询操作，查询 (x,y) 点的值就是差分数组 (1,1) 到 (x,y) 的二维前缀和对 2 取模
 *
 * 时间复杂度：
 * - 区间更新: O(log n * log n)
 * - 单点查询: O(log n * log n)
 * 空间复杂度: O(n * n)
 */

public class POJ2155_Matrix {
    // 二维树状数组
    private int[][] tree;
    private int n;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    private int lowbit(int i) {
```

```
    return i & -i;
}

/**
 * 二维树状数组初始化
 *
 * @param n 矩阵大小
 */
public POJ2155_Matrix(int n) {
    this.n = n;
    this.tree = new int[n + 1][n + 1];
}

/**
 * 二维树状数组单点增加操作
 *
 * @param x x 坐标（从 1 开始）
 * @param y y 坐标（从 1 开始）
 * @param v 增加的值
 */
private void add(int x, int y, int v) {
    for (int i = x; i <= n; i += lowbit(i)) {
        for (int j = y; j <= n; j += lowbit(j)) {
            tree[i][j] += v;
        }
    }
}

/**
 * 二维树状数组前缀和查询：计算从(1,1)到(x,y)的矩形区域内所有元素的和
 *
 * @param x x 坐标（从 1 开始）
 * @param y y 坐标（从 1 开始）
 * @return 前缀和
 */
private int sum(int x, int y) {
    int ans = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            ans += tree[i][j];
        }
    }
    return ans;
```

```
}

/**
 * 区间更新操作：将左上角为(x1,y1)、右下角为(x2,y2)的子矩阵中的每个元素取反
 *
 * @param x1 左上角 x 坐标
 * @param y1 左上角 y 坐标
 * @param x2 右下角 x 坐标
 * @param y2 右下角 y 坐标
 */
public void update(int x1, int y1, int x2, int y2) {
    add(x1, y1, 1);
    add(x1, y2 + 1, -1);
    add(x2 + 1, y1, -1);
    add(x2 + 1, y2 + 1, 1);
}

/**
 * 单点查询操作：查询位置(x,y)的值
 *
 * @param x x 坐标
 * @param y y 坐标
 * @return 位置(x,y)的值
 */
public int query(int x, int y) {
    return sum(x, y) % 2;
}

/**
 * 主函数：处理输入输出和调用相关操作
 * 注意：POJ 的输入输出格式比较特殊，需要严格按照题目要求
 */
public static void main(String[] args) throws IOException {
    // 使用高效的 IO 处理方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取测试用例数量
    in.nextToken();
    int testCases = (int) in.nval;

    for (int t = 0; t < testCases; t++) {
```

```
            if (t > 0) out.println(); // 每个测试用例之间输出一个空行

            // 读取矩阵大小和操作数量
            in.nextToken();
            int n = (int) in.nval;
            in.nextToken();
            int operations = (int) in.nval;

            // 初始化二维树状数组
            POJ2155_Matrix matrix = new POJ2155_Matrix(n);

            // 处理操作
            for (int i = 0; i < operations; i++) {
                in.nextToken();
                String op = in.sval;

                if (op.equals("C")) {
                    // 区间更新操作
                    in.nextToken(); int x1 = (int) in.nval;
                    in.nextToken(); int y1 = (int) in.nval;
                    in.nextToken(); int x2 = (int) in.nval;
                    in.nextToken(); int y2 = (int) in.nval;
                    matrix.update(x1, y1, x2, y2);
                } else {
                    // 单点查询操作
                    in.nextToken(); int x = (int) in.nval;
                    in.nextToken(); int y = (int) in.nval;
                    out.println(matrix.query(x, y));
                }
            }
        }

        // 刷新输出缓冲区并关闭 IO 流
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: POJ2155_Matrix.py

==========

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

"""
POJ 2155 Matrix
题目链接: http://poj.org/problem?id=2155

题目描述:
给定一个 N×N 的矩阵，初始时所有元素都为 0。
有两种操作：
1. "C x1 y1 x2 y2"：将左上角为 (x1,y1)、右下角为 (x2,y2) 的子矩阵中的每个元素取反（0 变 1，1 变 0）
2. "Q x y"：查询位置 (x,y) 的值

解题思路:
使用二维树状数组 + 差分思想来解决这个问题。
对于区间更新、单点查询的问题，可以使用二维差分数组配合二维树状数组：
1. 对于更新操作，我们只需要在差分数组的四个角上进行更新：
   - 在 (x1, y1) 处 +1
   - 在 (x1, y2+1) 处 -1
   - 在 (x2+1, y1) 处 -1
   - 在 (x2+1, y2+1) 处 +1
2. 对于查询操作，查询 (x,y) 点的值就是差分数组 (1,1) 到 (x,y) 的二维前缀和对 2 取模

时间复杂度：
- 区间更新: O(log n * log n)
- 单点查询: O(log n * log n)
空间复杂度: O(n * n)
"""



class POJ2155_Matrix:
    def __init__(self, n):
        """
        二维树状数组初始化

        :param n: 矩阵大小
        """
        self.n = n
        self.tree = [[0 for _ in range(n + 1)] for _ in range(n + 1)]

    def lowbit(self, i):
        """
        lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
```

