

=====
文件夹: class087_AdvancedSegmentTreeApplications
=====

[Markdown 文件]
=====

文件: ADDITIONAL_PROBLEMS.md
=====

线段树更多题目列表

1. LeetCode 题目

LeetCode 218. The Skyline Problem (天际线问题)

- **题目链接**: <https://leetcode.cn/problems/the-skyline-problem/>
- **题目描述**: 给定建筑物的位置和高度, 返回天际线的关键点
- **算法**: 线段树 + 扫描线 或 multiset
- **难度**: 困难

LeetCode 699. Falling Squares (掉落的方块)

- **题目链接**: <https://leetcode.cn/problems/falling-squares/>
- **题目描述**: 掉落的方块堆叠问题, 求每次掉落后的最大高度
- **算法**: 线段树 + 懒惰标记
- **难度**: 困难

LeetCode 732. My Calendar III (我的日程安排表 III)

- **题目链接**: <https://leetcode.cn/problems/my-calendar-iii/>
- **题目描述**: 检测 K 重预订
- **算法**: 动态开点线段树
- **难度**: 困难

LeetCode 715. Range Module (范围模块)

- **题目链接**: <https://leetcode.cn/problems/range-module/>
- **题目描述**: 范围模块, 支持添加、查询和删除区间
- **算法**: 动态开点线段树
- **难度**: 困难

2. Codeforces 题目

Codeforces 52C Circular RMQ (循环区间最小值查询)

- **题目链接**: <https://codeforces.com/problemset/problem/52/C>
- **题目描述**: 循环区间最小值查询
- **算法**: 线段树 + 懒惰标记
- **难度**: 中等

Codeforces 438D The Child and Sequence (区间取模)

- **题目链接**： <https://codeforces.com/problemset/problem/438/D>
- **题目描述**： 区间取模，区间最大值，区间和
- **算法**： 吉司机线段树
- **难度**： 困难

Codeforces 446C DZY Loves Fibonacci Numbers (斐波那契数列)

- **题目链接**： <https://codeforces.com/problemset/problem/446/C>
- **题目描述**： 斐波那契数列区间加法
- **算法**： 线段树 + 斐波那契性质
- **难度**： 困难

Codeforces 380C Sereja and Brackets (括号匹配)

- **题目链接**： <https://codeforces.com/problemset/problem/380/C>
- **题目描述**： 括号匹配查询
- **算法**： 线段树
- **难度**： 中等

3. SPOJ 题目

SPOJ GSS2 Can you answer these queries II (历史最大子段和)

- **题目链接**： <https://www.spoj.com/problems/GSS2/>
- **题目描述**： 历史最大子段和查询
- **算法**： 线段树 + 历史信息维护
- **难度**： 困难

SPOJ GSS4 Can you answer these queries IV (区间开方)

- **题目链接**： <https://www.spoj.com/problems/GSS4/>
- **题目描述**： 区间开方，区间求和
- **算法**： 线段树 + 区间操作
- **难度**： 中等

SPOJ GSS6 Can you answer these queries VI (平衡树)

- **题目链接**： <https://www.spoj.com/problems/GSS6/>
- **题目描述**： 支持插入、删除、修改、查询最大子段和
- **算法**： 平衡树 + 线段树
- **难度**： 困难

SPOJ GSS7 Can you answer these queries VII (树链剖分)

- **题目链接**： <https://www.spoj.com/problems/GSS7/>
- **题目描述**： 树上路径最大子段和查询
- **算法**： 树链剖分 + 线段树

- ****难度****: 困难

SPOJ HORRIBLE Horrible Queries (区间加法)

- ****题目链接****: <https://www.spoj.com/problems/HORRIBLE/>
- ****题目描述****: 区间加法, 区间求和
- ****算法****: 线段树 + 懒惰标记
- ****难度****: 中等

4. POJ 题目

POJ 2528 Mayor's posters (海报问题)

- ****题目链接****: <http://poj.org/problem?id=2528>
- ****题目描述****: 区间染色问题, 求可见海报数
- ****算法****: 线段树 + 离散化
- ****难度****: 中等

POJ 3667 Hotel (酒店房间分配)

- ****题目链接****: <http://poj.org/problem?id=3667>
- ****题目描述****: 区间分配问题, 支持区间占用和释放
- ****算法****: 线段树 + 区间合并
- ****难度****: 困难

POJ 1177 Picture (矩形周长并)

- ****题目链接****: <http://poj.org/problem?id=1177>
- ****题目描述****: 矩形周长并
- ****算法****: 线段树 + 扫描线
- ****难度****: 困难

5. HDU 题目

HDU 1542 Atlantis (矩形面积并)

- ****题目链接****: <http://acm.hdu.edu.cn/showproblem.php?pid=1542>
- ****题目描述****: 矩形面积并
- ****算法****: 线段树 + 扫描线 + 离散化
- ****难度****: 困难

HDU 438D The Child and Sequence

- ****题目链接****: <http://acm.hdu.edu.cn/showproblem.php?pid=438D>
- ****题目描述****: 区间取模, 区间最大值, 区间和
- ****算法****: 吉司机线段树
- ****难度****: 困难

HDU 1698 Just a Hook (区间更新)

- **题目链接**: <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
- **题目描述**: 区间更新, 区间求和
- **算法**: 线段树 + 懒惰标记
- **难度**: 中等

6. 洛谷题目

P3373 【模板】线段树 2 (区间乘法和加法)

- **题目链接**: <https://www.luogu.com.cn/problem/P3373>
- **题目描述**: 区间乘法和加法, 区间求和
- **算法**: 线段树 + 双懒惰标记
- **难度**: 中等

P6242 【模板】线段树 3 (区间最值操作、区间历史最值)

- **题目链接**: <https://www.luogu.com.cn/problem/P6242>
- **题目描述**: 区间加法、区间取 min、区间求和、区间最值、区间历史最值
- **算法**: 吉司机线段树 + 历史信息维护
- **难度**: 困难

P5490 【模板】扫描线 (矩形面积并)

- **题目链接**: <https://www.luogu.com.cn/problem/P5490>
- **题目描述**: 矩形面积并
- **算法**: 线段树 + 扫描线 + 离散化
- **难度**: 困难

7. 其他平台题目

ZOJ 1610 Count the Colors (区间染色)

- **题目链接**: <https://vjudge.net/problem/ZOJ-1610>
- **题目描述**: 区间染色问题
- **算法**: 线段树 + 离散化
- **难度**: 中等

这些题目涵盖了线段树的各种应用场景, 包括:

1. 基础区间操作 (加法、乘法、更新)
2. 区间最值操作 (取 min、取 max)
3. 区间历史信息维护
4. 动态开点线段树
5. 离散化技术
6. 扫描线算法
7. 树链剖分
8. 平衡树结合
9. 复杂区间操作 (取模、开方等)

通过练习这些题目，可以全面掌握线段树的各种应用技巧。

=====

文件: ANALYSIS.md

=====

线段树高级应用深度分析

1. 动态开点线段树分析

1.1 基本概念

动态开点线段树是一种按需创建节点的线段树实现方式。与传统线段树预先分配所有节点不同，动态开点线段树只在访问到某个区间时才创建相应的子节点。

1.2 核心思想

- 1. **节点动态分配**: 只有在需要访问某个区间时才创建对应的节点
- 2. **空间优化**: 避免为未访问的区间分配内存
- 3. **懒惰标记**: 结合懒惰标记技术优化区间操作

1.3 实现要点

- 1. 使用指针或数组下标动态管理节点
- 2. 在访问子节点前检查是否存在，不存在则创建
- 3. 合理估算最大节点数以避免内存溢出

1.4 时间复杂度

- 单次操作: $O(\log U)$ ，其中 U 是值域大小
- 总体复杂度: $O(q \log U)$ ，其中 q 是操作次数

1.5 空间复杂度

- $O(q \log U)$ ，远小于传统线段树的 $O(U)$

1.6 适用场景

- 1. 值域非常大（如 10^9 ）但操作次数较少的情况
- 2. 需要处理稀疏数据的情况
- 3. 内存受限但需要处理大范围数据的情况

2. 吉司机线段树分析

2.1 基本概念

吉司机线段树是由吉如一在 2016 年国家集训队论文中提出的线段树优化技术，主要用于处理区间最值操作（如区间取 min/max）。

2.2 核心理念

1. ****维护多个信息****: 最大值、次大值、最大值个数
2. ****分类讨论****: 根据更新值与当前区间最值的关系进行不同处理
3. ****势能分析****: 通过势能分析法证明时间复杂度

2.3 实现要点

1. 维护最大值(mx)、严格次大值(sem)、最大值个数(cnt)
2. 三种更新情况:
 - 更新值 \geq 最大值: 无需更新
 - 次大值 $<$ 更新值 $<$ 最大值: 直接更新最大值
 - 更新值 \leq 次大值: 递归处理子区间
3. 合理使用懒惰标记优化

2.4 时间复杂度

- 单次操作: $O(\log^2 n)$ 均摊
- 总体复杂度: $O(n \log^2 n)$

2.5 空间复杂度

- $O(n)$

2.6 适用场景

1. 区间最值操作 (如区间取 min/max)
2. 需要维护区间最值和次最值信息的情况
3. 对时间复杂度有一定要求但可以接受均摊复杂度的情况

3. 历史最值问题分析

3.1 基本概念

历史最值问题要求维护区间的历史信息, 如历史最大值、历史最小值等。这在一些实际应用中非常有用, 如记录某个区间的峰值信息。

3.2 核心理念

1. ****多重懒惰标记****: 使用多个懒惰标记维护不同的历史信息
2. ****最大值与非最大值区分****: 区分最大值和非最大值的处理方式
3. ****历史信息维护****: 在更新过程中维护历史信息

3.3 实现要点

1. 维护区间和(sum)、最大值(mx)、次大值(sem)、最大值个数(cnt)、历史最大值(max_history)
2. 维护懒惰标记: 最大值增加量(max_add)、其他值增加量(other_add)、最大值历史最大增加量(max_add_top)、其他值历史最大增加量(other_add_top)
3. 在 push_down 过程中正确处理各种标记

3.4 时间复杂度

- 单次操作: $O(\log n)$
- 总体复杂度: $O(n \log n)$

3.5 空间复杂度

- $O(n)$

3.6 适用场景

1. 需要维护历史信息的情况
2. 区间加法和最值操作混合的情况
3. 对历史峰值信息有查询需求的情况

4. 线段树优化技巧总结

4.1 懒惰标记优化

1. ****标记下传****: 在访问子节点前正确下传标记
2. ****标记合并****: 合理设计标记的合并方式
3. ****标记清除****: 在标记下传后及时清除

4.2 离散化技术

1. ****坐标压缩****: 将大数据范围映射到小范围
2. ****避免错误****: 在相邻点间添加额外点避免合并错误
3. ****映射维护****: 维护原始坐标与离散化坐标的映射关系

4.3 动态开点优化

1. ****按需创建****: 只在需要时创建节点
2. ****内存管理****: 合理估算最大节点数
3. ****节点复用****: 在可能的情况下复用已创建的节点

4.4 势能分析法

1. ****势能函数****: 设计合理的势能函数
2. ****摊还分析****: 通过势能变化分析摊还复杂度
3. ****复杂度证明****: 使用势能分析法证明算法复杂度

5. 跨语言实现对比

5.1 Java 实现特点

1. ****面向对象****: 良好的封装性和可扩展性
2. ****内存管理****: 自动垃圾回收, 无需手动管理内存
3. ****性能****: 相对较慢但开发效率高
4. ****适用场景****: 复杂数据结构实现, 面试算法题

5.2 C++ 实现特点

1. ****性能优势****: 执行效率高, 内存控制精细

2. ****模板编程****: 支持泛型编程
3. ****指针操作****: 可以直接操作内存
4. ****适用场景****: 竞赛编程, 对性能要求高的场景

5.3 Python 实现特点

1. ****语法简洁****: 代码简洁易读
2. ****开发效率****: 开发速度快
3. ****性能****: 相对较慢但可接受
4. ****适用场景****: 快速原型开发, 教学演示

6. 工程化考虑

6.1 异常处理

1. ****输入验证****: 验证输入参数的合法性
2. ****边界检查****: 检查数组访问边界
3. ****错误恢复****: 在出错时能够正确恢复状态

6.2 性能优化

1. ****内存池****: 使用内存池技术避免频繁内存分配
2. ****缓存优化****: 合理利用 CPU 缓存
3. ****算法优化****: 选择合适的算法和数据结构

6.3 可维护性

1. ****代码结构****: 良好的代码结构和模块化设计
2. ****注释文档****: 详细的注释和文档
3. ****命名规范****: 清晰的变量和函数命名

7. 学习路径建议

7.1 基础阶段

1. 理解线段树的基本原理
2. 掌握单点更新/查询操作
3. 练习基础的区间更新/查询题目

7.2 进阶阶段

1. 学习懒惰标记技术
2. 掌握动态开点线段树
3. 练习区间最值操作题目

7.3 高级阶段

1. 学习吉司机线段树
2. 掌握历史最值维护技术
3. 练习综合应用题目

7.4 实践阶段

1. 参加编程竞赛
2. 实际项目应用
3. 算法优化和改进

8. 常见问题与解决方案

8.1 内存超限

****问题****: 传统线段树空间需求过大

****解决方案****: 使用动态开点线段树

8.2 时间超限

****问题****: 朴素实现时间复杂度过高

****解决方案****: 使用吉司机线段树或势能分析优化

8.3 精度问题

****问题****: 整数溢出或浮点数精度问题

****解决方案****: 使用合适的数据类型，如 long long

8.4 实现错误

****问题****: push_up 或 push_down 实现错误

****解决方案****: 仔细检查标记下传和信息更新逻辑

9. 未来发展方向

9.1 新兴数据结构

1. ****李超线段树****: 处理凸包相关问题
2. ****线段树分治****: 结合分治算法解决复杂问题
3. ****可持久化线段树****: 支持历史版本查询

9.2 应用拓展

1. ****机器学习****: 在机器学习中应用线段树优化
2. ****大数据处理****: 在大数据处理中应用线段树
3. ****实时系统****: 在实时系统中应用线段树

9.3 算法融合

1. ****线段树与图论****: 结合图论算法解决复杂问题
2. ****线段树与字符串****: 处理字符串相关问题
3. ****线段树与计算几何****: 解决计算几何问题

10. 总结

线段树作为一种重要的数据结构，在算法竞赛和实际应用中都有广泛的应用。通过深入学习和实践动态开点、区间最值操作和历史最值维护等高级技术，我们可以解决更多复杂的问题。

掌握线段树不仅需要理解其基本原理，还需要在实践中不断优化和完善。希望本文的分析能够帮助读者更好地理解和应用线段树，在算法学习和实践中取得更好的成绩。

=====

文件: EXTENDED_PROBLEMS.md

=====

线段树扩展题目与实现

1. POJ 3468 A Simple Problem with Integers

题目描述

给定一个长度为 N 的整数序列，执行以下操作：

1. $C\ a\ b\ c$: 将区间 $[a, b]$ 中的每个数都加上 c
2. $Q\ a\ b$: 查询区间 $[a, b]$ 中所有数的和

解题思路

这是线段树的经典应用，使用懒惰标记来优化区间更新操作。

Java 实现

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

public class POJ3468 {
 static final int MAXN = 100005;
 static long[] sum = new long[MAXN << 2];
 static long[] add = new long[MAXN << 2];
 static int[] arr = new int[MAXN];
 static int n, m;

 static void pushUp(int rt) {
 sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
 }

 static void pushDown(int rt, int ln, int rn) {
 if (add[rt] != 0) {
 add[rt << 1] += add[rt];
 add[rt << 1 | 1] += add[rt];
 sum[rt << 1] += add[rt] * ln;

```

```

 sum[rt << 1 | 1] += add[rt] * rn;
 add[rt] = 0;
 }
}

static void build(int l, int r, int rt) {
 add[rt] = 0;
 if (l == r) {
 sum[rt] = arr[l];
 return;
 }
 int mid = (l + r) >> 1;
 build(l, mid, rt << 1);
 build(mid + 1, r, rt << 1 | 1);
 pushUp(rt);
}

static void update(int L, int R, long C, int l, int r, int rt) {
 if (L <= l && r <= R) {
 sum[rt] += C * (r - l + 1);
 add[rt] += C;
 return;
 }
 int mid = (l + r) >> 1;
 pushDown(rt, mid - l + 1, r - mid);
 if (L <= mid) update(L, R, C, l, mid, rt << 1);
 if (R > mid) update(L, R, C, mid + 1, r, rt << 1 | 1);
 pushUp(rt);
}

static long query(int L, int R, int l, int r, int rt) {
 if (L <= l && r <= R) {
 return sum[rt];
 }
 int mid = (l + r) >> 1;
 pushDown(rt, mid - l + 1, r - mid);
 long ans = 0;
 if (L <= mid) ans += query(L, R, l, mid, rt << 1);
 if (R > mid) ans += query(L, R, mid + 1, r, rt << 1 | 1);
 return ans;
}

public static void main(String[] args) throws IOException {

```

```

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
StringTokenizer st = new StringTokenizer(br.readLine());
n = Integer.parseInt(st.nextToken());
m = Integer.parseInt(st.nextToken());

st = new StringTokenizer(br.readLine());
for (int i = 1; i <= n; i++) {
 arr[i] = Integer.parseInt(st.nextToken());
}

build(1, n, 1);

for (int i = 1; i <= m; i++) {
 st = new StringTokenizer(br.readLine());
 String op = st.nextToken();
 if (op.equals("C")) {
 int a = Integer.parseInt(st.nextToken());
 int b = Integer.parseInt(st.nextToken());
 long c = Long.parseLong(st.nextToken());
 update(a, b, c, 1, n, 1);
 } else {
 int a = Integer.parseInt(st.nextToken());
 int b = Integer.parseInt(st.nextToken());
 out.println(query(a, b, 1, n, 1));
 }
}
out.flush();
out.close();
}
}
}

```

### ### C++实现