```
    例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

    :param i: 输入数字
    :return: 最低位的 1 所代表的数值
    """
    return i & -i

def add(self, x, y, v):
    """
    二维树状数组单点增加操作

    :param x: x 坐标（从 1 开始）
    :param y: y 坐标（从 1 开始）
    :param v: 增加的值
    """
    i = x
    while i <= self.n:
        j = y
        while j <= self.n:
            self.tree[i][j] += v
            j += self.lowbit(j)
        i += self.lowbit(i)

def sum(self, x, y):
    """
    二维树状数组前缀和查询：计算从(1,1)到(x,y)的矩形区域内所有元素的和

    :param x: x 坐标（从 1 开始）
    :param y: y 坐标（从 1 开始）
    :return: 前缀和
    """
    ans = 0
    i = x
    while i > 0:
        j = y
        while j > 0:
            ans += self.tree[i][j]
            j -= self.lowbit(j)
        i -= self.lowbit(i)
    return ans

def update(self, x1, y1, x2, y2):
    """
```

```
        区间更新操作：将左上角为(x1,y1)、右下角为(x2,y2)的子矩阵中的每个元素取反

        :param x1: 左上角 x 坐标
        :param y1: 左上角 y 坐标
        :param x2: 右下角 x 坐标
        :param y2: 右下角 y 坐标
        """
        self.add(x1, y1, 1)
        self.add(x1, y2 + 1, -1)
        self.add(x2 + 1, y1, -1)
        self.add(x2 + 1, y2 + 1, 1)

    def query(self, x, y):
        """
        单点查询操作：查询位置(x,y)的值

        :param x: x 坐标
        :param y: y 坐标
        :return: 位置(x,y)的值
        """
        return self.sum(x, y) % 2


def main():
    """
    主函数：处理输入输出和调用相关操作
    注意：POJ 的输入输出格式比较特殊，需要严格按照题目要求
    """
    # 读取测试用例数量
    test_cases = int(input())

    for t in range(test_cases):
        if t > 0:
            print()  # 每个测试用例之间输出一个空行

        # 读取矩阵大小和操作数量
        n, operations = map(int, input().split())

        # 初始化二维树状数组
        matrix = POJ2155_Matrix(n)

        # 处理操作
        for _ in range(operations):
```

```
            op, *args = input().split()

            if op == "C":
                # 区间更新操作
                x1, y1, x2, y2 = map(int, args)
                matrix.update(x1, y1, x2, y2)
            else:
                # 单点查询操作
                x, y = map(int, args)
                print(matrix.query(x, y))


# 由于POJ的在线评测系统可能不支持Python，此处省略实际运行代码
# 如果需要测试，可以取消下面的注释
# if __name__ == "__main__":
#     main()
```

==========

文件: POJ3468_SimpleProblemWithIntegers.cpp

==========

```
/*
 * POJ 3468 A Simple Problem with Integers
 * 题目链接: http://poj.org/problem?id=3468
 *
 * 题目描述:
 * 给定一个长度为 N 的数列 A，需要处理如下两种操作：
 * 1. "C a b c"：将区间 [a, b] 中的每个数都加上 c
 * 2. "Q a b"：求区间 [a, b] 中所有数的和
 *
 * 解题思路:
 * 使用树状数组实现区间更新、区间查询。
 * 这是树状数组的一个高级应用，需要使用差分的思想来处理区间更新。
 *
 * 设原数组为 A，差分数组为 D，其中 D[1] = A[1]，D[i] = A[i] - A[i-1] (i > 1)。
 *
 * 区间更新 [l, r] 增加 v：
 * - D[l] += v
 * - D[r+1] -= v
 *
 * 区间查询 [1, n] 的和：
 * - 设 sumD[i] = D[1] + D[2] + ... + D[i]
 * - 原数组前缀和 sumA[n] = (n+1) * sumD[n] - (D[1]*1 + D[2]*2 + ... + D[n]*n)
```

```
 *
 * 因此我们需要维护两个树状数组：
 * 1. tree1: 维护差分数组 D
 * 2. tree2: 维护 i*D[i]
 *
 * 时间复杂度：
 * - 区间更新: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(n)
 */

class POJ3468_SimpleProblemWithIntegers {
private:
    long long tree1[100001];  // 维护差分数组 D
    long long tree2[100001];  // 维护 i*D[i]
    int n;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    int lowbit(int i) {
        return i & -i;
    }

public:
    /**
     * 树状数组初始化
     *
     * @param n 数组长度
     */
    POJ3468_SimpleProblemWithIntegers(int n) {
        this->n = n;
        // 初始化数组
        for (int i = 0; i <= n; i++) {
            tree1[i] = 0;
            tree2[i] = 0;
        }
    }
```

```
/**
 * 在 tree1 和 tree2 中的 position 位置增加 value
 *
 * @param position 位置（从 1 开始）
 * @param value 增加的值
 */
void add(int position, long long value) {
    // 更新 tree1
    for (int i = position; i <= n; i += lowbit(i)) {
        tree1[i] += value;
    }

    // 更新 tree2
    for (int i = position; i <= n; i += lowbit(i)) {
        tree2[i] += (long long)position * value;
    }
}

/**
 * 查询 tree1 的前缀和 [1, position]
 *
 * @param position 查询位置
 * @return 前缀和
 */
long long sum1(int position) {
    long long ans = 0;
    for (int i = position; i > 0; i -= lowbit(i)) {
        ans += tree1[i];
    }
    return ans;
}

/**
 * 查询 tree2 的前缀和 [1, position]
 *
 * @param position 查询位置
 * @return 前缀和
 */
long long sum2(int position) {
    long long ans = 0;
    for (int i = position; i > 0; i -= lowbit(i)) {
        ans += tree2[i];
    }
```

```
        return ans;
    }

    /**
     * 区间更新：将区间 [l, r] 中的每个数都加上 value
     *
     * @param l 区间左端点
     * @param r 区间右端点
     * @param value 增加的值
     */
    void update(int l, int r, long long value) {
        add(l, value);
        add(r + 1, -value);
    }

    /**
     * 区间查询：求区间 [1, position] 的前缀和
     *
     * @param position 查询位置
     * @return 前缀和
     */
    long long prefixSum(int position) {
        return (long long)(position + 1) * sum1(position) - sum2(position);
    }

    /**
     * 区间查询：求区间 [l, r] 的和
     *
     * @param l 区间左端点
     * @param r 区间右端点
     * @return 区间和
     */
    long long rangeSum(int l, int r) {
        if (l == 1) {
            return prefixSum(r);
        }
        return prefixSum(r) - prefixSum(l - 1);
    }
};

/*
 * 主函数：处理输入输出和调用相关操作
 * 由于编译环境限制，此处省略主函数和测试代码
```

```
 *
 * 示例调用方式：
 * POJ3468_SimpleProblemWithIntegers solution(10);  // 创建大小为 10 的数组
 * solution.update(1, 3, 2);  // 将区间[1,3]都加上 2
 * long long result = solution.rangeSum(1, 3);  // 查询区间[1,3]的和
 */

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

文件: POJ3468_SimpleProblemWithIntegers.java
==========================================
package class108;

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

/**
 * POJ 3468 A Simple Problem with Integers
 * 题目链接: http://poj.org/problem?id=3468
 *
 * 题目描述:
 * 给定一个长度为 N 的数列 A，需要处理如下两种操作：
 * 1. "C a b c"：将区间 [a, b] 中的每个数都加上 c
 * 2. "Q a b"：求区间 [a, b] 中所有数的和
 *
 * 解题思路:
 * 使用树状数组实现区间更新、区间查询。
 * 这是树状数组的一个高级应用，需要使用差分的思想来处理区间更新。
 *
 * 设原数组为 A，差分数组为 D，其中 D[1] = A[1]，D[i] = A[i] - A[i-1] (i > 1)。
 *
 * 区间更新 [l, r] 增加 v：
 * - D[l] += v
 * - D[r+1] -= v
 *
 * 区间查询 [1, n] 的和：
 * - 设 sumD[i] = D[1] + D[2] + ... + D[i]
 * - 原数组前缀和 sumA[n] = (n+1) * sumD[n] - (D[1]*1 + D[2]*2 + ... + D[n]*n)
 *
 * 因此我们需要维护两个树状数组：
 * 1. tree1: 维护差分数组 D
 * 2. tree2: 维护 i*D[i]
 *
```

```
 * 时间复杂度：
 * - 区间更新: O(log n)
 * - 区间查询: O(log n)
 * 空间复杂度: O(n)
 */

public class POJ3468_SimpleProblemWithIntegers {
    private long[] tree1;  // 维护差分数组 D
    private long[] tree2;  // 维护 i*D[i]
    private int n;

    /**
     * lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
     * 例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)
     *
     * @param i 输入数字
     * @return 最低位的 1 所代表的数值
     */
    private int lowbit(int i) {
        return i & -i;
    }

    /**
     * 树状数组初始化
     *
     * @param n 数组长度
     */
    public POJ3468_SimpleProblemWithIntegers(int n) {
        this.n = n;
        this.tree1 = new long[n + 1];
        this.tree2 = new long[n + 1];
    }

    /**
     * 在 tree1 和 tree2 中的 position 位置增加 value
     *
     * @param position 位置（从 1 开始）
     * @param value 增加的值
     */
    private void add(int position, long value) {
        // 更新 tree1
        for (int i = position; i <= n; i += lowbit(i)) {
            tree1[i] += value;
```

```
        }

        // 更新 tree2
        for (int i = position; i <= n; i += lowbit(i)) {
            tree2[i] += position * value;
        }
    }

    /**
     * 查询 tree1 的前缀和 [1, position]
     *
     * @param position 查询位置
     * @return 前缀和
     */
    private long sum1(int position) {
        long ans = 0;
        for (int i = position; i > 0; i -= lowbit(i)) {
            ans += tree1[i];
        }
        return ans;
    }