```
```cpp
```

```

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

```

```

const int MAXN = 100005;
long long sum[MAXN << 2], add[MAXN << 2];
int arr[MAXN];
int n, m;

```

```

void pushUp(int rt) {
    sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
}

void pushDown(int rt, int ln, int rn) {
    if (add[rt]) {
        add[rt << 1] += add[rt];
        add[rt << 1 | 1] += add[rt];
        sum[rt << 1] += add[rt] * ln;
        sum[rt << 1 | 1] += add[rt] * rn;
        add[rt] = 0;
    }
}

void build(int l, int r, int rt) {
    add[rt] = 0;
    if (l == r) {
        scanf("%d", &arr[l]);
        sum[rt] = arr[l];
        return;
    }
    int mid = (l + r) >> 1;
    build(l, mid, rt << 1);
    build(mid + 1, r, rt << 1 | 1);
    pushUp(rt);
}

void update(int L, int R, long long C, int l, int r, int rt) {
    if (L <= l && r <= R) {
        sum[rt] += C * (r - l + 1);
        add[rt] += C;
        return;
    }
    int mid = (l + r) >> 1;
    pushDown(rt, mid - l + 1, r - mid);
    if (L <= mid) update(L, R, C, l, mid, rt << 1);
    if (R > mid) update(L, R, C, mid + 1, r, rt << 1 | 1);
    pushUp(rt);
}

long long query(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {

```

```

        return sum[rt];
    }
    int mid = (l + r) >> 1;
    pushDown(rt, mid - 1 + 1, r - mid);
    long long ans = 0;
    if (L <= mid) ans += query(L, R, 1, mid, rt << 1);
    if (R > mid) ans += query(L, R, mid + 1, r, rt << 1 | 1);
    return ans;
}

```

```

int main() {
    scanf("%d%d", &n, &m);
    build(1, n, 1);

    for (int i = 1; i <= m; i++) {
        char op[2];
        scanf("%s", op);
        if (op[0] == 'C') {
            int a, b;
            long long c;
            scanf("%d%d%lld", &a, &b, &c);
            update(a, b, c, 1, n, 1);
        } else {
            int a, b;
            scanf("%d%d", &a, &b);
            printf("%lld\n", query(a, b, 1, n, 1));
        }
    }
    return 0;
}

```

Python 实现

```

```python

```

```

import sys

```

```

class SegmentTree:

```

```

 def __init__(self, arr):
 self.n = len(arr)
 self.sum = [0] * (4 * self.n)
 self.add = [0] * (4 * self.n)
 self.arr = arr
 self.build(1, 0, self.n - 1)

```

```

def push_up(self, rt):
 self.sum[rt] = self.sum[2 * rt] + self.sum[2 * rt + 1]

def push_down(self, rt, ln, rn):
 if self.add[rt] != 0:
 self.add[2 * rt] += self.add[rt]
 self.add[2 * rt + 1] += self.add[rt]
 self.sum[2 * rt] += self.add[rt] * ln
 self.sum[2 * rt + 1] += self.add[rt] * rn
 self.add[rt] = 0

def build(self, rt, l, r):
 self.add[rt] = 0
 if l == r:
 self.sum[rt] = self.arr[l]
 return
 mid = (l + r) // 2
 self.build(2 * rt, l, mid)
 self.build(2 * rt + 1, mid + 1, r)
 self.push_up(rt)

def update(self, L, R, C, l, r, rt):
 if L <= l and r <= R:
 self.sum[rt] += C * (r - l + 1)
 self.add[rt] += C
 return
 mid = (l + r) // 2
 self.push_down(rt, mid - l + 1, r - mid)
 if L <= mid:
 self.update(L, R, C, l, mid, 2 * rt)
 if R > mid:
 self.update(L, R, C, mid + 1, r, 2 * rt + 1)
 self.push_up(rt)

def query(self, L, R, l, r, rt):
 if L <= l and r <= R:
 return self.sum[rt]
 mid = (l + r) // 2
 self.push_down(rt, mid - l + 1, r - mid)
 ans = 0
 if L <= mid:
 ans += self.query(L, R, l, mid, 2 * rt)

```

```

 if R > mid:
 ans += self.query(L, R, mid + 1, r, 2 * rt + 1)
 return ans

def main():
 input = sys.stdin.read
 data = input().split()

 idx = 0
 n = int(data[idx])
 m = int(data[idx + 1])
 idx += 2

 arr = [int(data[idx + i]) for i in range(n)]
 idx += n

 seg_tree = SegmentTree(arr)

 for _ in range(m):
 op = data[idx]
 idx += 1

 if op == 'C':
 a = int(data[idx]) - 1 # 转换为 0 索引
 b = int(data[idx + 1]) - 1
 c = int(data[idx + 2])
 idx += 3
 seg_tree.update(a, b, c, 0, n - 1, 1)
 else: # op == 'Q'
 a = int(data[idx]) - 1
 b = int(data[idx + 1]) - 1
 idx += 2
 print(seg_tree.query(a, b, 0, n - 1, 1))

if __name__ == "__main__":
 main()

```

## 2. POJ 2528 Mayor's posters

### 题目描述

城市的墙上贴海报，每张海报贴在一个连续区间上。后来贴的海报会覆盖之前贴的海报。求最后可以看到多少张不同的海报。

#### ### 解题思路

这是一个区间染色问题，由于值域很大( $10^7$ )，需要使用离散化技术。有两种处理方法：

1. 倒序处理：从后往前贴海报，只贴未被覆盖的部分
2. 离散化+线段树：将坐标离散化后用线段树维护区间颜色

#### ### Java 实现

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

public class POJ2528 {
    static final int MAXN = 10005;
    static int[] li = new int[MAXN];
    static int[] ri = new int[MAXN];
    static int[] x = new int[MAXN * 6]; // 离散化数组
    static int[] sum = new int[MAXN * 24]; // 线段树
    static int[] lazy = new int[MAXN * 24]; // 懒惰标记
    static int n, tot;

    static void pushUp(int rt) {
        sum[rt] = sum[rt << 1] | sum[rt << 1 | 1];
    }

    static void pushDown(int rt) {
        if (lazy[rt] != 0) {
            lazy[rt << 1] = lazy[rt];
            lazy[rt << 1 | 1] = lazy[rt];
            sum[rt << 1] = lazy[rt];
            sum[rt << 1 | 1] = lazy[rt];
            lazy[rt] = 0;
        }
    }

    static void build(int l, int r, int rt) {
        lazy[rt] = 0;
        sum[rt] = 0;
        if (l == r) return;
        int mid = (l + r) >> 1;
        build(l, mid, rt << 1);
        build(mid + 1, r, rt << 1 | 1);
    }
}
```

```

static void update(int L, int R, int C, int l, int r, int rt) {
    if (L <= l && r <= R) {
        sum[rt] = 1 << C;
        lazy[rt] = 1 << C;
        return;
    }
    pushDown(rt);
    int mid = (l + r) >> 1;
    if (L <= mid) update(L, R, C, l, mid, rt << 1);
    if (R > mid) update(L, R, C, mid + 1, r, rt << 1 | 1);
    pushUp(rt);
}

static int query(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return sum[rt];
    }
    pushDown(rt);
    int mid = (l + r) >> 1;
    int ans = 0;
    if (L <= mid) ans |= query(L, R, l, mid, rt << 1);
    if (R > mid) ans |= query(L, R, mid + 1, r, rt << 1 | 1);
    return ans;
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
    int T = Integer.parseInt(br.readLine());

    while (T-- > 0) {
        n = Integer.parseInt(br.readLine());
        tot = 0;

        // 读取区间并构建离散化数组
        for (int i = 1; i <= n; i++) {
            StringTokenizer st = new StringTokenizer(br.readLine());
            li[i] = Integer.parseInt(st.nextToken());
            ri[i] = Integer.parseInt(st.nextToken());
            x[++tot] = li[i];
            x[++tot] = ri[i];
        }
    }
}

```

```

// 离散化
Arrays.sort(x, 1, tot + 1);
int m = 1;
for (int i = 2; i <= tot; i++) {
    if (x[i] != x[i - 1]) {
        x[++m] = x[i];
    }
}
tot = m;

// 添加额外点避免相邻区间合并错误
m = tot;
for (int i = 1; i < m; i++) {
    if (x[i + 1] - x[i] > 1) {
        x[++tot] = x[i] + 1;
    }
}

Arrays.sort(x, 1, tot + 1);

// 构建坐标映射
HashMap<Integer, Integer> mp = new HashMap<>();
for (int i = 1; i <= tot; i++) {
    mp.put(x[i], i);
}

// 建树并倒序处理
build(1, tot, 1);
for (int i = n; i >= 1; i--) {
    int l = mp.get(li[i]);
    int r = mp.get(ri[i]);
    update(1, r, i, 1, tot, 1);
}

// 统计可见海报数
int result = query(1, tot, 1, tot, 1);
int count = 0;
for (int i = 1; i <= n; i++) {
    if ((result & (1 << i)) != 0) {
        count++;
    }
}
out.println(count);

```

```

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

```

#### C++实现

```

```cpp
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <map>
using namespace std;

const int MAXN = 10005;
int li[MAXN], ri[MAXN];
int x[MAXN * 6];
int sum[MAXN * 24], lazy[MAXN * 24];
int n, tot;

void pushUp(int rt) {
    sum[rt] = sum[rt << 1] | sum[rt << 1 | 1];
}

void pushDown(int rt) {
    if (lazy[rt]) {
        lazy[rt << 1] = lazy[rt];
        lazy[rt << 1 | 1] = lazy[rt];
        sum[rt << 1] = lazy[rt];
        sum[rt << 1 | 1] = lazy[rt];
        lazy[rt] = 0;
    }
}

void build(int l, int r, int rt) {
    lazy[rt] = 0;
    sum[rt] = 0;
    if (l == r) return;
    int mid = (l + r) >> 1;
    build(l, mid, rt << 1);
    build(mid + 1, r, rt << 1 | 1);
}

```

```

void update(int L, int R, int C, int l, int r, int rt) {
    if (L <= l && r <= R) {
        sum[rt] = 1 << C;
        lazy[rt] = 1 << C;
        return;
    }
    pushDown(rt);
    int mid = (l + r) >> 1;
    if (L <= mid) update(L, R, C, l, mid, rt << 1);
    if (R > mid) update(L, R, C, mid + 1, r, rt << 1 | 1);
    pushUp(rt);
}

int query(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return sum[rt];
    }
    pushDown(rt);
    int mid = (l + r) >> 1;
    int ans = 0;
    if (L <= mid) ans |= query(L, R, l, mid, rt << 1);
    if (R > mid) ans |= query(L, R, mid + 1, r, rt << 1 | 1);
    return ans;
}

int main() {
    int T;
    scanf("%d", &T);

    while (T--) {
        scanf("%d", &n);
        tot = 0;

        // 读取区间并构建离散化数组
        for (int i = 1; i <= n; i++) {
            scanf("%d%d", &li[i], &ri[i]);
            x[++tot] = li[i];
            x[++tot] = ri[i];
        }

        // 离散化
        sort(x + 1, x + tot + 1);
    }
}

```

```

int m = 1;
for (int i = 2; i <= tot; i++) {
    if (x[i] != x[i - 1]) {
        x[++m] = x[i];
    }
}
tot = m;

// 添加额外点避免相邻区间合并错误
m = tot;
for (int i = 1; i < m; i++) {
    if (x[i + 1] - x[i] > 1) {
        x[++tot] = x[i] + 1;
    }
}

sort(x + 1, x + tot + 1);

// 构建坐标映射
map<int, int> mp;
for (int i = 1; i <= tot; i++) {
    mp[x[i]] = i;
}

// 建树并倒序处理
build(1, tot, 1);
for (int i = n; i >= 1; i--) {
    int l = mp[li[i]];
    int r = mp[ri[i]];
    update(1, r, i, l, tot, 1);
}

// 统计可见海报数
int result = query(1, tot, 1, tot, 1);
int count = 0;
for (int i = 1; i <= n; i++) {
    if (result & (1 << i)) {
        count++;
    }
}
printf("%d\n", count);
}
return 0;

```

```
}  
...
```

Python 实现

```
```python
```

```
import sys
```

```
from collections import defaultdict
```

```
class SegmentTree:
```

```
 def __init__(self, n):
```

```
 self.n = n
```

```
 self.sum = [0] * (4 * n)
```

```
 self.lazy = [0] * (4 * n)
```

```
 def push_up(self, rt):
```

```
 self.sum[rt] = self.sum[2 * rt] | self.sum[2 * rt + 1]
```

```
 def push_down(self, rt):
```

```
 if self.lazy[rt] != 0:
```

```
 self.lazy[2 * rt] = self.lazy[rt]
```

```
 self.lazy[2 * rt + 1] = self.lazy[rt]
```

```
 self.sum[2 * rt] = self.lazy[rt]
```

```
 self.sum[2 * rt + 1] = self.lazy[rt]
```

```
 self.lazy[rt] = 0
```

```
 def build(self, l, r, rt):
```

```
 self.lazy[rt] = 0
```

```
 self.sum[rt] = 0
```

```
 if l == r:
```

```
 return
```

```
 mid = (l + r) // 2
```

```
 self.build(l, mid, 2 * rt)
```

```
 self.build(mid + 1, r, 2 * rt + 1)
```

```
 def update(self, L, R, C, l, r, rt):
```

```
 if L <= l and r <= R:
```

```
 self.sum[rt] = 1 << C
```

```
 self.lazy[rt] = 1 << C
```

```
 return
```

```
 self.push_down(rt)
```

```
 mid = (l + r) // 2
```

```
 if L <= mid:
```

```
 self.update(L, R, C, l, mid, 2 * rt)
```

```

 if R > mid:
 self.update(L, R, C, mid + 1, r, 2 * rt + 1)
 self.push_up(rt)

def query(self, L, R, l, r, rt):
 if L <= l and r <= R:
 return self.sum[rt]
 self.push_down(rt)
 mid = (l + r) // 2
 ans = 0
 if L <= mid:
 ans |= self.query(L, R, l, mid, 2 * rt)
 if R > mid:
 ans |= self.query(L, R, mid + 1, r, 2 * rt + 1)
 return ans

def main():
 input = sys.stdin.read
 data = input().split()

 idx = 0
 T = int(data[idx])
 idx += 1

 results = []

 for _ in range(T):
 n = int(data[idx])
 idx += 1

 li = [0] * (n + 1)
 ri = [0] * (n + 1)
 x = []

 # 读取区间并构建离散化数组
 for i in range(1, n + 1):
 li[i] = int(data[idx])
 ri[i] = int(data[idx + 1])
 idx += 2
 x.append(li[i])
 x.append(ri[i])

 # 离散化

```

```

x = sorted(list(set(x)))
tot = len(x)

添加额外点避免相邻区间合并错误
extra_x = []
for i in range(tot - 1):
 if x[i + 1] - x[i] > 1:
 extra_x.append(x[i] + 1)
x.extend(extra_x)
x.sort()
tot = len(x)

构建坐标映射
mp = {x[i]: i + 1 for i in range(tot)}

建树并倒序处理
seg_tree = SegmentTree(tot)
seg_tree.build(1, tot, 1)
for i in range(n, 0, -1):
 l = mp[li[i]]
 r = mp[ri[i]]
 seg_tree.update(l, r, i, 1, tot, 1)

统计可见海报数
result = seg_tree.query(1, tot, 1, tot, 1)
count = 0
for i in range(1, n + 1):
 if result & (1 << i):
 count += 1
results.append(str(count))

print('\n'.join(results))

if __name__ == "__main__":
 main()
...

```

### ## 3. Codeforces 438D The Child and Sequence

#### ### 题目描述

维护一个序列，支持以下操作：

1. 1 1 r: 查询区间  $[1, r]$  的最大值
2. 2 1 r: 查询区间  $[1, r]$  的和

3. 3 1 r x: 将区间  $[1, r]$  中的每个数对  $x$  取模

### 解题思路

这也是吉司机线段树的经典应用。当一个数对  $x$  取模后，如果这个数小于  $x$ ，则不会发生变化。我们可以维护最大值，当最大值小于模数时直接返回，否则递归处理。

### Java 实现

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

public class CF438D {
    static final int MAXN = 100005;
    static long[] sum = new long[MAXN << 2];
    static int[] max = new int[MAXN << 2];
    static int[] arr = new int[MAXN];
    static int n, m;

    static void pushUp(int rt) {
        sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
        max[rt] = Math.max(max[rt << 1], max[rt << 1 | 1]);
    }

    static void build(int l, int r, int rt) {
        if (l == r) {
            sum[rt] = max[rt] = arr[l];
            return;
        }
        int mid = (l + r) >> 1;
        build(l, mid, rt << 1);
        build(mid + 1, r, rt << 1 | 1);
        pushUp(rt);
    }

    static void update(int L, int R, int x, int l, int r, int rt) {
        if (max[rt] < x) return;
        if (l == r) {
            sum[rt] %= x;
            max[rt] %= x;
            return;
        }
        int mid = (l + r) >> 1;
        if (L <= mid) update(L, R, x, l, mid, rt << 1);
```

```

        if (R > mid) update(L, R, x, mid + 1, r, rt << 1 | 1);
        pushUp(rt);
    }

    static long querySum(int L, int R, int l, int r, int rt) {
        if (L <= l && r <= R) {
            return sum[rt];
        }
        int mid = (l + r) >> 1;
        long ans = 0;
        if (L <= mid) ans += querySum(L, R, l, mid, rt << 1);
        if (R > mid) ans += querySum(L, R, mid + 1, r, rt << 1 | 1);
        return ans;
    }

    static int queryMax(int L, int R, int l, int r, int rt) {
        if (L <= l && r <= R) {
            return max[rt];
        }
        int mid = (l + r) >> 1;
        int ans = 0;
        if (L <= mid) ans = Math.max(ans, queryMax(L, R, l, mid, rt << 1));
        if (R > mid) ans = Math.max(ans, queryMax(L, R, mid + 1, r, rt << 1 | 1));
        return ans;
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
        StringTokenizer st = new StringTokenizer(br.readLine());
        n = Integer.parseInt(st.nextToken());
        m = Integer.parseInt(st.nextToken());

        st = new StringTokenizer(br.readLine());
        for (int i = 1; i <= n; i++) {
            arr[i] = Integer.parseInt(st.nextToken());
        }

        build(1, n, 1);

        for (int i = 1; i <= m; i++) {
            st = new StringTokenizer(br.readLine());
            int op = Integer.parseInt(st.nextToken());

```

```

        if (op == 1) {
            int l = Integer.parseInt(st.nextToken());
            int r = Integer.parseInt(st.nextToken());
            out.println(querySum(l, r, 1, n, 1));
        } else if (op == 2) {
            int l = Integer.parseInt(st.nextToken());
            int r = Integer.parseInt(st.nextToken());
            int x = Integer.parseInt(st.nextToken());
            update(l, r, x, 1, n, 1);
        } else {
            int l = Integer.parseInt(st.nextToken());
            int r = Integer.parseInt(st.nextToken());
            out.println(queryMax(l, r, 1, n, 1));
        }
    }
    out.flush();
    out.close();
}
}
```

```

### ### C++实现

```

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

const int MAXN = 100005;
long long sum[MAXN << 2];
int max_val[MAXN << 2];
int arr[MAXN];
int n, m;

void pushUp(int rt) {
    sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
    max_val[rt] = max(max_val[rt << 1], max_val[rt << 1 | 1]);
}

void build(int l, int r, int rt) {
    if (l == r) {
        scanf("%d", &arr[l]);
        sum[rt] = max_val[rt] = arr[l];
        return;
    }
}

```

```

    }
    int mid = (l + r) >> 1;
    build(l, mid, rt << 1);
    build(mid + 1, r, rt << 1 | 1);
    pushUp(rt);
}

void update(int L, int R, int x, int l, int r, int rt) {
    if (max_val[rt] < x) return;
    if (l == r) {
        sum[rt] %= x;
        max_val[rt] %= x;
        return;
    }
    int mid = (l + r) >> 1;
    if (L <= mid) update(L, R, x, l, mid, rt << 1);
    if (R > mid) update(L, R, x, mid + 1, r, rt << 1 | 1);
    pushUp(rt);
}

long long querySum(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return sum[rt];
    }
    int mid = (l + r) >> 1;
    long long ans = 0;
    if (L <= mid) ans += querySum(L, R, l, mid, rt << 1);
    if (R > mid) ans += querySum(L, R, mid + 1, r, rt << 1 | 1);
    return ans;
}

int queryMax(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return max_val[rt];
    }
    int mid = (l + r) >> 1;
    int ans = 0;
    if (L <= mid) ans = max(ans, queryMax(L, R, l, mid, rt << 1));
    if (R > mid) ans = max(ans, queryMax(L, R, mid + 1, r, rt << 1 | 1));
    return ans;
}

int main() {

```

```

scanf("%d%d", &n, &m);
build(1, n, 1);

for (int i = 1; i <= m; i++) {
    int op;
    scanf("%d", &op);
    if (op == 1) {
        int l, r;
        scanf("%d%d", &l, &r);
        printf("%lld\n", querySum(1, r, 1, n, 1));
    } else if (op == 2) {
        int l, r, x;
        scanf("%d%d%d", &l, &r, &x);
        update(1, r, x, 1, n, 1);
    } else {
        int l, r;
        scanf("%d%d", &l, &r);
        printf("%d\n", queryMax(1, r, 1, n, 1));
    }
}
return 0;
}
```

```

### Python 实现

```
```python
```

```
import sys
```

```
class SegmentTree:
```

```
    def __init__(self, arr):
```

```
        self.n = len(arr)
```

```
        self.sum = [0] * (4 * self.n)
```

```
        self.max_val = [0] * (4 * self.n)
```

```
        self.arr = arr
```

```
        self.build(1, 0, self.n - 1)
```

```
    def push_up(self, rt):
```

```
        self.sum[rt] = self.sum[2 * rt] + self.sum[2 * rt + 1]
```

```
        self.max_val[rt] = max(self.max_val[2 * rt], self.max_val[2 * rt + 1])
```

```
    def build(self, rt, l, r):
```

```
        if l == r:
```

```
            self.sum[rt] = self.max_val[rt] = self.arr[l]
```

```

        return
    mid = (l + r) // 2
    self.build(2 * rt, l, mid)
    self.build(2 * rt + 1, mid + 1, r)
    self.push_up(rt)

def update(self, L, R, x, l, r, rt):
    if self.max_val[rt] < x:
        return
    if l == r:
        self.sum[rt] %= x
        self.max_val[rt] %= x
        return
    mid = (l + r) // 2
    if L <= mid:
        self.update(L, R, x, l, mid, 2 * rt)
    if R > mid:
        self.update(L, R, x, mid + 1, r, 2 * rt + 1)
    self.push_up(rt)

def query_sum(self, L, R, l, r, rt):
    if L <= l and r <= R:
        return self.sum[rt]
    mid = (l + r) // 2
    ans = 0
    if L <= mid:
        ans += self.query_sum(L, R, l, mid, 2 * rt)
    if R > mid:
        ans += self.query_sum(L, R, mid + 1, r, 2 * rt + 1)
    return ans

def query_max(self, L, R, l, r, rt):
    if L <= l and r <= R:
        return self.max_val[rt]
    mid = (l + r) // 2
    ans = 0
    if L <= mid:
        ans = max(ans, self.query_max(L, R, l, mid, 2 * rt))
    if R > mid:
        ans = max(ans, self.query_max(L, R, mid + 1, r, 2 * rt + 1))
    return ans

def main():

```

```

input = sys.stdin.read
data = input().split()

idx = 0
n = int(data[idx])
m = int(data[idx + 1])
idx += 2

arr = [int(data[idx + i]) for i in range(n)]
idx += n

seg_tree = SegmentTree(arr)

results = []
for _ in range(m):
    op = int(data[idx])
    idx += 1

    if op == 1:
        l = int(data[idx]) - 1
        r = int(data[idx + 1]) - 1
        idx += 2
        results.append(str(seg_tree.query_sum(l, r, 0, n - 1, 1)))
    elif op == 2:
        l = int(data[idx]) - 1
        r = int(data[idx + 1]) - 1
        x = int(data[idx + 2])
        idx += 3
        seg_tree.update(l, r, x, 0, n - 1, 1)
    else: # op == 3
        l = int(data[idx]) - 1
        r = int(data[idx + 1]) - 1
        idx += 2
        results.append(str(seg_tree.query_max(l, r, 0, n - 1, 1)))

print('\n'.join(results))

if __name__ == "__main__":
    main()

```

4. SPOJ GSS1 - Can you answer these queries I

题目描述

给定一个长度为 n 的整数序列，执行 m 次查询操作，每次查询 $[l, r]$ 区间内的最大子段和。

解题思路

使用线段树维护区间信息，每个节点存储以下信息：

1. 区间最大子段和 (maxSum)
2. 区间从左端点开始的最大子段和 (lSum)
3. 区间到右端点结束的最大子段和 (rSum)
4. 区间总和 (sum)

Java 实现

```
```java
// 详细实现见 Code06_MaximumSubarraySum.java
```
```

C++实现

```
```cpp
// 详细实现见 Code06_MaximumSubarraySum.cpp
```
```

Python 实现

```
```python
详细实现见 Code06_MaximumSubarraySum.py
```
```

5. LeetCode 731. 我的日程安排表 II

题目描述

设计一个日程类 `MyCalendar`，包含以下功能：有一个 `book(int start, int end)` 方法。它意味着在 `start` 到 `end` 时间内增加一个日程安排，注意，这里的时间是半开区间，即 $[start, end)$ 。当三个日程安排有一些时间上的交叉时（例如三个日程安排都在同一时间内），就会产生三重预订。每次调用 `book` 方法会产生一个日程，如果发生了三重预定则返回 `false`，并取消该日程，否则添加该日程。

解题思路

这是动态开点线段树的经典应用场景，因为时间范围可以达到 10^9 ，无法使用普通线段树。我们需要：

1. 使用动态开点线段树维护每个时间点的日程数量
2. 每次查询区间内的最大值，判断是否会导致三重预订
3. 如果不会，则更新区间

Java 实现

```
```java
import java.util.*;
```

```

class MyCalendarTwo {
 // 动态开点线段树节点定义
 static class Node {
 int max; // 当前区间的最大预订数
 int lazy; // 懒惰标记，表示待下传的增量
 Node left; // 左子节点
 Node right; // 右子节点
 }

 private Node root; // 线段树根节点
 private final int MAX_RANGE = 1_000_000_000; // 最大时间范围

 public MyCalendarTwo() {
 root = new Node();
 }

 // 区间更新：将 [L, R) 区间内的所有值加上 val
 private void update(Node node, int start, int end, int L, int R, int val) {
 // 如果当前区间完全包含在目标区间内
 if (L <= start && end <= R) {
 node.max += val;
 node.lazy += val;
 return;
 }

 // 下传懒惰标记
 pushDown(node);

 int mid = start + (end - start) / 2;
 // 更新左子区间
 if (L <= mid) {
 if (node.left == null) node.left = new Node();
 update(node.left, start, mid, L, R, val);
 }
 // 更新右子区间
 if (R > mid) {
 if (node.right == null) node.right = new Node();
 update(node.right, mid + 1, end, L, R, val);
 }

 // 上传信息
 node.max = Math.max(
 (node.left != null ? node.left.max : 0),

```

```

 (node.right != null ? node.right.max : 0)
);
}

// 查询区间 [L, R) 内的最大值
private int query(Node node, int start, int end, int L, int R) {
 // 如果当前节点为空, 返回 0
 if (node == null) return 0;

 // 如果当前区间完全包含在目标区间内
 if (L <= start && end <= R) {
 return node.max;
 }

 // 下传懒惰标记
 pushDown(node);

 int mid = start + (end - start) / 2;
 int maxVal = 0;

 // 查询左子区间
 if (L <= mid) {
 maxVal = Math.max(maxVal, query(node.left, start, mid, L, R));
 }

 // 查询右子区间
 if (R > mid) {
 maxVal = Math.max(maxVal, query(node.right, mid + 1, end, L, R));
 }

 return maxVal;
}

// 下传懒惰标记
private void pushDown(Node node) {
 if (node.lazy != 0) {
 // 为左子节点创建并传递标记
 if (node.left == null) node.left = new Node();
 node.left.max += node.lazy;
 node.left.lazy += node.lazy;

 // 为右子节点创建并传递标记
 if (node.right == null) node.right = new Node();
 node.right.max += node.lazy;
 }
}

```

```

 node.right.lazy += node.lazy;

 // 清除当前节点的懒惰标记
 node.lazy = 0;
 }
}

public boolean book(int start, int end) {
 // 先查询当前区间的最大预订数
 int currentMax = query(root, 0, MAX_RANGE, start, end - 1);

 // 如果添加后会超过 2（即出现三重预订），则返回 false
 if (currentMax >= 2) {
 return false;
 }

 // 否则，更新区间，增加预订数
 update(root, 0, MAX_RANGE, start, end - 1, 1);
 return true;
}
}

```

// 测试代码

```

class Main {
 public static void main(String[] args) {
 MyCalendarTwo myCalendar = new MyCalendarTwo();
 System.out.println(myCalendar.book(10, 20)); // 返回 true
 System.out.println(myCalendar.book(50, 60)); // 返回 true
 System.out.println(myCalendar.book(10, 40)); // 返回 true
 System.out.println(myCalendar.book(5, 15)); // 返回 false
 System.out.println(myCalendar.book(5, 10)); // 返回 true
 System.out.println(myCalendar.book(25, 55)); // 返回 true
 }
}
...

```

### C++实现

```

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

class MyCalendarTwo {

```

```

private:
    // 动态开点线段树节点定义
    struct Node {
        int maxVal;    // 当前区间的最大预订数
        int lazy;      // 懒惰标记，表示待下传的增量
        Node* left;    // 左子节点
        Node* right;   // 右子节点

        Node() : maxVal(0), lazy(0), left(nullptr), right(nullptr) {}
    };

    Node* root;        // 线段树根节点
    const int MAX_RANGE = 1e9; // 最大时间范围

    // 区间更新：将 [L, R] 区间内的所有值加上 val
    void update(Node* &node, int start, int end, int L, int R, int val) {
        if (!node) node = new Node();

        // 如果当前区间完全包含在目标区间内
        if (L <= start && end <= R) {
            node->maxVal += val;
            node->lazy += val;
            return;
        }

        // 下传懒惰标记
        pushDown(node);

        int mid = start + (end - start) / 2;
        // 更新左子区间
        if (L <= mid) {
            update(node->left, start, mid, L, R, val);
        }
        // 更新右子区间
        if (R > mid) {
            update(node->right, mid + 1, end, L, R, val);
        }

        // 上传信息
        node->maxVal = max(
            (node->left ? node->left->maxVal : 0),
            (node->right ? node->right->maxVal : 0)
        );
    }

```

```
}
```

```
// 查询区间 [L, R] 内的最大值
```

```
int query(Node* node, int start, int end, int L, int R) {
```

```
    if (!node) return 0;
```

```
    // 如果当前区间完全包含在目标区间内
```

```
    if (L <= start && end <= R) {
```

```
        return node->maxVal;
```

```
    }
```

```
    // 下传懒惰标记
```

```
    pushDown(node);
```

```
    int mid = start + (end - start) / 2;
```

```
    int maxVal = 0;
```

```
    // 查询左子区间
```

```
    if (L <= mid) {
```

```
        maxVal = max(maxVal, query(node->left, start, mid, L, R));
```

```
    }
```

```
    // 查询右子区间
```

```
    if (R > mid) {
```

```
        maxVal = max(maxVal, query(node->right, mid + 1, end, L, R));
```

```
    }
```

```
    return maxVal;
```

```
}
```

```
// 下传懒惰标记
```

```
void pushDown(Node* node) {
```

```
    if (node->lazy != 0) {
```

```
        // 为左子节点创建并传递标记
```

```
        if (!node->left) node->left = new Node();
```

```
        node->left->maxVal += node->lazy;
```

```
        node->left->lazy += node->lazy;
```

```
        // 为右子节点创建并传递标记
```

```
        if (!node->right) node->right = new Node();
```

```
        node->right->maxVal += node->lazy;
```

```
        node->right->lazy += node->lazy;
```

```
        // 清除当前节点的懒惰标记
```

```

        node->lazy = 0;
    }
}

// 释放节点内存
void freeTree(Node* node) {
    if (node) {
        freeTree(node->left);
        freeTree(node->right);
        delete node;
    }
}

public:
    MyCalendarTwo() {
        root = new Node();
    }

    ~MyCalendarTwo() {
        freeTree(root);
    }

    bool book(int start, int end) {
        // 先查询当前区间的最大预订数
        int currentMax = query(root, 0, MAX_RANGE, start, end - 1);

        // 如果添加后会超过 2（即出现三重预订），则返回 false
        if (currentMax >= 2) {
            return false;
        }