    /**
     * 查询 tree2 的前缀和 [1, position]
     *
     * @param position 查询位置
     * @return 前缀和
     */
    private long sum2(int position) {
        long ans = 0;
        for (int i = position; i > 0; i -= lowbit(i)) {
            ans += tree2[i];
        }
        return ans;
    }

    /**
     * 区间更新：将区间 [l, r] 中的每个数都加上 value
     *
     * @param l 区间左端点
     * @param r 区间右端点
     * @param value 增加的值
     */
```

```
public void update(int l, int r, long value) {
    add(l, value);
    add(r + 1, -value);
}

/**
 * 区间查询：求区间 [1, position] 的前缀和
 *
 * @param position 查询位置
 * @return 前缀和
 */
public long prefixSum(int position) {
    return (position + 1) * sum1(position) - sum2(position);
}

/**
 * 区间查询：求区间 [l, r] 的和
 *
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 区间和
 */
public long rangeSum(int l, int r) {
    if (l == 1) {
        return prefixSum(r);
    }
    return prefixSum(r) - prefixSum(l - 1);
}

/**
 * 主函数：处理输入输出和调用相关操作
 */
public static void main(String[] args) throws IOException {
    // 使用高效的 IO 处理方式
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer in = new StreamTokenizer(br);
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));

    // 读取数组长度和操作数量
    in.nextToken();
    int n = (int) in.nval;
    in.nextToken();
    int q = (int) in.nval;
```

```
        // 初始化树状数组
        POJ3468_SimpleProblemWithIntegers solution = new POJ3468_SimpleProblemWithIntegers(n);

        // 读取初始数组并构建差分数组
        long[] a = new long[n + 1];
        for (int i = 1; i <= n; i++) {
            in.nextToken();
            a[i] = (long) in.nval;
        }

        // 通过单点更新的方式构建初始差分数组
        for (int i = 1; i <= n; i++) {
            solution.update(i, i, a[i] - a[i - 1]);
        }

        // 处理操作
        for (int i = 0; i < q; i++) {
            in.nextToken();
            String op = in.sval;

            if (op.equals("C")) {
                // 区间更新操作
                in.nextToken(); int l = (int) in.nval;
                in.nextToken(); int r = (int) in.nval;
                in.nextToken(); long c = (long) in.nval;
                solution.update(l, r, c);
            } else {
                // 区间查询操作
                in.nextToken(); int l = (int) in.nval;
                in.nextToken(); int r = (int) in.nval;
                out.println(solution.rangeSum(l, r));
            }
        }

        // 刷新输出缓冲区并关闭 IO 流
        out.flush();
        out.close();
        br.close();
    }
}
```

==========

文件: POJ3468_SimpleProblemWithIntegers.py
==========

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

"""
POJ 3468 A Simple Problem with Integers
题目链接: http://poj.org/problem?id=3468

题目描述:
给定一个长度为 N 的数列 A，需要处理如下两种操作：
1. "C a b c"：将区间 [a, b] 中的每个数都加上 c
2. "Q a b"：求区间 [a, b] 中所有数的和

解题思路:
使用树状数组实现区间更新、区间查询。
这是树状数组的一个高级应用，需要使用差分的思想来处理区间更新。

设原数组为 A，差分数组为 D，其中 D[1] = A[1]，D[i] = A[i] - A[i-1] (i > 1)。

区间更新 [l, r] 增加 v：
- D[l] += v
- D[r+1] -= v

区间查询 [1, n] 的和：
- 设 sumD[i] = D[1] + D[2] + ... + D[i]
- 原数组前缀和 sumA[n] = (n+1) * sumD[n] - (D[1]*1 + D[2]*2 + ... + D[n]*n)

因此我们需要维护两个树状数组：
1. tree1: 维护差分数组 D
2. tree2: 维护 i*D[i]

时间复杂度：
- 区间更新: O(log n)
- 区间查询: O(log n)
空间复杂度: O(n)
"""



class POJ3468_SimpleProblemWithIntegers:
    def __init__(self, n):
        """
```

```
    树状数组初始化

    :param n: 数组长度
    """
    self.n = n
    self.tree1 = [0] * (n + 1)  # 维护差分数组 D
    self.tree2 = [0] * (n + 1)  # 维护 i*D[i]

def lowbit(self, i):
    """
    lowbit 函数：获取数字的二进制表示中最右边的 1 所代表的数值
    例如：x=6(110) 返回 2(010)，x=12(1100) 返回 4(0100)

    :param i: 输入数字
    :return: 最低位的 1 所代表的数值
    """
    return i & -i

def add(self, position, value):
    """
    在 tree1 和 tree2 中的 position 位置增加 value

    :param position: 位置（从 1 开始）
    :param value: 增加的值
    """
    # 更新 tree1
    i = position
    while i <= self.n:
        self.tree1[i] += value
        i += self.lowbit(i)

    # 更新 tree2
    i = position
    while i <= self.n:
        self.tree2[i] += position * value
        i += self.lowbit(i)

def sum1(self, position):
    """
    查询 tree1 的前缀和 [1, position]

    :param position: 查询位置
    :return: 前缀和
```

```
        """
        ans = 0
        i = position
        while i > 0:
            ans += self.tree1[i]
            i -= self.lowbit(i)
        return ans

    def sum2(self, position):
        """
        查询 tree2 的前缀和 [1, position]

        :param position: 查询位置
        :return: 前缀和
        """
        ans = 0
        i = position
        while i > 0:
            ans += self.tree2[i]
            i -= self.lowbit(i)
        return ans

    def update(self, l, r, value):
        """
        区间更新：将区间 [l, r] 中的每个数都加上 value

        :param l: 区间左端点
        :param r: 区间右端点
        :param value: 增加的值
        """
        self.add(l, value)
        self.add(r + 1, -value)

    def prefixSum(self, position):
        """
        区间查询：求区间 [1, position] 的前缀和

        :param position: 查询位置
        :return: 前缀和
        """
        return (position + 1) * self.sum1(position) - self.sum2(position)

    def rangeSum(self, l, r):
```

```
        """
        区间查询：求区间 [l, r] 的和

        :param l: 区间左端点
        :param r: 区间右端点
        :return: 区间和
        """
        if l == 1:
            return self.prefixSum(r)
        return self.prefixSum(r) - self.prefixSum(l - 1)


def main():
    """
    主函数：处理输入输出和调用相关操作
    由于 POJ 的在线评测系统可能不支持 Python，此处省略实际运行代码
    """
    # 示例用法
    # solution = POJ3468_SimpleProblemWithIntegers(10)  # 创建大小为 10 的数组
    # solution.update(1, 3, 2)  # 将区间[1,3]都加上 2
    # result = solution.rangeSum(1, 3)  # 查询区间[1,3]的和
    pass


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

==========

文件: TwoDimensionFenwickTree.java

==========

```
package class108;

/**
 * 二维树状数组区间增加、区间查询实现
 *
 * 本文件包含了二维树状数组区间更新和区间查询操作的详细实现
 * 支持二维平面上的高效矩形区域更新和查询操作
 *
 * 测试链接: https://www.luogu.com.cn/problem/P4514
 *
 * 核心思想：
 * 使用四个树状数组维护差分数组的不同组合项，通过数学推导得出的公式支持区间操作
```

```
 *
 * 时间复杂度分析：
 * - 区间更新: O(log n * log m)
 * - 区间查询: O(log n * log m)
 * 空间复杂度: O(n * m)
 */

/**
 * 二维树状数组区间更新区间查询的数学原理深度推导：
 *
 * 1. 二维差分数组的定义：
 *    设原二维数组为 a[i][j]，差分数组为 d[i][j]，则满足：
 *    a[i][j] = sum_{x=1 到 i} sum_{y=1 到 j} d[x][y]
 *
 * 2. 二维前缀和的计算：
 *    sum_{x=1 到 i} sum_{y=1 到 j} a[x][y] = sum_{x=1 到 i} sum_{y=1 到 j} sum_{p=1 到 x} sum_{q=1 到 y} d[p][q]
 *    通过交换求和顺序并展开，可以得到：
 *    sum_{p=1 到 i} sum_{q=1 到 j} d[p][q] * (i-p+1) * (j-q+1)
 *    = sum_{p=1 到 i} sum_{q=1 到 j} d[p][q] * (i+1)(j+1) - d[p][q] * (i+1)q - d[p][q] * p(j+1) + d[p][q] * pq
 *    = (i+1)(j+1)sum1 - (i+1)sum2 - (j+1)sum3 + sum4
 *    其中：
 *    sum1 = sum_{p=1 到 i} sum_{q=1 到 j} d[p][q]
 *    sum2 = sum_{p=1 到 i} sum_{q=1 到 j} d[p][q] * q
 *    sum3 = sum_{p=1 到 i} sum_{q=1 到 j} d[p][q] * p
 *    sum4 = sum_{p=1 到 i} sum_{q=1 到 j} d[p][q] * p * q
 *
 * 3. 区间更新的转换：
 *    当对矩形区域[a,b]到[c,d]加上 v 时，差分数组的变化为：
 *    d[a][b] += v
 *    d[a][d+1] -= v
 *    d[c+1][b] -= v
 *    d[c+1][d+1] += v
 *    这是二维差分的标准做法
 *
 * 4. 四个树状数组的维护：
 *    info1[i][j] 维护 d[i][j]
 *    info2[i][j] 维护 d[i][j] * i
 *    info3[i][j] 维护 d[i][j] * j
 *    info4[i][j] 维护 d[i][j] * i * j
 *    这样可以高效计算上述四个 sum 值
 */
```