        // 否则，更新区间，增加预订数
        update(root, 0, MAX_RANGE, start, end - 1, 1);
        return true;
    }
};

// 测试代码
int main() {
    MyCalendarTwo myCalendar;
    cout << boolalpha << myCalendar.book(10, 20) << endl; // 输出 true
    cout << boolalpha << myCalendar.book(50, 60) << endl; // 输出 true
    cout << boolalpha << myCalendar.book(10, 40) << endl; // 输出 true
}

```

```

cout << boolalpha << myCalendar.book(5, 15) << endl; // 输出 false
cout << boolalpha << myCalendar.book(5, 10) << endl; // 输出 true
cout << boolalpha << myCalendar.book(25, 55) << endl; // 输出 true
return 0;
}
```

```

#### Python 实现

```

```python
class MyCalendarTwo:
    # 动态开点线段树节点定义
    class Node:
        def __init__(self):
            self.max_val = 0 # 当前区间的最大预订数
            self.lazy = 0    # 懒惰标记，表示待下传的增量
            self.left = None # 左子节点
            self.right = None # 右子节点

    def __init__(self):
        self.root = self.Node()
        self.MAX_RANGE = 10**9 # 最大时间范围

    # 区间更新：将 [L, R] 区间内的所有值加上 val
    def _update(self, node, start, end, L, R, val):
        if not node:
            node = self.Node()

        # 如果当前区间完全包含在目标区间内
        if L <= start and end <= R:
            node.max_val += val
            node.lazy += val
            return node

        # 下传懒惰标记
        self._push_down(node)

        mid = start + (end - start) // 2
        # 更新左子区间
        if L <= mid:
            if not node.left:
                node.left = self.Node()
            node.left = self._update(node.left, start, mid, L, R, val)

        # 更新右子区间

```

```

    if R > mid:
        if not node.right:
            node.right = self.Node()
        node.right = self._update(node.right, mid + 1, end, L, R, val)

# 上传信息
node.max_val = max(
    node.left.max_val if node.left else 0,
    node.right.max_val if node.right else 0
)
return node

# 查询区间 [L, R] 内的最大值
def _query(self, node, start, end, L, R):
    if not node:
        return 0

    # 如果当前区间完全包含在目标区间内
    if L <= start and end <= R:
        return node.max_val

    # 下传懒惰标记
    self._push_down(node)

    mid = start + (end - start) // 2
    max_val = 0

    # 查询左子区间
    if L <= mid:
        max_val = max(max_val, self._query(node.left, start, mid, L, R))
    # 查询右子区间
    if R > mid:
        max_val = max(max_val, self._query(node.right, mid + 1, end, L, R))

    return max_val

# 下传懒惰标记
def _push_down(self, node):
    if node.lazy != 0:
        # 为左子节点创建并传递标记
        if not node.left:
            node.left = self.Node()
        node.left.max_val += node.lazy

```

```

        node.left.lazy += node.lazy

        # 为右子节点创建并传递标记
        if not node.right:
            node.right = self.Node()
        node.right.max_val += node.lazy
        node.right.lazy += node.lazy

        # 清除当前节点的懒惰标记
        node.lazy = 0

def book(self, start: int, end: int) -> bool:
    # 先查询当前区间的最大预订数
    current_max = self._query(self.root, 0, self.MAX_RANGE, start, end - 1)

    # 如果添加后会超过 2（即出现三重预订），则返回 false
    if current_max >= 2:
        return False

    # 否则，更新区间，增加预订数
    self.root = self._update(self.root, 0, self.MAX_RANGE, start, end - 1, 1)
    return True

# 测试代码
if __name__ == "__main__":
    my_calendar = MyCalendarTwo()
    print(my_calendar.book(10, 20)) # 输出 True
    print(my_calendar.book(50, 60)) # 输出 True
    print(my_calendar.book(10, 40)) # 输出 True
    print(my_calendar.book(5, 15))  # 输出 False
    print(my_calendar.book(5, 10))  # 输出 True
    print(my_calendar.book(25, 55)) # 输出 True
    ...

```

6. LeetCode 699. 掉落的方块

题目描述

在二维平面上的 x 轴上，放置着一些方块。给你一个二维整数数组 `positions`，其中 `positions[i] = [lefti, sideLengthi]` 表示：第 i 个方块边长为 `sideLengthi`，其左侧边与 x 轴上坐标点 `lefti` 对齐。每个方块都从一个比目前所有的落地方块更高的高度掉落而下。方块沿 y 轴负方向下落，直到着陆到另一个正方形的顶边或者是 x 轴上。一旦着陆，它就会固定在原地，无法移动。在每个方块掉落后，你必须记录目前所有已经落稳的方块堆叠的最高高度。返回一个整数数组 `ans`，其中 `ans[i]` 表示在第 i 块方块掉落后堆叠的最高高度。

解题思路

这是动态开点线段树的另一个经典应用。我们需要：

1. 对于每个方块，确定其覆盖的区间 $[left, left + sideLength)$
2. 查询该区间当前的最大高度，这将是新方块的底部高度
3. 新方块的顶部高度为底部高度 + sideLength
4. 更新该区间的高度为新方块的顶部高度
5. 记录当前全局的最大高度

Java 实现

```
```java
import java.util.*;

class Solution {
 // 动态开点线段树节点定义
 static class Node {
 int max; // 当前区间的最大高度
 int lazy; // 懒惰标记，表示待下传的更新值
 Node left; // 左子节点
 Node right; // 右子节点
 }

 // 区间更新（将区间设置为某个值）
 private void update(Node node, int start, int end, int L, int R, int val) {
 // 如果当前节点为空，创建新节点
 if (node == null) {
 node = new Node();
 }

 // 如果当前区间完全包含在目标区间内，且新值大于当前最大值，则更新
 if (L <= start && end <= R && val > node.max) {
 node.max = val;
 node.lazy = val;
 return;
 }

 // 下传懒惰标记
 pushDown(node);

 int mid = start + (end - start) / 2;
 // 更新左子区间
 if (L <= mid) {
 if (node.left == null) node.left = new Node();

```

```

 update(node.left, start, mid, L, R, val);
 }
 // 更新右子区间
 if (R > mid) {
 if (node.right == null) node.right = new Node();
 update(node.right, mid + 1, end, L, R, val);
 }

 // 上传信息
 node.max = Math.max(
 (node.left != null ? node.left.max : 0),
 (node.right != null ? node.right.max : 0)
);
}

// 查询区间 [L, R] 内的最大值
private int query(Node node, int start, int end, int L, int R) {
 // 如果当前节点为空, 返回 0
 if (node == null) return 0;

 // 如果当前区间完全包含在目标区间内
 if (L <= start && end <= R) {
 return node.max;
 }

 // 下传懒惰标记
 pushDown(node);

 int mid = start + (end - start) / 2;
 int maxVal = 0;

 // 查询左子区间
 if (L <= mid) {
 maxVal = Math.max(maxVal, query(node.left, start, mid, L, R));
 }

 // 查询右子区间
 if (R > mid) {
 maxVal = Math.max(maxVal, query(node.right, mid + 1, end, L, R));
 }

 return maxVal;
}

```

```

// 下传懒惰标记
private void pushDown(Node node) {
 if (node.lazy != 0) {
 // 为左子节点创建并传递标记
 if (node.left == null) node.left = new Node();
 if (node.lazy > node.left.max) {
 node.left.max = node.lazy;
 node.left.lazy = node.lazy;
 }

 // 为右子节点创建并传递标记
 if (node.right == null) node.right = new Node();
 if (node.lazy > node.right.max) {
 node.right.max = node.lazy;
 node.right.lazy = node.lazy;
 }

 // 清除当前节点的懒惰标记
 node.lazy = 0;
 }
}

public List<Integer> fallingSquares(int[][] positions) {
 Node root = new Node();
 List<Integer> result = new ArrayList<>();
 int maxHeight = 0;

 // 确定坐标范围，进行离散化（可选优化）
 // 这里为了简化，使用较大的范围
 int MAX_RANGE = 1_000_000_000;

 for (int[] pos : positions) {
 int left = pos[0];
 int sideLength = pos[1];
 int right = left + sideLength - 1;

 // 查询当前区间的最大高度
 int currentMax = query(root, 0, MAX_RANGE, left, right);

 // 计算新方块的顶部高度
 int newHeight = currentMax + sideLength;

 // 更新区间高度
 }
}

```

```

 update(root, 0, MAX_RANGE, left, right, newHeight);

 // 更新全局最大高度
 maxHeight = Math.max(maxHeight, newHeight);
 result.add(maxHeight);
 }

 return result;
}

// 测试代码
class Main {
 public static void main(String[] args) {
 Solution solution = new Solution();

 // 测试用例 1
 int[][] positions1 = {{1, 2}, {2, 3}, {6, 1}};
 System.out.println(solution.fallingSquares(positions1)); // 输出 [2, 5, 5]

 // 测试用例 2
 int[][] positions2 = {{100, 100}, {200, 100}};
 System.out.println(solution.fallingSquares(positions2)); // 输出 [100, 100]
 }
}

```

### ### C++实现

```

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

class Solution {
private:
    // 动态开点线段树节点定义
    struct Node {
        int maxHeight; // 当前区间的最大高度
        int lazy;       // 懒惰标记, 表示待下传的更新值
        Node* left;     // 左子节点
        Node* right;    // 右子节点
    };

```

```

Node() : maxHeight(0), lazy(0), left(nullptr), right(nullptr) {}
};

// 区间更新（将区间设置为某个值，但只在新值更大时更新）
void update(Node* &node, int start, int end, int L, int R, int val) {
    if (!node) node = new Node();

    // 如果当前区间完全包含在目标区间内，且新值大于当前最大值，则更新
    if (L <= start && end <= R && val > node->maxHeight) {
        node->maxHeight = val;
        node->lazy = val;
        return;
    }

    // 下传懒惰标记
    pushDown(node);

    int mid = start + (end - start) / 2;
    // 更新左子区间
    if (L <= mid) {
        update(node->left, start, mid, L, R, val);
    }
    // 更新右子区间
    if (R > mid) {
        update(node->right, mid + 1, end, L, R, val);
    }

    // 上传信息
    node->maxHeight = max(
        (node->left ? node->left->maxHeight : 0),
        (node->right ? node->right->maxHeight : 0)
    );
}

// 查询区间 [L, R] 内的最大值
int query(Node* node, int start, int end, int L, int R) {
    if (!node) return 0;

    // 如果当前区间完全包含在目标区间内
    if (L <= start && end <= R) {
        return node->maxHeight;
    }

```

```

// 下传懒惰标记
pushDown(node);

int mid = start + (end - start) / 2;
int maxVal = 0;

// 查询左子区间
if (L <= mid) {
    maxVal = max(maxVal, query(node->left, start, mid, L, R));
}
// 查询右子区间
if (R > mid) {
    maxVal = max(maxVal, query(node->right, mid + 1, end, L, R));
}

return maxVal;
}

```

```

// 下传懒惰标记
void pushDown(Node* node) {
    if (node->lazy != 0) {
        // 为左子节点创建并传递标记
        if (!node->left) node->left = new Node();
        if (node->lazy > node->left->maxHeight) {
            node->left->maxHeight = node->lazy;
            node->left->lazy = node->lazy;
        }

        // 为右子节点创建并传递标记
        if (!node->right) node->right = new Node();
        if (node->lazy > node->right->maxHeight) {
            node->right->maxHeight = node->lazy;
            node->right->lazy = node->lazy;
        }

        // 清除当前节点的懒惰标记
        node->lazy = 0;
    }
}

```

```

// 释放节点内存
void freeTree(Node* node) {
    if (node) {

```

```

        freeTree(node->left);
        freeTree(node->right);
        delete node;
    }
}

public:
    vector<int> fallingSquares(vector<vector<int>>& positions) {
        Node* root = new Node();
        vector<int> result;
        int maxHeight = 0;

        // 确定坐标范围
        const int MAX_RANGE = 1e9;

        for (auto& pos : positions) {
            int left = pos[0];
            int sideLength = pos[1];
            int right = left + sideLength - 1;

            // 查询当前区间的最大高度
            int currentMax = query(root, 0, MAX_RANGE, left, right);

            // 计算新方块的顶部高度
            int newHeight = currentMax + sideLength;

            // 更新区间高度
            update(root, 0, MAX_RANGE, left, right, newHeight);

            // 更新全局最大高度
            maxHeight = max(maxHeight, newHeight);
            result.push_back(maxHeight);
        }

        freeTree(root);
        return result;
    }
};

// 测试代码
int main() {
    Solution solution;

```

```

// 测试用例 1
vector<vector<int>> positions1 = {{1, 2}, {2, 3}, {6, 1}};
vector<int> result1 = solution.fallingSquares(positions1);
cout << "[";
for (int i = 0; i < result1.size(); i++) {
    cout << result1[i];
    if (i < result1.size() - 1) cout << ", ";
}
cout << "]" << endl; // 输出 [2, 5, 5]

// 测试用例 2
vector<vector<int>> positions2 = {{100, 100}, {200, 100}};
vector<int> result2 = solution.fallingSquares(positions2);
cout << "[";
for (int i = 0; i < result2.size(); i++) {
    cout << result2[i];
    if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl; // 输出 [100, 100]

return 0;
}
```

```

### Python 实现

```

```python
class Solution:
    # 动态开点线段树节点定义
    class Node:
        def __init__(self):
            self.max_height = 0 # 当前区间的最大高度
            self.lazy = 0       # 懒惰标记, 表示待下传的更新值
            self.left = None    # 左子节点
            self.right = None   # 右子节点

    def fallingSquares(self, positions):
        root = self.Node()
        result = []
        max_height = 0

        # 确定坐标范围
        MAX_RANGE = 10**9
```

```

```

for pos in positions:
 left = pos[0]
 side_length = pos[1]
 right = left + side_length - 1

 # 查询当前区间的最大高度
 current_max = self._query(root, 0, MAX_RANGE, left, right)

 # 计算新方块的顶部高度
 new_height = current_max + side_length

 # 更新区间高度
 self._update(root, 0, MAX_RANGE, left, right, new_height)

 # 更新全局最大高度
 max_height = max(max_height, new_height)
 result.append(max_height)

return result

区间更新（将区间设置为某个值，但只在新值更大时更新）
def _update(self, node, start, end, L, R, val):
 if not node:
 node = self.Node()

 # 如果当前区间完全包含在目标区间内，且新值大于当前最大值，则更新
 if L <= start and end <= R and val > node.max_height:
 node.max_height = val
 node.lazy = val
 return node

 # 下传懒惰标记
 self._push_down(node)

 mid = start + (end - start) // 2
 # 更新左子区间
 if L <= mid:
 if not node.left:
 node.left = self.Node()
 node.left = self._update(node.left, start, mid, L, R, val)
 # 更新右子区间
 if R > mid:
 if not node.right:

```

```

 node.right = self.Node()
 node.right = self._update(node.right, mid + 1, end, L, R, val)

上传信息
node.max_height = max(
 node.left.max_height if node.left else 0,
 node.right.max_height if node.right else 0
)
return node

查询区间 [L, R] 内的最大值
def _query(self, node, start, end, L, R):
 if not node:
 return 0

 # 如果当前区间完全包含在目标区间内
 if L <= start and end <= R:
 return node.max_height

 # 下传懒惰标记
 self._push_down(node)

 mid = start + (end - start) // 2
 max_val = 0

 # 查询左子区间
 if L <= mid:
 max_val = max(max_val, self._query(node.left, start, mid, L, R))
 # 查询右子区间
 if R > mid:
 max_val = max(max_val, self._query(node.right, mid + 1, end, L, R))

 return max_val

下传懒惰标记
def _push_down(self, node):
 if node.lazy != 0:
 # 为左子节点创建并传递标记
 if not node.left:
 node.left = self.Node()
 if node.lazy > node.left.max_height:
 node.left.max_height = node.lazy
 node.left.lazy = node.lazy

```

```

 # 为右子节点创建并传递标记
 if not node.right:
 node.right = self.Node()
 if node.lazy > node.right.max_height:
 node.right.max_height = node.lazy
 node.right.lazy = node.lazy

 # 清除当前节点的懒惰标记
 node.lazy = 0

测试代码
if __name__ == "__main__":
 solution = Solution()

 # 测试用例 1
 positions1 = [[1, 2], [2, 3], [6, 1]]
 print(solution.fallingSquares(positions1)) # 输出 [2, 5, 5]

 # 测试用例 2
 positions2 = [[100, 100], [200, 100]]
 print(solution.fallingSquares(positions2)) # 输出 [100, 100]

```

## ## 5. SPOJ KGSS - Maximum Sum

### ### 题目描述

给定一个长度为  $n$  的整数序列，执行  $m$  次操作：

1.  $U\ i\ x$ : 将第  $i$  个位置的值更新为  $x$
2.  $Q\ l\ r$ : 查询  $[l, r]$  区间内两个最大值的和

### ### 解题思路

使用线段树维护区间信息，每个节点存储区间最大值和次大值。

### ### Java 实现