```
/* C++代码实现 */
/* 取消注释以下代码可以直接在C++环境中编译运行 */

//#include <cstdio>
//using namespace std;
//
///**
// * 最大数据范围，根据题目要求设置
// */
//const int MAXN = 2050;
//const int MAXM = 2050;
//
///**
// * 维护四个二维树状数组
// * info1[i][j]: 维护差分数组 d[i][j]
// * info2[i][j]: 维护 d[i][j] * i
// * info3[i][j]: 维护 d[i][j] * j
// * info4[i][j]: 维护 d[i][j] * i * j
// */
//int info1[MAXN][MAXM], info2[MAXN][MAXM], info3[MAXN][MAXM], info4[MAXN][MAXM];
//int n, m;
//
///**
// * lowbit 函数：获取数字 i 的二进制表示中最低位的 1 所对应的值
// *
// * @param i 输入的整数
// * @return i 的二进制表示中最低位的 1 所对应的值
// */
//int lowbit(int i) {
//    return i & -i;  // 利用位运算获取最低位的 1
//}
//
///**
// * 在点(x,y)处更新差分数组，并同时维护四个树状数组
// *
// * @param x 行坐标（从 1 开始）
// * @param y 列坐标（从 1 开始）
// * @param v 要增加的值
// */
//void add(int x, int y, int v) {
//    // 计算四个需要更新的树状数组对应的值
//    int v1 = v;            // 对应 info1: d[i][j]
```

```
//    int v2 = x * v;        // 对应 info2: d[i][j] * i
//    int v3 = y * v;        // 对应 info3: d[i][j] * j
//    int v4 = x * y * v;    // 对应 info4: d[i][j] * i * j
//
//    // 更新四个树状数组
//    for (int i = x; i <= n; i += lowbit(i)) {
//        for (int j = y; j <= m; j += lowbit(j)) {
//            info1[i][j] += v1;
//            info2[i][j] += v2;
//            info3[i][j] += v3;
//            info4[i][j] += v4;
//        }
//    }
//}
//
///**
// * 计算二维前缀和(1,1)~(x,y)
// * 使用数学推导得出的公式，结合四个树状数组计算二维前缀和
// *
// * @param x 行坐标（从 1 开始）
// * @param y 列坐标（从 1 开始）
// * @return (1,1)~(x,y)矩形区域的和
// */
//int sum(int x, int y) {
//    int ans = 0;
//    // 遍历树状数组计算前缀和
//    for (int i = x; i > 0; i -= lowbit(i)) {
//        for (int j = y; j > 0; j -= lowbit(j)) {
//            // 数学公式计算，由二维前缀和展开推导得出
//            ans += (x + 1) * (y + 1) * info1[i][j] - (y + 1) * info2[i][j] - (x + 1) *
info3[i][j] + info4[i][j];
//        }
//    }
//    return ans;
//}
//
///**
// * 给矩形区域(a,b)~(c,d)的所有元素加 v
// * 利用二维差分数组的特性，将矩形区域更新转换为四个角落点的更新
// *
// * @param a 左上区域行坐标（从 1 开始）
// * @param b 左上区域列坐标（从 1 开始）
// * @param c 右下区域行坐标（从 1 开始）
```

```
// * @param d 右下区域列坐标（从 1 开始）
// * @param v 要增加的值
// */
//void add(int a, int b, int c, int d, int v) {
//    // 利用二维差分数组的特性，对四个角点进行更新
//    add(a, b, v);          // (a,b)处加 v
//    add(c + 1, d + 1, v); // (c+1,d+1)处加 v
//    add(a, d + 1, -v);    // (a,d+1)处减 v
//    add(c + 1, b, -v);    // (c+1,b)处减 v
//}
//
///**
// * 查询区域和(a,b)~(c,d)
// * 利用二维前缀和的容斥原理，通过四个前缀和的组合计算出目标区域的和
// *
// * @param a 左上区域行坐标（从 1 开始）
// * @param b 左上区域列坐标（从 1 开始）
// * @param c 右下区域行坐标（从 1 开始）
// * @param d 右下区域列坐标（从 1 开始）
// * @return (a,b)~(c,d)矩形区域的和
// */
//int range(int a, int b, int c, int d) {
//    // 容斥原理：全量减去两边加上重叠部分
//    return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
//}
//
///**
// * 主函数，处理输入输出和操作请求
// */
//int main() {
//    char op;  // 操作类型
//    int a, b, c, d, v;  // 坐标和值
//
//    // 读取初始操作
//    scanf("%s", &op);
//    scanf("%d%d", &n, &m);  // 读取二维数组大小
//
//    // 处理操作直到文件结束
//    while (scanf("%s", &op) != EOF) {
//        if (op == 'X') {  // X 命令：更新数组大小
//            scanf("%d%d", &n, &m);
//        } else if (op == 'L') {  // L 命令：区间更新操作
//            scanf("%d%d%d%d%d", &a, &b, &c, &d, &v);
```

```
//            add(a, b, c, d, v);  // 执行区间更新
//        } else {  // 查询命令：区间查询操作
//            scanf("%d%d%d%d", &a, &b, &c, &d);
//            printf("%d\n", range(a, b, c, d));  // 输出查询结果
//        }
//    }
//
//    return 0;
//}

/**
 * 以下是 Java 实现的二维树状数组区间更新区间查询代码
 */
public class TwoDimensionFenwickTree {

    /**
     * 维护四个二维树状数组
     */
    private long[][] info1; // 维护 d[i][j]
    private long[][] info2; // 维护 d[i][j] * i
    private long[][] info3; // 维护 d[i][j] * j
    private long[][] info4; // 维护 d[i][j] * i * j
    private int n; // 行数
    private int m; // 列数