```

```java
// 详细实现见 Code07_MaximumTwoValuesSum.java
```

```

### ### C++实现

```

```cpp
// 详细实现见 Code07_MaximumTwoValuesSum.cpp
```

```

#### Python 实现

```
```python
# 详细实现见 Code07_MaximumTwoValuesSum.py
```
```

## 6. POJ 2777 - Count Color

#### 题目描述

给定一个长度为 L 的板条，初始时所有位置都是颜色 1，执行 0 次操作：

1. "C A B C"：将区间 [A, B] 染成颜色 C
2. "P A B"：查询区间 [A, B] 中有多少种不同的颜色

#### 解题思路

使用线段树维护区间信息，每个节点存储区间颜色集合(用位运算表示)，结合懒惰标记实现区间染色。

#### Java 实现

```
```java
// 详细实现见 Code08_CountColor.java
```
```

#### C++实现

```
```cpp
// 详细实现见 Code08_CountColor.cpp
```
```

#### Python 实现

```
```python
# 详细实现见 Code08_CountColor.py
```
```

## 7. LeetCode 715. Range Module

#### 题目描述

实现一个 Range 模块，支持以下操作：

1. `addRange(int left, int right)`：添加一个区间 [left, right) 到模块中
2. `queryRange(int left, int right)`：查询区间 [left, right) 是否完全被覆盖
3. `removeRange(int left, int right)`：移除区间 [left, right) 中的所有元素

#### 解题思路

使用动态开点线段树实现，因为数据范围很大 ( $10^9$ ) 但实际操作次数有限。

#### Java 实现

```
```java
```

```

public class RangeModule {
    // 动态开点线段树节点类
    private static class Node {
        int left, right; // 左右子节点索引
        boolean covered; // 当前区间是否被完全覆盖
        boolean lazy;    // 懒惰标记
    }

    private List<Node> tree; // 使用 List 动态存储节点
    private final int MAX_VAL = 1000000000;

    public RangeModule() {
        tree = new ArrayList<>();
        tree.add(new Node()); // 根节点索引为 0
    }

    // 处理懒惰标记
    private void pushDown(int node, int start, int end) {
        if (tree.get(node).lazy && start < end) {
            // 确保左右子节点存在
            if (tree.get(node).left == 0) {
                tree.add(new Node());
                tree.get(node).left = tree.size() - 1;
            }
            if (tree.get(node).right == 0) {
                tree.add(new Node());
                tree.get(node).right = tree.size() - 1;
            }

            int left = tree.get(node).left;
            int right = tree.get(node).right;

            // 下传标记
            tree.get(left).covered = tree.get(node).covered;
            tree.get(left).lazy = true;
            tree.get(right).covered = tree.get(node).covered;
            tree.get(right).lazy = true;

            // 清除当前节点标记
            tree.get(node).lazy = false;
        }
    }
}

```

// 更新区间

```
private void update(int node, int start, int end, int l, int r, boolean covered) {
    if (start > r || end < l) {
        return;
    }

    if (l <= start && end <= r) {
        tree.get(node).covered = covered;
        tree.get(node).lazy = true;
        return;
    }

    pushDown(node, start, end);
    int mid = start + (end - start) / 2;

    // 递归更新左右子树
    if (tree.get(node).left == 0) {
        tree.add(new Node());
        tree.get(node).left = tree.size() - 1;
    }
    if (tree.get(node).right == 0) {
        tree.add(new Node());
        tree.get(node).right = tree.size() - 1;
    }

    update(tree.get(node).left, start, mid, l, r, covered);
    update(tree.get(node).right, mid + 1, end, l, r, covered);
}
```

// 查询区间

```
private boolean query(int node, int start, int end, int l, int r) {
    if (start > r || end < l) {
        return true; // 不在查询范围内, 返回 true 不影响结果
    }

    if (l <= start && end <= r) {
        return tree.get(node).covered;
    }

    pushDown(node, start, end);
    int mid = start + (end - start) / 2;

    // 确保左右子节点存在
```

```

        if (tree.get(node).left == 0) {
            tree.add(new Node());
            tree.get(node).left = tree.size() - 1;
        }
        if (tree.get(node).right == 0) {
            tree.add(new Node());
            tree.get(node).right = tree.size() - 1;
        }

        return query(tree.get(node).left, start, mid, l, r) &&
            query(tree.get(node).right, mid + 1, end, l, r);
    }

    public void addRange(int left, int right) {
        update(0, 0, MAX_VAL, left, right - 1, true);
    }

    public boolean queryRange(int left, int right) {
        return query(0, 0, MAX_VAL, left, right - 1);
    }

    public void removeRange(int left, int right) {
        update(0, 0, MAX_VAL, left, right - 1, false);
    }
}

```

C++实现

```

```cpp
#include <vector>
using namespace std;

struct Node {
 int left = 0, right = 0;
 bool covered = false;
 bool lazy = false;
};

class RangeModule {
private:
 vector<Node> tree;
 const int MAX_VAL = 1e9;

```

```

void pushDown(int node, int start, int end) {
 if (tree[node].lazy && start < end) {
 // 确保左右子节点存在
 if (tree[node].left == 0) {
 tree.push_back(Node());
 tree[node].left = tree.size() - 1;
 }
 if (tree[node].right == 0) {
 tree.push_back(Node());
 tree[node].right = tree.size() - 1;
 }

 int left = tree[node].left;
 int right = tree[node].right;

 tree[left].covered = tree[node].covered;
 tree[left].lazy = true;
 tree[right].covered = tree[node].covered;
 tree[right].lazy = true;

 tree[node].lazy = false;
 }
}

void update(int node, int start, int end, int l, int r, bool covered) {
 if (start > r || end < l) return;

 if (l <= start && end <= r) {
 tree[node].covered = covered;
 tree[node].lazy = true;
 return;
 }

 pushDown(node, start, end);
 int mid = start + (end - start) / 2;

 if (tree[node].left == 0) {
 tree.push_back(Node());
 tree[node].left = tree.size() - 1;
 }
 if (tree[node].right == 0) {
 tree.push_back(Node());
 tree[node].right = tree.size() - 1;
 }
}

```

```

 }

 update(tree[node].left, start, mid, l, r, covered);
 update(tree[node].right, mid + 1, end, l, r, covered);
}

bool query(int node, int start, int end, int l, int r) {
 if (start > r || end < l) return true;

 if (l <= start && end <= r) {
 return tree[node].covered;
 }

 pushDown(node, start, end);
 int mid = start + (end - start) / 2;

 if (tree[node].left == 0) {
 tree.push_back(Node());
 tree[node].left = tree.size() - 1;
 }
 if (tree[node].right == 0) {
 tree.push_back(Node());
 tree[node].right = tree.size() - 1;
 }

 return query(tree[node].left, start, mid, l, r) &&
 query(tree[node].right, mid + 1, end, l, r);
}

public:
 RangeModule() {
 tree.push_back(Node()); // 根节点
 }

 void addRange(int left, int right) {
 update(0, 0, MAX_VAL, left, right - 1, true);
 }

 bool queryRange(int left, int right) {
 return query(0, 0, MAX_VAL, left, right - 1);
 }

 void removeRange(int left, int right) {
 update(0, 0, MAX_VAL, left, right - 1, false);
 }

```

```

 }
};
...

Python 实现
```python
class RangeModule:
    def __init__(self):
        # 使用字典动态存储节点，键为节点索引，值为节点信息
        self.tree = {0: {'left': 0, 'right': 0, 'covered': False, 'lazy': False}}
        self.node_count = 1 # 当前已使用的节点数
        self.MAX_VAL = 10**9

    def push_down(self, node, start, end):
        if self.tree[node]['lazy'] and start < end:
            # 确保左右子节点存在
            if self.tree[node]['left'] == 0:
                self.tree[node]['left'] = self.node_count
                self.tree[self.node_count] = {'left': 0, 'right': 0, 'covered': False, 'lazy':
False}

                self.node_count += 1
            if self.tree[node]['right'] == 0:
                self.tree[node]['right'] = self.node_count
                self.tree[self.node_count] = {'left': 0, 'right': 0, 'covered': False, 'lazy':
False}

                self.node_count += 1

            left_node = self.tree[node]['left']
            right_node = self.tree[node]['right']

            # 下传标记
            self.tree[left_node]['covered'] = self.tree[node]['covered']
            self.tree[left_node]['lazy'] = True
            self.tree[right_node]['covered'] = self.tree[node]['covered']
            self.tree[right_node]['lazy'] = True

            # 清除当前节点标记
            self.tree[node]['lazy'] = False

    def update(self, node, start, end, l, r, covered):
        if start > r or end < l:
            return

```

```

    if l <= start and end <= r:
        self.tree[node]['covered'] = covered
        self.tree[node]['lazy'] = True
        return

    self.push_down(node, start, end)
    mid = start + (end - start) // 2

    left_node = self.tree[node]['left']
    right_node = self.tree[node]['right']

    self.update(left_node, start, mid, l, r, covered)
    self.update(right_node, mid + 1, end, l, r, covered)

def query(self, node, start, end, l, r):
    if start > r or end < l:
        return True # 不在查询范围内

    if l <= start and end <= r:
        return self.tree[node]['covered']

    self.push_down(node, start, end)
    mid = start + (end - start) // 2

    left_node = self.tree[node]['left']
    right_node = self.tree[node]['right']

    return self.query(left_node, start, mid, l, r) and \
           self.query(right_node, mid + 1, end, l, r)

def addRange(self, left: int, right: int) -> None:
    self.update(0, 0, self.MAX_VAL, left, right - 1, True)

def queryRange(self, left: int, right: int) -> bool:
    return self.query(0, 0, self.MAX_VAL, left, right - 1)

def removeRange(self, left: int, right: int) -> None:
    self.update(0, 0, self.MAX_VAL, left, right - 1, False)

```

8. LeetCode 218. The Skyline Problem

题目描述

给定 n 座建筑物，每个建筑物由左下角坐标、右上角坐标和高度表示。找出这些建筑物在二维平面上形成的天

际线。

解题思路

使用扫描线算法结合线段树：

1. 将所有建筑物的左右边界作为事件点
2. 按 x 坐标排序事件点
3. 遍历事件点，维护当前活跃的高度集合
4. 使用线段树高效处理区间更新和最大值查询

Java 实现

```
```java
import java.util.*;

public class SkylineProblem {
 // 线段树节点类
 private static class Node {
 int maxHeight;
 int addMark; // 懒惰标记
 }

 private Node[] tree;

 // 构建线段树
 private void build(int node, int start, int end) {
 tree[node].maxHeight = 0;
 tree[node].addMark = 0;

 if (start < end) {
 int mid = start + (end - start) / 2;
 build(2 * node, start, mid);
 build(2 * node + 1, mid + 1, end);
 }
 }

 // 下传懒惰标记
 private void pushDown(int node) {
 if (tree[node].addMark != 0) {
 int leftNode = 2 * node;
 int rightNode = 2 * node + 1;

 // 更新左右子节点的最大高度
 tree[leftNode].maxHeight = Math.max(tree[leftNode].maxHeight, tree[node].addMark);
 tree[rightNode].maxHeight = Math.max(tree[rightNode].maxHeight, tree[node].addMark);
 }
 }
}
```

```

 // 更新左右子节点的懒惰标记
 tree[leftNode].addMark = Math.max(tree[leftNode].addMark, tree[node].addMark);
 tree[rightNode].addMark = Math.max(tree[rightNode].addMark, tree[node].addMark);

 // 清除当前节点的懒惰标记
 tree[node].addMark = 0;
 }
}

// 更新区间
private void update(int node, int start, int end, int l, int r, int height) {
 if (start > r || end < l) {
 return;
 }

 if (l <= start && end <= r) {
 tree[node].maxHeight = Math.max(tree[node].maxHeight, height);
 tree[node].addMark = Math.max(tree[node].addMark, height);
 return;
 }

 pushDown(node);
 int mid = start + (end - start) / 2;
 update(2 * node, start, mid, l, r, height);
 update(2 * node + 1, mid + 1, end, l, r, height);

 // 更新当前节点的最大高度
 tree[node].maxHeight = Math.max(tree[2 * node].maxHeight, tree[2 * node + 1].maxHeight);
}

// 查询单点
private int query(int node, int start, int end, int idx) {
 if (start == end) {
 return tree[node].maxHeight;
 }

 pushDown(node);
 int mid = start + (end - start) / 2;
 if (idx <= mid) {
 return query(2 * node, start, mid, idx);
 } else {
 return query(2 * node + 1, mid + 1, end, idx);
 }
}

```

```
}
}
```

```
public List<List<Integer>> getSkyline(int[][] buildings) {
 // 收集所有 x 坐标并排序去重
 Set<Integer> xSet = new TreeSet<>();
 for (int[] building : buildings) {
 xSet.add(building[0]);
 xSet.add(building[1]);
 }
 List<Integer> xCoords = new ArrayList<>(xSet);
 Map<Integer, Integer> xToIdx = new HashMap<>();
 for (int i = 0; i < xCoords.size(); i++) {
 xToIdx.put(xCoords.get(i), i);
 }

 // 初始化线段树
 int n = xCoords.size();
 tree = new Node[4 * n];
 for (int i = 0; i < 4 * n; i++) {
 tree[i] = new Node();
 }
 build(1, 0, n - 1);

 // 处理所有建筑物
 for (int[] building : buildings) {
 int l = xToIdx.get(building[0]);
 int r = xToIdx.get(building[1]) - 1; // 闭区间
 int height = building[2];
 if (l <= r) {
 update(1, 0, n - 1, l, r, height);
 }
 }

 // 生成天际线
 List<List<Integer>> result = new ArrayList<>();
 int prevHeight = 0;
 for (int i = 0; i < xCoords.size(); i++) {
 int currentHeight = query(1, 0, n - 1, i);
 if (currentHeight != prevHeight) {
 result.add(Arrays.asList(xCoords.get(i), currentHeight));
 prevHeight = currentHeight;
 }
 }
}
```

```

 }
 return result;
}
}

```

#### C++实现

```
```cpp
```

```

#include <vector>
#include <set>
#include <map>
#include <algorithm>
using namespace std;

```

```

struct Node {
    int maxHeight = 0;
    int addMark = 0;
};

```

```

class SkylineProblem {
private:

```

```
    vector<Node> tree;
```

```

    void build(int node, int start, int end) {
        tree[node].maxHeight = 0;
        tree[node].addMark = 0;

        if (start < end) {
            int mid = start + (end - start) / 2;
            build(2 * node, start, mid);
            build(2 * node + 1, mid + 1, end);
        }
    }

```

```

    void pushDown(int node) {
        if (tree[node].addMark != 0) {
            int leftNode = 2 * node;
            int rightNode = 2 * node + 1;

            tree[leftNode].maxHeight = max(tree[leftNode].maxHeight, tree[node].addMark);
            tree[rightNode].maxHeight = max(tree[rightNode].maxHeight, tree[node].addMark);

            tree[leftNode].addMark = max(tree[leftNode].addMark, tree[node].addMark);
            tree[rightNode].addMark = max(tree[rightNode].addMark, tree[node].addMark);
        }
    }

```

```

        tree[node].addMark = 0;
    }
}

void update(int node, int start, int end, int l, int r, int height) {
    if (start > r || end < l) {
        return;
    }

    if (l <= start && end <= r) {
        tree[node].maxHeight = max(tree[node].maxHeight, height);
        tree[node].addMark = max(tree[node].addMark, height);
        return;
    }

    pushDown(node);
    int mid = start + (end - start) / 2;
    update(2 * node, start, mid, l, r, height);
    update(2 * node + 1, mid + 1, end, l, r, height);

    tree[node].maxHeight = max(tree[2 * node].maxHeight, tree[2 * node + 1].maxHeight);
}

int query(int node, int start, int end, int idx) {
    if (start == end) {
        return tree[node].maxHeight;
    }

    pushDown(node);
    int mid = start + (end - start) / 2;
    if (idx <= mid) {
        return query(2 * node, start, mid, idx);
    } else {
        return query(2 * node + 1, mid + 1, end, idx);
    }
}

public:
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        set<int> xSet;
        for (const auto& building : buildings) {
            xSet.insert(building[0]);
            xSet.insert(building[1]);

```

```

    }

    vector<int> xCoords(xSet.begin(), xSet.end());
    map<int, int> xToIdx;
    for (int i = 0; i < xCoords.size(); i++) {
        xToIdx[xCoords[i]] = i;
    }

    int n = xCoords.size();
    tree.resize(4 * n);
    build(1, 0, n - 1);

    for (const auto& building : buildings) {
        int l = xToIdx[building[0]];
        int r = xToIdx[building[1]] - 1; // 闭区间
        int height = building[2];
        if (l <= r) {
            update(1, 0, n - 1, l, r, height);
        }
    }

    vector<vector<int>> result;
    int prevHeight = 0;
    for (int i = 0; i < xCoords.size(); i++) {
        int currentHeight = query(1, 0, n - 1, i);
        if (currentHeight != prevHeight) {
            result.push_back({xCoords[i], currentHeight});
            prevHeight = currentHeight;
        }
    }
    return result;
}

};
...

```

Python 实现

```
```python
```

```
class SkylineProblem:
```

```
 def __init__(self):
```

```
 self.tree = [] # 线段树数组
```

```
 def build(self, node, start, end):
```

```
 # 确保树的大小足够
```

```
 if len(self.tree) <= node:
```

```

 self.tree += [{'maxHeight': 0, 'addMark': 0} for _ in range(node - len(self.tree) +
1)]

self.tree[node]['maxHeight'] = 0
self.tree[node]['addMark'] = 0

if start < end:
 mid = start + (end - start) // 2
 self.build(2 * node, start, mid)
 self.build(2 * node + 1, mid + 1, end)

def push_down(self, node):
 if len(self.tree) <= node:
 return

 if self.tree[node]['addMark'] != 0:
 # 确保左右子节点存在
 left_node = 2 * node
 right_node = 2 * node + 1

 if len(self.tree) <= left_node:
 self.tree += [{'maxHeight': 0, 'addMark': 0} for _ in range(left_node -
len(self.tree) + 1)]
 if len(self.tree) <= right_node:
 self.tree += [{'maxHeight': 0, 'addMark': 0} for _ in range(right_node -
len(self.tree) + 1)]

 # 更新左右子节点
 self.tree[left_node]['maxHeight'] = max(self.tree[left_node]['maxHeight'],
self.tree[node]['addMark'])
 self.tree[right_node]['maxHeight'] = max(self.tree[right_node]['maxHeight'],
self.tree[node]['addMark'])

 self.tree[left_node]['addMark'] = max(self.tree[left_node]['addMark'],
self.tree[node]['addMark'])
 self.tree[right_node]['addMark'] = max(self.tree[right_node]['addMark'],
self.tree[node]['addMark'])

 # 清除当前节点标记
 self.tree[node]['addMark'] = 0

def update(self, node, start, end, l, r, height):
 if start > r or end < l:

```

```

 return

确保节点存在
if len(self.tree) <= node:
 self.tree += [{'maxHeight': 0, 'addMark': 0} for _ in range(node - len(self.tree) +
1)]

if l <= start and end <= r:
 self.tree[node]['maxHeight'] = max(self.tree[node]['maxHeight'], height)
 self.tree[node]['addMark'] = max(self.tree[node]['addMark'], height)
 return

self.push_down(node)
mid = start + (end - start) // 2
self.update(2 * node, start, mid, l, r, height)
self.update(2 * node + 1, mid + 1, end, l, r, height)

确保左右子节点存在
left_node = 2 * node
right_node = 2 * node + 1
if len(self.tree) <= left_node:
 self.tree += [{'maxHeight': 0, 'addMark': 0} for _ in range(left_node -
len(self.tree) + 1)]
if len(self.tree) <= right_node:
 self.tree += [{'maxHeight': 0, 'addMark': 0} for _ in range(right_node -
len(self.tree) + 1)]

更新当前节点的最大高度
self.tree[node]['maxHeight'] = max(self.tree[left_node]['maxHeight'],
self.tree[right_node]['maxHeight'])

def query(self, node, start, end, idx):
 # 确保节点存在
 if len(self.tree) <= node:
 self.tree += [{'maxHeight': 0, 'addMark': 0} for _ in range(node - len(self.tree) +
1)]

 if start == end:
 return self.tree[node]['maxHeight']

 self.push_down(node)
 mid = start + (end - start) // 2
 if idx <= mid:

```

```

 return self.query(2 * node, start, mid, idx)
 else:
 return self.query(2 * node + 1, mid + 1, end, idx)

def get_skyline(self, buildings):
 # 收集所有 x 坐标并排序去重
 x_set = set()
 for building in buildings:
 x_set.add(building[0])
 x_set.add(building[1])
 x_coords = sorted(x_set)
 x_to_idx = {x: i for i, x in enumerate(x_coords)}

 n = len(x_coords)
 self.tree = [] # 重置线段树
 self.build(1, 0, n - 1)

 # 处理所有建筑物
 for building in buildings:
 l = x_to_idx[building[0]]
 r = x_to_idx[building[1]] - 1 # 闭区间
 height = building[2]
 if l <= r:
 self.update(1, 0, n - 1, l, r, height)

 # 生成天际线
 result = []
 prev_height = 0
 for i in range(n):
 current_height = self.query(1, 0, n - 1, i)
 if current_height != prev_height:
 result.append([x_coords[i], current_height])
 prev_height = current_height

 return result
'''

总结

线段树的应用场景与技巧

```

线段树作为一种强大的数据结构，适用于以下场景：

1. **\*\*区间查询与更新\*\***:
  - 区间求和、最大值、最小值查询
  - 区间加法、乘法、赋值操作
  - 时间复杂度:  $O(\log n)$  per operation
2. **\*\*离散化技术\*\***:
  - 当数据范围很大但实际用到的点较少时
  - 如 POJ 2528 Mayor's posters 中的坐标压缩
3. **\*\*懒惰标记优化\*\***:
  - 延迟传播更新操作, 避免不必要的子树访问
  - 适用于区间批量操作
  - 注意标记的合并顺序与优先级
4. **\*\*动态开点线段树\*\***:
  - 适用于值域范围极大的场景 (如  $10^9$ )
  - 按需创建节点, 节省空间
  - 如 LeetCode 715 Range Module
5. **\*\*线段树的扩展变体\*\***:
  - 吉司机线段树: 处理区间取模等操作
  - 线段树维护多个信息: 如最大子段和问题需要维护四个值
  - 扫描线算法结合线段树: 如 LeetCode 218 Skyline Problem
6. **\*\*括号匹配问题\*\***:
  - 维护区间内的括号匹配情况
  - 需要记录左括号数量、右括号数量、匹配数

### ### 解题技巧总结

1. **\*\*节点信息设计\*\***:
  - 根据问题需求设计合适的节点结构
  - 考虑需要维护哪些区间信息
  - 如何高效合并子节点信息
2. **\*\*懒惰标记处理\*\***:
  - 正确处理标记的下传顺序
  - 注意标记的叠加规则 (如多次加法的合并)
  - 确保不会遗漏标记的传播
3. **\*\*边界条件处理\*\***:
  - 单点更新与查询的正确性
  - 区间不完全覆盖时的递归处理

- 避免数组越界等低级错误

#### 4. **\*\*性能优化\*\***:

- 离散化处理大范围数据
- 动态开点减少空间使用
- 合理预估线段树数组大小（通常为  $4*n$ ）

#### 5. **\*\*多语言实现注意事项\*\***:

- Java 中注意整数溢出问题
- C++中注意 vector 的预分配大小
- Python 中注意递归深度限制（可能需要非递归实现）

### ## 9. Codeforces 446C - DZY Loves Fibonacci Numbers

#### #### 题目描述

给定一个序列，支持两种操作：

1. 将区间  $[L, R]$  加上斐波那契数列  $(F[1], F[2], \dots, F[R-L+1])$
2. 查询区间  $[L, R]$  的和

#### #### 解题思路

使用线段树维护区间和，并结合数学性质：斐波那契数列的前缀和满足一定规律，可以通过维护两个参数  $a$  和  $b$  来表示斐波那契数列的起始项。

#### #### Java 实现

```
```java
public class DZYLovesFibonacciNumbers {
    private static final int MOD = 1000000009;

    static class Node {
        long sum;        // 区间和
        long a, b;        // 斐波那契增量的参数
        boolean hasAdd; // 是否有增量标记
    }

    private Node[] tree;
    private long[] F, S; // 斐波那契数列和前缀和数组

    // 快速幂求逆元
    private long pow(long a, long b) {
        long res = 1;
        while (b > 0) {
            if (b % 2 == 1) res = res * a % MOD;
            a = a * a % MOD;
        }
    }
}
```

```

        b /= 2;
    }
    return res;
}

// 预处理斐波那契数列和前缀和
private void precompute(int n) {
    F = new long[n + 3];
    S = new long[n + 3];
    F[1] = 1; F[2] = 1;
    S[1] = 1; S[2] = 2;

    for (int i = 3; i <= n; i++) {
        F[i] = (F[i-1] + F[i-2]) % MOD;
        S[i] = (S[i-1] + F[i]) % MOD;
    }
}

// 计算斐波那契数列的第 n 项
private long fib(long n) {
    if (n <= 0) return 0;
    if (n <= F.length - 1) return F[(int)n];
    // 对于大 n 可以用矩阵快速幂，但这里假设预处理足够大
    return 0;
}

// 计算斐波那契数列前 n 项和
private long sumFib(long n) {
    if (n <= 0) return 0;
    if (n <= S.length - 1) return S[(int)n];
    // 公式:  $S(n) = F(n+2) - 1$ 
    return (fib(n+2) - 1 + MOD) % MOD;
}

// 计算区间[L, R]的增量和
private long getAddSum(long a, long b, long L, long R) {
    long len = R - L + 1;
    // 斐波那契数列的区间和
    long term1 = a * (sumFib(len + 1) - 1 + MOD) % MOD;
    long term2 = b * (sumFib(len) - 1 + MOD) % MOD;
    return (term1 + term2) % MOD;
}

```

// 下传标记

```
private void pushDown(int node, int start, int end) {
    if (tree[node].hasAdd) {
        int left = 2 * node;
        int right = 2 * node + 1;
        int mid = start + (end - start) / 2;

        // 计算左右子区间的增量参数
        long a = tree[node].a;
        long b = tree[node].b;

        // 左子区间: [start, mid]
        tree[left].a = (tree[left].a + a) % MOD;
        tree[left].b = (tree[left].b + b) % MOD;
        tree[left].sum = (tree[left].sum + getAddSum(a, b, 1, mid - start + 1)) % MOD;
        tree[left].hasAdd = true;

        // 右子区间: [mid+1, end], 起始项为 F[mid-start+2], F[mid-start+3]
        long newA = (a * F[mid - start + 1] % MOD + b * F[mid - start] % MOD) % MOD;
        long newB = (a * F[mid - start + 2] % MOD + b * F[mid - start + 1] % MOD) % MOD;
        tree[right].a = (tree[right].a + newA) % MOD;
        tree[right].b = (tree[right].b + newB) % MOD;
        tree[right].sum = (tree[right].sum + getAddSum(newA, newB, 1, end - mid)) % MOD;
        tree[right].hasAdd = true;

        // 清除当前节点标记
        tree[node].a = 0;
        tree[node].b = 0;
        tree[node].hasAdd = false;
    }
}
```

// 构建线段树

```
private void build(int node, int start, int end, long[] arr) {
    tree[node].sum = 0;
    tree[node].a = 0;
    tree[node].b = 0;
    tree[node].hasAdd = false;

    if (start == end) {
        tree[node].sum = arr[start] % MOD;
        return;
    }
}
```

```

    int mid = start + (end - start) / 2;
    build(2 * node, start, mid, arr);
    build(2 * node + 1, mid + 1, end, arr);
    tree[node].sum = (tree[2 * node].sum + tree[2 * node + 1].sum) % MOD;
}

```

// 区间更新

```

private void update(int node, int start, int end, int l, int r) {
    if (start > r || end < l) return;

    if (l <= start && end <= r) {
        // 计算该区间对应的斐波那契起始项
        long a = F[start - l + 1];
        long b = F[start - l + 2];

        tree[node].a = (tree[node].a + a) % MOD;
        tree[node].b = (tree[node].b + b) % MOD;
        tree[node].sum = (tree[node].sum + getAddSum(a, b, l, end - start + 1)) % MOD;
        tree[node].hasAdd = true;
        return;
    }

    pushDown(node, start, end);
    int mid = start + (end - start) / 2;
    update(2 * node, start, mid, l, r);
    update(2 * node + 1, mid + 1, end, l, r);
    tree[node].sum = (tree[2 * node].sum + tree[2 * node + 1].sum) % MOD;
}

```

// 区间查询

```

private long query(int node, int start, int end, int l, int r) {
    if (start > r || end < l) return 0;

    if (l <= start && end <= r) {
        return tree[node].sum;
    }

    pushDown(node, start, end);
    int mid = start + (end - start) / 2;
    long leftSum = query(2 * node, start, mid, l, r);
    long rightSum = query(2 * node + 1, mid + 1, end, l, r);
    return (leftSum + rightSum) % MOD;
}

```

```

    }

    public static void main(String[] args) {
        // 示例使用
    }
}

```

C++实现

```
```cpp
```

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
const int MOD = 1000000009;
```

```
struct Node {
 long long sum;
 long long a, b;
 bool hasAdd;
 Node() : sum(0), a(0), b(0), hasAdd(false) {}
};

```

```
class DZYLovesFibonacciNumbers {
```

```
private:
```

```
 vector<Node> tree;
```

```
 vector<long long> F, S;
```

```
 long long pow(long long a, long long b) {
 long long res = 1;
 while (b > 0) {
 if (b % 2 == 1) res = res * a % MOD;
 a = a * a % MOD;
 b /= 2;
 }
 return res;
 }

```

```
 void precompute(int n) {
 F.resize(n + 3);
 S.resize(n + 3);
 F[1] = 1; F[2] = 1;
 S[1] = 1; S[2] = 2;
 }

```

```

 for (int i = 3; i <= n; i++) {
 F[i] = (F[i-1] + F[i-2]) % MOD;
 S[i] = (S[i-1] + F[i]) % MOD;
 }
}

```

```

long long fib(long long n) {
 if (n <= 0) return 0;
 if (n < F.size()) return F[n];
 // 对于大 n 可以用矩阵快速幂
 return 0;
}

```

```

long long sumFib(long long n) {
 if (n <= 0) return 0;
 if (n < S.size()) return S[n];
 // 公式: $S(n) = F(n+2) - 1$
 return (fib(n+2) - 1 + MOD) % MOD;
}

```

```

long long getAddSum(long long a, long long b, long long L, long long R) {
 long long len = R - L + 1;
 long long term1 = a * (sumFib(len + 1) - 1 + MOD) % MOD;
 long long term2 = b * (sumFib(len) - 1 + MOD) % MOD;
 return (term1 + term2) % MOD;
}

```

```

void pushDown(int node, int start, int end) {
 if (tree[node].hasAdd) {
 int left = 2 * node;
 int right = 2 * node + 1;
 int mid = start + (end - start) / 2;

 long long a = tree[node].a;
 long long b = tree[node].b;

 // 左子区间
 tree[left].a = (tree[left].a + a) % MOD;
 tree[left].b = (tree[left].b + b) % MOD;
 tree[left].sum = (tree[left].sum + getAddSum(a, b, 1, mid - start + 1)) % MOD;
 tree[left].hasAdd = true;

 // 右子区间

```

```

 long long newA = (a * F[mid - start + 1] % MOD + b * F[mid - start] % MOD) % MOD;
 long long newB = (a * F[mid - start + 2] % MOD + b * F[mid - start + 1] % MOD) % MOD;
 tree[right].a = (tree[right].a + newA) % MOD;
 tree[right].b = (tree[right].b + newB) % MOD;
 tree[right].sum = (tree[right].sum + getAddSum(newA, newB, 1, end - mid)) % MOD;
 tree[right].hasAdd = true;

 // 清除标记
 tree[node].a = 0;
 tree[node].b = 0;
 tree[node].hasAdd = false;
 }
}

void build(int node, int start, int end, const vector<long long>& arr) {
 if (start == end) {
 tree[node].sum = arr[start] % MOD;
 return;
 }

 int mid = start + (end - start) / 2;
 build(2 * node, start, mid, arr);
 build(2 * node + 1, mid + 1, end, arr);
 tree[node].sum = (tree[2 * node].sum + tree[2 * node + 1].sum) % MOD;
}

void update(int node, int start, int end, int l, int r) {
 if (start > r || end < l) return;

 if (l <= start && end <= r) {
 long long a = F[start - 1 + 1];
 long long b = F[start - 1 + 2];

 tree[node].a = (tree[node].a + a) % MOD;
 tree[node].b = (tree[node].b + b) % MOD;
 tree[node].sum = (tree[node].sum + getAddSum(a, b, 1, end - start + 1)) % MOD;
 tree[node].hasAdd = true;
 return;
 }

 pushDown(node, start, end);
 int mid = start + (end - start) / 2;
 update(2 * node, start, mid, l, r);

```

```

 update(2 * node + 1, mid + 1, end, l, r);
 tree[node].sum = (tree[2 * node].sum + tree[2 * node + 1].sum) % MOD;
 }

 long long query(int node, int start, int end, int l, int r) {
 if (start > r || end < l) return 0;

 if (l <= start && end <= r) {
 return tree[node].sum;
 }

 pushDown(node, start, end);
 int mid = start + (end - start) / 2;
 long long leftSum = query(2 * node, start, mid, l, r);
 long long rightSum = query(2 * node + 1, mid + 1, end, l, r);
 return (leftSum + rightSum) % MOD;
 }
public:
 void solve() {
 int n, m;
 cin >> n >> m;

 precompute(n);
 tree.resize(4 * n);

 vector<long long> arr(n + 1);
 for (int i = 1; i <= n; i++) {
 cin >> arr[i];
 }
 build(1, 1, n, arr);

 while (m--) {
 int op, l, r;
 cin >> op >> l >> r;
 if (op == 1) {
 update(1, 1, n, l, r);
 } else {
 cout << query(1, 1, n, l, r) << endl;
 }
 }
 }
};