    /**
     * 构造函数
     * @param n 最大行数
     * @param m 最大列数
     */
    public TwoDimensionFenwickTree(int n, int m) {
        this.n = n;
        this.m = m;
        // 初始化四个树状数组，索引从 1 开始
        info1 = new long[n + 2][m + 2];
        info2 = new long[n + 2][m + 2];
        info3 = new long[n + 2][m + 2];
        info4 = new long[n + 2][m + 2];
    }

    /**
     * lowbit 函数
     * @param i 输入整数
```

```
     * @return 最低位1所对应的值
     */
    private int lowbit(int i) {
        return i & -i;
    }

    /**
     * 在点(x,y)处增加v，同时更新四个树状数组
     * @param x 行坐标（从1开始）
     * @param y 列坐标（从1开始）
     * @param v 要增加的值
     */
    private void add(int x, int y, long v) {
        long v1 = v;
        long v2 = v * x;
        long v3 = v * y;
        long v4 = v * x * y;

        // 更新四个树状数组
        for (int i = x; i <= n; i += lowbit(i)) {
            for (int j = y; j <= m; j += lowbit(j)) {
                info1[i][j] += v1;
                info2[i][j] += v2;
                info3[i][j] += v3;
                info4[i][j] += v4;
            }
        }
    }

    /**
     * 计算前缀和(1,1)~(x,y)
     * @param x 行坐标（从1开始）
     * @param y 列坐标（从1开始）
     * @return 前缀和结果
     */
    private long sum(int x, int y) {
        long ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) {
            for (int j = y; j > 0; j -= lowbit(j)) {
                // 应用数学公式
                ans += (x + 1) * (y + 1) * info1[i][j] - (y + 1) * info2[i][j] - (x + 1) *
info3[i][j] + info4[i][j];
            }
```

```
        }
        return ans;
    }

    /**
     * 对矩形区域(a,b)~(c,d)的所有元素加 v
     * @param a 左上区域行坐标（从 1 开始）
     * @param b 左上区域列坐标（从 1 开始）
     * @param c 右下区域行坐标（从 1 开始）
     * @param d 右下区域列坐标（从 1 开始）
     * @param v 要增加的值
     */
    public void rangeAdd(int a, int b, int c, int d, long v) {
        add(a, b, v);
        add(a, d + 1, -v);
        add(c + 1, b, -v);
        add(c + 1, d + 1, v);
    }

    /**
     * 查询矩形区域(a,b)~(c,d)的和
     * @param a 左上区域行坐标（从 1 开始）
     * @param b 左上区域列坐标（从 1 开始）
     * @param c 右下区域行坐标（从 1 开始）
     * @param d 右下区域列坐标（从 1 开始）
     * @return 区域和
     */
    public long rangeQuery(int a, int b, int c, int d) {
        return sum(c, d) - sum(a - 1, d) - sum(c, b - 1) + sum(a - 1, b - 1);
    }

    /**
     * 设置矩阵大小
     * @param n 新的行数
     * @param m 新的列数
     */
    public void setSize(int n, int m) {
        this.n = n;
        this.m = m;
        // 重置树状数组
        info1 = new long[n + 2][m + 2];
        info2 = new long[n + 2][m + 2];
        info3 = new long[n + 2][m + 2];
```

```
        info4 = new long[n + 2][m + 2];
    }

    /**
     * 主方法，用于处理输入输出
     */
    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        String op = sc.next();
        int n = sc.nextInt();
        int m = sc.nextInt();

        TwoDimensionFenwickTree ft = new TwoDimensionFenwickTree(n, m);

        while (sc.hasNext()) {
            op = sc.next();
            if (op.equals("X")) {
                n = sc.nextInt();
                m = sc.nextInt();
                ft.setSize(n, m);
            } else if (op.equals("L")) {
                int a = sc.nextInt();
                int b = sc.nextInt();
                int c = sc.nextInt();
                int d = sc.nextInt();
                long v = sc.nextLong();
                ft.rangeAdd(a, b, c, d, v);
            } else {
                int a = sc.nextInt();
                int b = sc.nextInt();
                int c = sc.nextInt();
                int d = sc.nextInt();
                System.out.println(ft.rangeQuery(a, b, c, d));
            }
        }
        sc.close();
    }
}

/**
 * 以下是Python实现的二维树状数组区间更新区间查询代码
 *
 * class TwoDimensionFenwickTree:
```

```
*     def __init__(self, n, m):
*         self.n = n
*         self.m = m
*         # 初始化四个树状数组，索引从 1 开始
*         self.info1 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护 d[i][j]
*         self.info2 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护 d[i][j] * i
*         self.info3 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护 d[i][j] * j
*         self.info4 = [[0] * (m + 2) for _ in range(n + 2)]  # 维护 d[i][j] * i * j
*
*     def _lowbit(self, x):
*         # lowbit 函数
*         return x & -x
*
*     def _add(self, x, y, v):
*         # 在点(x,y)处增加 v，同时更新四个树状数组
*         v1 = v
*         v2 = v * x
*         v3 = v * y
*         v4 = v * x * y
*
*         i = x
*         while i <= self.n:
*             j = y
*             while j <= self.m:
*                 self.info1[i][j] += v1
*                 self.info2[i][j] += v2
*                 self.info3[i][j] += v3
*                 self.info4[i][j] += v4
*                 j += self._lowbit(j)
*             i += self._lowbit(i)
*
*     def _sum(self, x, y):
*         # 计算前缀和(1,1)~(x,y)
*         ans = 0
*         i = x
*         while i > 0:
*             j = y
*             while j > 0:
*                 # 应用数学公式
*                 ans += (x + 1) * (y + 1) * self.info1[i][j] \
*                        - (y + 1) * self.info2[i][j] \
*                        - (x + 1) * self.info3[i][j] \
*                        + self.info4[i][j]
```