```

### Python 实现

```
```python
```

```
MOD = 10**9 + 9
```

```
class DZYLovesFibonacciNumbers:
```

```
    def __init__(self):
```

```
        self.tree = []
```

```
        self.F = []
```

```
        self.S = []
```

```
    def pow_mod(self, a, b):
```

```
        res = 1
```

```
        while b > 0:
```

```
            if b % 2 == 1:
```

```
                res = res * a % MOD
```

```
            a = a * a % MOD
```

```
            b //= 2
```

```
        return res
```

```
    def precompute(self, n):
```

```
        self.F = [0] * (n + 3)
```

```
        self.S = [0] * (n + 3)
```

```
        self.F[1] = 1
```

```
        self.F[2] = 1
```

```
        self.S[1] = 1
```

```
        self.S[2] = 2
```

```
        for i in range(3, n + 1):
```

```
            self.F[i] = (self.F[i-1] + self.F[i-2]) % MOD
```

```
            self.S[i] = (self.S[i-1] + self.F[i]) % MOD
```

```
    def fib(self, n):
```

```
        if n <= 0:
```

```
            return 0
```

```
        if n < len(self.F):
```

```
            return self.F[n]
```

```
        # 对于大 n 可以用矩阵快速幂
```

```
        return 0
```

```
    def sum_fib(self, n):
```

```
        if n <= 0:
```

```
            return 0
```

```

    if n < len(self.S):
        return self.S[n]
    # 公式:  $S(n) = F(n+2) - 1$ 
    return (self.fib(n+2) - 1 + MOD) % MOD

def get_add_sum(self, a, b, L, R):
    len_ = R - L + 1
    term1 = a * (self.sum_fib(len_ + 1) - 1 + MOD) % MOD
    term2 = b * (self.sum_fib(len_) - 1 + MOD) % MOD
    return (term1 + term2) % MOD

def push_down(self, node, start, end):
    if self.tree[node]['hasAdd']:
        left = 2 * node
        right = 2 * node + 1
        mid = start + (end - start) // 2

        a = self.tree[node]['a']
        b = self.tree[node]['b']

        # 确保左右子节点存在
        if len(self.tree) <= left:
            self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(left - len(self.tree) + 1)]
        if len(self.tree) <= right:
            self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(right - len(self.tree) + 1)]

        # 左子区间
        self.tree[left]['a'] = (self.tree[left]['a'] + a) % MOD
        self.tree[left]['b'] = (self.tree[left]['b'] + b) % MOD
        self.tree[left]['sum'] = (self.tree[left]['sum'] + self.get_add_sum(a, b, 1, mid - start + 1)) % MOD
        self.tree[left]['hasAdd'] = True

        # 右子区间
        new_a = (a * self.F[mid - start + 1] % MOD + b * self.F[mid - start] % MOD) % MOD
        new_b = (a * self.F[mid - start + 2] % MOD + b * self.F[mid - start + 1] % MOD) % MOD
        self.tree[right]['a'] = (self.tree[right]['a'] + new_a) % MOD
        self.tree[right]['b'] = (self.tree[right]['b'] + new_b) % MOD
        self.tree[right]['sum'] = (self.tree[right]['sum'] + self.get_add_sum(new_a, new_b, 1, end - mid)) % MOD
        self.tree[right]['hasAdd'] = True

```

```

        # 清除标记
        self.tree[node]['a'] = 0
        self.tree[node]['b'] = 0
        self.tree[node]['hasAdd'] = False

def build(self, node, start, end, arr):
    # 确保节点存在
    if len(self.tree) <= node:
        self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(node -
len(self.tree) + 1)]

    if start == end:
        self.tree[node]['sum'] = arr[start] % MOD
        return

    mid = start + (end - start) // 2
    self.build(2 * node, start, mid, arr)
    self.build(2 * node + 1, mid + 1, end, arr)
    # 确保左右子节点存在
    left = 2 * node
    right = 2 * node + 1
    if len(self.tree) <= left:
        self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(left -
len(self.tree) + 1)]
    if len(self.tree) <= right:
        self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(right -
len(self.tree) + 1)]

    self.tree[node]['sum'] = (self.tree[left]['sum'] + self.tree[right]['sum']) % MOD

def update(self, node, start, end, l, r):
    if start > r or end < l:
        return

    # 确保节点存在
    if len(self.tree) <= node:
        self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(node -
len(self.tree) + 1)]

    if l <= start and end <= r:
        a = self.F[start - 1 + 1]
        b = self.F[start - 1 + 2]

```

```

        self.tree[node]['a'] = (self.tree[node]['a'] + a) % MOD
        self.tree[node]['b'] = (self.tree[node]['b'] + b) % MOD
        self.tree[node]['sum'] = (self.tree[node]['sum'] + self.get_add_sum(a, b, 1, end -
start + 1)) % MOD
        self.tree[node]['hasAdd'] = True
        return

    self.push_down(node, start, end)
    mid = start + (end - start) // 2
    self.update(2 * node, start, mid, 1, r)
    self.update(2 * node + 1, mid + 1, end, 1, r)
    # 确保左右子节点存在
    left = 2 * node
    right = 2 * node + 1
    if len(self.tree) <= left:
        self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(left -
len(self.tree) + 1)]
    if len(self.tree) <= right:
        self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(right -
len(self.tree) + 1)]

    self.tree[node]['sum'] = (self.tree[left]['sum'] + self.tree[right]['sum']) % MOD

def query(self, node, start, end, l, r):
    if start > r or end < l:
        return 0

    # 确保节点存在
    if len(self.tree) <= node:
        self.tree += [{'sum': 0, 'a': 0, 'b': 0, 'hasAdd': False} for _ in range(node -
len(self.tree) + 1)]

    if l <= start and end <= r:
        return self.tree[node]['sum']

    self.push_down(node, start, end)
    mid = start + (end - start) // 2
    left_sum = self.query(2 * node, start, mid, l, r)
    right_sum = self.query(2 * node + 1, mid + 1, end, l, r)
    return (left_sum + right_sum) % MOD

def solve(self):

```

示例使用

pass

10. SPOJ HORRIBLE - Horrible Queries

题目描述

给定一个数组，支持两种操作：

1. 将区间[L, R]的每个元素加上一个值 X
2. 查询区间[L, R]的元素和

解题思路

使用线段树维护区间和，并结合懒惰标记进行区间更新优化。

Java 实现

```
```java
public class HorribleQueries {
 static class Node {
 long sum; // 区间和
 long addMark; // 加法标记
 }

 private Node[] tree;
 private int n;

 public HorribleQueries(int[] arr) {
 n = arr.length;
 tree = new Node[4 * n];
 for (int i = 0; i < 4 * n; i++) {
 tree[i] = new Node();
 }
 build(1, 0, n - 1, arr);
 }

 // 下传标记
 private void pushDown(int node, int start, int end) {
 if (tree[node].addMark != 0) {
 int left = 2 * node;
 int right = 2 * node + 1;
 int mid = start + (end - start) / 2;

 // 左右子区间的长度
 long leftLen = mid - start + 1;
 long rightLen = end - mid;
```

```

 // 更新左右子节点的和
 tree[left].sum += tree[node].addMark * leftLen;
 tree[right].sum += tree[node].addMark * rightLen;

 // 传递标记
 tree[left].addMark += tree[node].addMark;
 tree[right].addMark += tree[node].addMark;

 // 清除当前节点的标记
 tree[node].addMark = 0;
 }
}

// 构建线段树
private void build(int node, int start, int end, int[] arr) {
 if (start == end) {
 tree[node].sum = arr[start];
 return;
 }

 int mid = start + (end - start) / 2;
 build(2 * node, start, mid, arr);
 build(2 * node + 1, mid + 1, end, arr);
 tree[node].sum = tree[2 * node].sum + tree[2 * node + 1].sum;
}

// 区间更新
private void update(int node, int start, int end, int l, int r, long val) {
 if (start > r || end < l) return;

 if (l <= start && end <= r) {
 // 当前区间完全包含在目标区间内
 tree[node].sum += val * (end - start + 1);
 tree[node].addMark += val;
 return;
 }

 pushDown(node, start, end);
 int mid = start + (end - start) / 2;
 update(2 * node, start, mid, l, r, val);
 update(2 * node + 1, mid + 1, end, l, r, val);
 tree[node].sum = tree[2 * node].sum + tree[2 * node + 1].sum;
}

```

```

}

// 区间查询
private long query(int node, int start, int end, int l, int r) {
 if (start > r || end < l) return 0;

 if (l <= start && end <= r) {
 return tree[node].sum;
 }

 pushDown(node, start, end);
 int mid = start + (end - start) / 2;
 long leftSum = query(2 * node, start, mid, l, r);
 long rightSum = query(2 * node + 1, mid + 1, end, l, r);
 return leftSum + rightSum;
}

// 公开方法
public void rangeAdd(int l, int r, long val) {
 update(1, 0, n - 1, l, r, val);
}

public long rangeQuery(int l, int r) {
 return query(1, 0, n - 1, l, r);
}
}
...

```

### ### C++实现

```

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

struct Node {
    long long sum;
    long long addMark;
    Node() : sum(0), addMark(0) {}
};

class HorribleQueries {
private:
    vector<Node> tree;

```

```
int n;
```

```
void pushDown(int node, int start, int end) {  
    if (tree[node].addMark != 0) {  
        int left = 2 * node;  
        int right = 2 * node + 1;  
        int mid = start + (end - start) / 2;  
  
        long long leftLen = mid - start + 1;  
        long long rightLen = end - mid;  
  
        tree[left].sum += tree[node].addMark * leftLen;  
        tree[right].sum += tree[node].addMark * rightLen;  
  
        tree[left].addMark += tree[node].addMark;  
        tree[right].addMark += tree[node].addMark;  
  
        tree[node].addMark = 0;  
    }  
}
```

```
void build(int node, int start, int end, const vector<long long>& arr) {  
    if (start == end) {  
        tree[node].sum = arr[start];  
        return;  
    }  
  
    int mid = start + (end - start) / 2;  
    build(2 * node, start, mid, arr);  
    build(2 * node + 1, mid + 1, end, arr);  
    tree[node].sum = tree[2 * node].sum + tree[2 * node + 1].sum;  
}
```

```
void update(int node, int start, int end, int l, int r, long long val) {  
    if (start > r || end < l) return;  
  
    if (l <= start && end <= r) {  
        tree[node].sum += val * (end - start + 1);  
        tree[node].addMark += val;  
        return;  
    }  
  
    pushDown(node, start, end);
```

```

        int mid = start + (end - start) / 2;
        update(2 * node, start, mid, l, r, val);
        update(2 * node + 1, mid + 1, end, l, r, val);
        tree[node].sum = tree[2 * node].sum + tree[2 * node + 1].sum;
    }

long long query(int node, int start, int end, int l, int r) {
    if (start > r || end < l) return 0;

    if (l <= start && end <= r) {
        return tree[node].sum;
    }

    pushDown(node, start, end);
    int mid = start + (end - start) / 2;
    long long leftSum = query(2 * node, start, mid, l, r);
    long long rightSum = query(2 * node + 1, mid + 1, end, l, r);
    return leftSum + rightSum;
}

public:
    HorribleQueries(const vector<long long>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        build(1, 0, n - 1, arr);
    }

    void rangeAdd(int l, int r, long long val) {
        update(1, 0, n - 1, l, r, val);
    }

    long long rangeQuery(int l, int r) {
        return query(1, 0, n - 1, l, r);
    }

    void solve() {
        int t;
        cin >> t;
        while (t--) {
            int n, q;
            cin >> n >> q;
            vector<long long> arr(n, 0);
            HorribleQueries hq(arr);
            while (q--) {

```

```

        int op, l, r;
        long long val;
        cin >> op >> l >> r;
        l--; r--; // 转换为 0-based 索引
        if (op == 0) {
            cin >> val;
            hq.rangeAdd(l, r, val);
        } else {
            cout << hq.rangeQuery(l, r) << endl;
        }
    }
}

};
```

```

### Python 实现

```
```python
```

```
class HorribleQueries:
```

```
    def __init__(self, arr):
```

```
        self.n = len(arr)
```

```
        self.tree = [{'sum': 0, 'addMark': 0} for _ in range(4 * self.n)]
```

```
        self.build(1, 0, self.n - 1, arr)
```

```
    def push_down(self, node, start, end):
```

```
        if self.tree[node]['addMark'] != 0:
```

```
            left = 2 * node
```

```
            right = 2 * node + 1
```

```
            mid = start + (end - start) // 2
```

```
            left_len = mid - start + 1
```

```
            right_len = end - mid
```

```
            # 更新左右子节点的和
```

```
            self.tree[left]['sum'] += self.tree[node]['addMark'] * left_len
```

```
            self.tree[right]['sum'] += self.tree[node]['addMark'] * right_len
```

```
            # 传递标记
```

```
            self.tree[left]['addMark'] += self.tree[node]['addMark']
```

```
            self.tree[right]['addMark'] += self.tree[node]['addMark']
```

```
            # 清除当前节点的标记
```

```
            self.tree[node]['addMark'] = 0
```

```

def build(self, node, start, end, arr):
    if start == end:
        self.tree[node]['sum'] = arr[start]
        return

    mid = start + (end - start) // 2
    self.build(2 * node, start, mid, arr)
    self.build(2 * node + 1, mid + 1, end, arr)
    self.tree[node]['sum'] = self.tree[2 * node]['sum'] + self.tree[2 * node + 1]['sum']

def update(self, node, start, end, l, r, val):
    if start > r or end < l:
        return

    if l <= start and end <= r:
        self.tree[node]['sum'] += val * (end - start + 1)
        self.tree[node]['addMark'] += val
        return

    self.push_down(node, start, end)
    mid = start + (end - start) // 2
    self.update(2 * node, start, mid, l, r, val)
    self.update(2 * node + 1, mid + 1, end, l, r, val)
    self.tree[node]['sum'] = self.tree[2 * node]['sum'] + self.tree[2 * node + 1]['sum']

def query(self, node, start, end, l, r):
    if start > r or end < l:
        return 0

    if l <= start and end <= r:
        return self.tree[node]['sum']

    self.push_down(node, start, end)
    mid = start + (end - start) // 2
    left_sum = self.query(2 * node, start, mid, l, r)
    right_sum = self.query(2 * node + 1, mid + 1, end, l, r)
    return left_sum + right_sum

def range_add(self, l, r, val):
    self.update(1, 0, self.n - 1, l, r, val)

def range_query(self, l, r):

```

```
return self.query(1, 0, self.n - 1, l, r)
```

总结

通过以上扩展题目，我们可以看到线段树在各种场景下的强大应用。从基础的区间查询与更新，到更复杂的动态开点、扫描线算法、斐波那契数列区间更新等，线段树都能高效处理。

在实现线段树时，需要特别注意以下几点：

1. ****节点信息设计****：根据问题需求设计合适的节点结构，包括需要维护的数据和懒惰标记类型
2. ****懒惰标记的正确传播****：确保标记能够正确地传给子节点，并在适当的时候清除
3. ****边界条件处理****：特别注意区间边界的处理，避免出现错误
4. ****性能优化****：对于大规模数据，可以考虑离散化、动态开点等技术
5. ****多语言实现差异****：不同语言在数据类型、递归深度等方面有不同的限制，需要适当调整实现方式

线段树作为一种通用的数据结构，在算法竞赛和实际工程中都有广泛的应用。掌握线段树的各种变体和优化技巧，对于解决复杂的区间操作问题非常有帮助。

通过这些题目的学习，我们不仅能够掌握线段树的基本用法，还能够深入理解其设计思想和优化技巧，为解决更复杂的算法问题打下坚实的基础。

在实际应用中，线段树通常与其他算法（如扫描线、离散化）结合使用，以解决更复杂的问题。因此，在学习线段树的同时，也应该关注相关算法和技术的学习，形成完整的知识体系。

1. ****POJ 3468 A Simple Problem with Integers**** 是线段树区间更新和查询的经典模板题
2. ****POJ 2528 Mayor's posters**** 展示了离散化在线段树中的重要应用
3. ****Codeforces 438D The Child and Sequence**** 是吉司机线段树的另一个经典应用，处理取模操作
4. ****SP0J GSS1 - Can you answer these queries I**** 展示了线段树在最大子段和问题中的应用
5. ****SP0J KGSS - Maximum Sum**** 展示了线段树在维护最大值和次大值中的应用
6. ****POJ 2777 - Count Color**** 展示了线段树在区间染色和颜色计数中的应用

线段树的其他重要应用场景还包括：

7. ****动态开点线段树****：适用于值域范围极大但实际使用点稀疏的情况，如 LeetCode 715. Range Module
8. ****扫描线算法结合线段树****：解决区间覆盖、面积计算等问题，如 LeetCode 218. The Skyline Problem
9. ****线段树优化 DP****：例如某些区间转移的动态规划问题
10. ****线段树分治****：处理离线的时间相关查询和更新操作