```
*                 j -= self._lowbit(j)
*             i -= self._lowbit(i)
*         return ans
*
*     def range_add(self, a, b, c, d, v):
*         # 对矩形区域(a,b)~(c,d)的所有元素加 v
*         self._add(a, b, v)
*         self._add(a, d + 1, -v)
*         self._add(c + 1, b, -v)
*         self._add(c + 1, d + 1, v)
*
*     def range_query(self, a, b, c, d):
*         # 查询矩形区域(a,b)~(c,d)的和
*         return (self._sum(c, d) -
*                 self._sum(a - 1, d) -
*                 self._sum(c, b - 1) +
*                 self._sum(a - 1, b - 1))
*
*     def set_size(self, n, m):
*         # 设置矩阵大小
*         self.n = n
*         self.m = m
*         # 重置树状数组
*         self.info1 = [[0] * (m + 2) for _ in range(n + 2)]
*         self.info2 = [[0] * (m + 2) for _ in range(n + 2)]
*         self.info3 = [[0] * (m + 2) for _ in range(n + 2)]
*         self.info4 = [[0] * (m + 2) for _ in range(n + 2)]
*
* # 主函数
* def main():
*     import sys
*     input = sys.stdin.read().split()
*     ptr = 0
*
*     op = input[ptr]
*     ptr += 1
*     n = int(input[ptr])
*     ptr += 1
*     m = int(input[ptr])
*     ptr += 1
*
*     ft = TwoDimensionFenwickTree(n, m)
*
```

```
 *     while ptr < len(input):
 *         op = input[ptr]
 *         ptr += 1
 *
 *         if op == 'X':
 *             n = int(input[ptr])
 *             ptr += 1
 *             m = int(input[ptr])
 *             ptr += 1
 *             ft.set_size(n, m)
 *         elif op == 'L':
 *             a = int(input[ptr])
 *             ptr += 1
 *             b = int(input[ptr])
 *             ptr += 1
 *             c = int(input[ptr])
 *             ptr += 1
 *             d = int(input[ptr])
 *             ptr += 1
 *             v = int(input[ptr])
 *             ptr += 1
 *             ft.range_add(a, b, c, d, v)
 *         else:
 *             a = int(input[ptr])
 *             ptr += 1
 *             b = int(input[ptr])
 *             ptr += 1
 *             c = int(input[ptr])
 *             ptr += 1
 *             d = int(input[ptr])
 *             ptr += 1
 *             print(ft.range_query(a, b, c, d))
 *
 * if __name__ == '__main__':
 *     main()
 */

/**
 * 二维树状数组区间更新区间查询的高级分析与工程实践：
 *
 * 1. 三维及以上扩展：
 *    - 对于三维情况，需要 8 个树状数组维护不同的乘积项
 *    - 每增加一个维度，需要维护的树状数组数量翻倍
```

```
*     - 高维情况下时间复杂度为O(log^k n)，其中k为维度
*     - 实际应用中二维已经是较为常用的情况，三维及以上使用较少
*
* 2. 数据类型溢出问题：
*     - 对于大数据量，必须使用long类型或long long类型
*     - 二维情况下数值会累积更快，尤其需要注意溢出
*     - 在Java中使用long，C++中使用long long，Python自动处理大数
*
* 3. 性能优化技巧：
*     - 缓存热点区域的查询结果
*     - 使用快速IO方法处理大规模输入
*     - 预分配足够空间，避免动态扩容
*     - 对于稀疏矩阵，考虑使用哈希表实现
*
* 4. 内存优化策略：
*     - 对于大规模数据，可以考虑使用离散化技术
*     - 对于特定问题，可以根据实际数据范围优化数组大小
*     - 避免在栈上分配大型数组，可能导致栈溢出
*
* 5. 实际应用场景：
*     - 图像处理中的区域滤镜和效果处理
*     - 二维统计数据的动态更新和查询
*     - 游戏开发中的区域影响计算
*     - 地理信息系统中的空间分析
*
* 6. 与线段树的对比：
*     - 二维树状数组实现更简单，常数更小
*     - 线段树可以支持更复杂的区间操作（如区间最大值）
*     - 树状数组在实际应用中通常更快，但功能相对受限
*
* 7. 调试技巧：
*     - 使用小例子测试各个操作的正确性
*     - 打印中间状态以验证树状数组的更新是否正确
*     - 对比一维情况，逐步扩展到二维
*     - 确保数学公式的正确性，这是最容易出错的地方
*/

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