11. LeetCode 731. My Calendar II

题目描述

实现一个 MyCalendarTwo 类来存放你的日程安排。如果要添加的时间内不会导致三重预订时，则可以存储这个新的日程安排。

MyCalendarTwo 有一个 `book(int start, int end)` 方法。它意味着在 `start` 到 `end` 时间内增加一个日程安排，注意，这里的时间是半开区间，即 $[start, end)$ ，实数 x 的范围为， $start \leq x < end$ 。

当三个日程安排有一些时间上的重叠时（例如三个日程安排都在同一时间内），就会产生三重预订。

每次调用 `MyCalendarTwo.book` 方法时，如果可以将日程安排成功添加而不会导致三重预订，返回 `true`。否则，返回 `false` 并且不要添加该日程安排。

解题思路

这道题可以使用动态开点线段树来高效处理。由于时间范围可能很大（到 10^9 ），使用普通线段树会占用过多内存，而动态开点线段树只在需要时创建节点，更加高效。

Java 实现

```
```java
// LeetCode 731. 我的日程安排表 II - Java 实现
// 动态开点线段树解法
public class MyCalendarTwo {
 // 线段树节点
 private static class Node {
 int left; // 左边界
 int right; // 右边界
 int count; // 当前区间的覆盖次数
 int add; // 懒惰标记
 Node leftChild; // 左子节点
 Node rightChild; // 右子节点

 public Node(int left, int right) {
 this.left = left;
 this.right = right;
 }
 }

 private Node root; // 线段树的根节点
 private final int MAX_TIME = 1_000_000_000; // 根据题目约束设置最大值

 public MyCalendarTwo() {
 // 初始化根节点，覆盖题目中可能的时间范围
 root = new Node(0, MAX_TIME);
 }

 /**
 * 尝试预订一个新的区间
 */
}
```

```
* @param start 开始时间
* @param end 结束时间（不包含）
* @return 如果预订成功（不产生三重预订）返回 true，否则返回 false
*/
```

```
public boolean book(int start, int end) {
 // 首先查询区间[start, end-1]的最大覆盖次数
 if (query(root, start, end - 1) >= 2) {
 return false; // 已经有两个预订，不能再预订
 }
 // 然后对区间[start, end-1]进行加 1 操作
 update(root, start, end - 1, 1);
 return true;
}
```

```
/**
```

```
 * 查询区间[L, R]的最大覆盖次数
```

```
*/
```

```
private int query(Node node, int L, int R) {
 if (node == null) return 0;

 // 当前节点的区间完全包含在查询区间外
 if (node.right < L || node.left > R) {
 return 0;
 }

 // 当前节点的区间完全包含在查询区间内
 if (L <= node.left && node.right <= R) {
 return node.count;
 }

 // 否则需要递归查询左右子节点
 pushDown(node);
 int leftMax = query(node.leftChild, L, R);
 int rightMax = query(node.rightChild, L, R);
 return Math.max(leftMax, rightMax);
}
```

```
/**
```

```
 * 更新区间[L, R]，增加 val
```

```
*/
```

```
private void update(Node node, int L, int R, int val) {
 // 当前节点的区间完全包含在更新区间外
 if (node.right < L || node.left > R) {
```

```

 return;
 }

 // 当前节点的区间完全包含在更新区间内
 if (L <= node.left && node.right <= R) {
 node.count += val;
 node.add += val;
 return;
 }

 // 否则需要递归更新左右子节点
 pushDown(node);
 update(node.leftChild, L, R, val);
 update(node.rightChild, L, R, val);
 // 更新当前节点的最大覆盖次数
 node.count = Math.max(node.leftChild != null ? node.leftChild.count : 0,
 node.rightChild != null ? node.rightChild.count : 0);
}

/**
 * 下推懒惰标记，创建子节点（动态开点）
 */
private void pushDown(Node node) {
 int mid = node.left + (node.right - node.left) / 2;

 if (node.leftChild == null) {
 node.leftChild = new Node(node.left, mid);
 }
 if (node.rightChild == null) {
 node.rightChild = new Node(mid + 1, node.right);
 }

 if (node.add != 0) {
 node.leftChild.count += node.add;
 node.leftChild.add += node.add;
 node.rightChild.count += node.add;
 node.rightChild.add += node.add;
 node.add = 0; // 清除当前节点的懒惰标记
 }
}

public static void main(String[] args) {
 MyCalendarTwo calendar = new MyCalendarTwo();
}

```

```

 System.out.println(calendar.book(10, 20)); // 预期输出: true
 System.out.println(calendar.book(50, 60)); // 预期输出: true
 System.out.println(calendar.book(10, 40)); // 预期输出: true
 System.out.println(calendar.book(5, 15)); // 预期输出: false
 System.out.println(calendar.book(5, 10)); // 预期输出: true
 System.out.println(calendar.book(25, 55)); // 预期输出: true
 }
}
```

```

C++实现

```

```cpp
// LeetCode 731. My Calendar II - C++实现
#include <iostream>
using namespace std;

class MyCalendarTwo {
private:
 // 线段树节点
 struct Node {
 long long left, right; // 使用 long long 避免溢出
 int count; // 当前区间的最大覆盖次数
 int add; // 懒惰标记
 Node *leftChild, *rightChild;

 Node(long long l, long long r) : left(l), right(r), count(0), add(0),
 leftChild(nullptr), rightChild(nullptr) {}
 };

 Node* root; // 线段树的根节点
 const long long MAX_TIME = 1e9; // 根据题目约束设置最大值

 // 下推懒惰标记, 创建子节点 (动态开点)
 void pushDown(Node* node) {
 long long mid = node->left + (node->right - node->left) / 2;

 if (!node->leftChild) {
 node->leftChild = new Node(node->left, mid);
 }
 if (!node->rightChild) {
 node->rightChild = new Node(mid + 1, node->right);
 }
 }
}
```

```

```

    if (node->add != 0) {
        // 下传标记到左右子节点
        node->leftChild->count += node->add;
        node->leftChild->add += node->add;
        node->rightChild->count += node->add;
        node->rightChild->add += node->add;
        node->add = 0; // 清除当前节点的懒惰标记
    }
}

// 更新区间[L, R], 增加 val
void update(Node* node, long long L, long long R, int val) {
    // 当前节点的区间完全包含在更新区间外
    if (node->right < L || node->left > R) {
        return;
    }

    // 当前节点的区间完全包含在更新区间内
    if (L <= node->left && node->right <= R) {
        node->count += val;
        node->add += val;
        return;
    }

    // 否则需要递归更新左右子节点
    pushDown(node);
    update(node->leftChild, L, R, val);
    update(node->rightChild, L, R, val);
    // 更新当前节点的最大覆盖次数
    node->count = max(node->leftChild->count, node->rightChild->count);
}

// 查询区间[L, R]的最大覆盖次数
int query(Node* node, long long L, long long R) {
    if (!node) return 0;

    // 当前节点的区间完全包含在查询区间外
    if (node->right < L || node->left > R) {
        return 0;
    }

    // 当前节点的区间完全包含在查询区间内
    if (L <= node->left && node->right <= R) {

```

```

        return node->count;
    }

    // 否则需要递归查询左右子节点
    pushDown(node);
    int leftMax = query(node->leftChild, L, R);
    int rightMax = query(node->rightChild, L, R);
    return max(leftMax, rightMax);
}

// 释放内存，避免内存泄漏
void clearTree(Node* node) {
    if (!node) return;
    clearTree(node->leftChild);
    clearTree(node->rightChild);
    delete node;
}

public:
    MyCalendarTwo() {
        // 初始化根节点，覆盖题目中可能的时间范围
        root = new Node(0, MAX_TIME);
    }

    ~MyCalendarTwo() {
        // 析构函数中释放内存
        clearTree(root);
    }

    bool book(int start, int end) {
        // 将 end-1，因为题目中的时间是半开区间 [start, end)
        long long L = start;
        long long R = end - 1;

        // 首先查询区间[start, end-1]的最大覆盖次数
        if (query(root, L, R) >= 2) {
            return false; // 已经有两个预订，不能再预订
        }

        // 然后对区间[start, end-1]进行加 1 操作
        update(root, L, R, 1);
        return true;
    }
}

```

```
};
```

```
int main() {  
    MyCalendarTwo calendar;  
    cout << (calendar.book(10, 20) ? "true" : "false") << endl; // 输出: true  
    cout << (calendar.book(50, 60) ? "true" : "false") << endl; // 输出: true  
    cout << (calendar.book(10, 40) ? "true" : "false") << endl; // 输出: true  
    cout << (calendar.book(5, 15) ? "true" : "false") << endl; // 输出: false  
    cout << (calendar.book(5, 10) ? "true" : "false") << endl; // 输出: true  
    cout << (calendar.book(25, 55) ? "true" : "false") << endl; // 输出: true  
    return 0;  
}
```

```
### Python 实现
```

```
```python
```

```
LeetCode 731. My Calendar II - Python 实现
```

```
class MyCalendarTwo:
```

```
 def __init__(self):
```

```
 # 初始化根节点，覆盖题目中可能的时间范围
```

```
 self.root = {'left': 0, 'right': 10**9, 'count': 0, 'add': 0,
 'leftChild': None, 'rightChild': None}
```

```
 def book(self, start: int, end: int) -> bool:
```

```
 """
```

```
 尝试预订一个新的区间
```

```
 :param start: 开始时间
```

```
 :param end: 结束时间（不包含）
```

```
 :return: 如果预订成功（不产生三重预订）返回 True，否则返回 False
```

```
 """
```

```
 # 由于时间是半开区间 [start, end)，我们查询 [start, end-1]
```

```
 if self._query(self.root, start, end - 1) >= 2:
```

```
 return False # 已经有两个预订，不能再预订
```

```
 # 更新区间，增加预订次数
```

```
 self._update(self.root, start, end - 1, 1)
```

```
 return True
```

```
 def _push_down(self, node):
```

```
 """
```

```
 下推懒惰标记，创建子节点（动态开点）
```

```
 """
```

```
 mid = node['left'] + (node['right'] - node['left']) // 2
```

```

如果子节点不存在，则创建
if not node['leftChild']:
 node['leftChild'] = {'left': node['left'], 'right': mid,
 'count': 0, 'add': 0,
 'leftChild': None, 'rightChild': None}
if not node['rightChild']:
 node['rightChild'] = {'left': mid + 1, 'right': node['right'],
 'count': 0, 'add': 0,
 'leftChild': None, 'rightChild': None}

```

```

如果有懒惰标记需要下传
if node['add'] != 0:
 # 下传标记到左右子节点
 node['leftChild']['count'] += node['add']
 node['leftChild']['add'] += node['add']
 node['rightChild']['count'] += node['add']
 node['rightChild']['add'] += node['add']
 # 清除当前节点的懒惰标记
 node['add'] = 0

```

```

def _update(self, node, L, R, val):

```

```

 """

```

```

 更新区间[L, R]，增加 val

```

```

 """

```

```

当前节点的区间完全包含在更新区间外

```

```

if node['right'] < L or node['left'] > R:

```

```

 return

```

```

当前节点的区间完全包含在更新区间内

```

```

if L <= node['left'] and node['right'] <= R:

```

```

 node['count'] += val

```

```

 node['add'] += val

```

```

 return

```

```

否则需要递归更新左右子节点

```

```

self._push_down(node)

```

```

self._update(node['leftChild'], L, R, val)

```

```

self._update(node['rightChild'], L, R, val)

```

```

更新当前节点的最大覆盖次数

```

```

node['count'] = max(node['leftChild']['count'], node['rightChild']['count'])

```

```

def _query(self, node, L, R):

```

```

"""
查询区间[L, R]的最大覆盖次数
"""

if not node:
 return 0

当前节点的区间完全包含在查询区间外
if node['right'] < L or node['left'] > R:
 return 0

当前节点的区间完全包含在查询区间内
if L <= node['left'] and node['right'] <= R:
 return node['count']

否则需要递归查询左右子节点
self._push_down(node)
left_max = self._query(node['leftChild'], L, R)
right_max = self._query(node['rightChild'], L, R)
return max(left_max, right_max)

测试代码
if __name__ == "__main__":
 calendar = MyCalendarTwo()
 print(calendar.book(10, 20)) # 输出: True
 print(calendar.book(50, 60)) # 输出: True
 print(calendar.book(10, 40)) # 输出: True
 print(calendar.book(5, 15)) # 输出: False
 print(calendar.book(5, 10)) # 输出: True
 print(calendar.book(25, 55)) # 输出: True

```

## ## 12. LeetCode 699. Falling Squares

### ### 题目描述

在无限长的数轴（即  $x$  轴）上，我们根据给定的顺序放置对应的正方形方块。

第  $i$  个掉落的方块（ $\text{positions}[i] = (\text{left}, \text{side\_length})$ ）是正方形，其左下角在  $(\text{left}, 0)$ ，右上角在  $(\text{left} + \text{side\_length}, \text{side\_length})$ 。

当方块掉落并停在某个高度时，它会立即粘住下面的任何方块，或者粘在地面上（如果没有方块在它下面的话）。方块一旦粘住，就不会再移动。

返回一个堆叠高度列表  $\text{ans}$ ，其中  $\text{ans}[i]$  表示在第  $i$  个方块掉落之后，最高堆叠方块的高度。

#### #### 解题思路

这道题可以使用动态开点线段树来维护每个区间的最大高度。当放置一个新方块时，我们需要：

1. 查询该方块覆盖区间的当前最大高度
2. 计算新方块的顶部高度（当前最大高度 + 方块边长）
3. 更新该区间的高度为新方块的顶部高度
4. 记录全局最大高度

#### #### Java 实现

```
```java
// LeetCode 699. Falling Squares - Java 实现
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class FallingSquares {
    // 线段树节点
    static class Node {
        int max;        // 当前区间的最大高度
        int lazy;        // 懒惰标记，表示待下传的更新值
        Node left;       // 左子节点
        Node right;      // 右子节点
    }

    private Node root;
    private final int MAX_RANGE = 1_000_000_000; // 根据题目约束设置最大值范围

    public FallingSquares() {
        root = new Node();
    }

    /**
     * 处理掉落的方块，返回每次掉落后的最大高度列表
     */
    public List<Integer> fallingSquares(int[][] positions) {
        List<Integer> result = new ArrayList<>();
        int maxHeight = 0;

        for (int[] pos : positions) {
            int left = pos[0];
            int sideLength = pos[1];
            int right = left + sideLength - 1;
```

```

        // 查询当前区间的最大高度
        int currentMax = query(root, 0, MAX_RANGE, left, right);

        // 计算新方块的顶部高度
        int newHeight = currentMax + sideLength;

        // 更新区间高度
        update(root, 0, MAX_RANGE, left, right, newHeight);

        // 更新全局最大高度
        maxHeight = Math.max(maxHeight, newHeight);
        result.add(maxHeight);
    }

    return result;
}

/**
 * 更新区间[L, R]的最大高度为 val
 * 这里使用的是区间覆盖操作
 */
private void update(Node node, int l, int r, int L, int R, int val) {
    // 确保节点存在
    if (node == null) {
        return;
    }

    // 当前区间完全包含在更新区间内
    if (L <= l && r <= R) {
        // 使用区间覆盖，只保留最大值
        node.max = Math.max(node.max, val);
        node.lazy = Math.max(node.lazy, val);
        return;
    }

    // 下推懒惰标记
    pushDown(node);

    int mid = l + (r - l) / 2;
    if (L <= mid) {
        // 确保左子节点存在
        if (node.left == null) {
            node.left = new Node();

```

```

    }
    update(node.left, l, mid, L, R, val);
}
if (R > mid) {
    // 确保右子节点存在
    if (node.right == null) {
        node.right = new Node();
    }
    update(node.right, mid + 1, r, L, R, val);
}

// 更新当前节点的最大值
int leftMax = (node.left != null) ? node.left.max : 0;
int rightMax = (node.right != null) ? node.right.max : 0;
node.max = Math.max(leftMax, rightMax);
}

/**
 * 查询区间[L, R]的最大高度
 */
private int query(Node node, int l, int r, int L, int R) {
    if (node == null) {
        return 0;
    }

    // 当前区间完全包含在查询区间内
    if (L <= l && r <= R) {
        return node.max;
    }

    // 下推懒惰标记
    pushDown(node);

    int mid = l + (r - l) / 2;
    int maxVal = 0;

    if (L <= mid && node.left != null) {
        maxVal = Math.max(maxVal, query(node.left, l, mid, L, R));
    }

    if (R > mid && node.right != null) {
        maxVal = Math.max(maxVal, query(node.right, mid + 1, r, L, R));
    }
}

```

```

        return maxVal;
    }

    /**
     * 下推懒惰标记
     */
    private void pushDown(Node node) {
        if (node.lazy != 0) {
            // 确保左右子节点存在
            if (node.left == null) {
                node.left = new Node();
            }
            if (node.right == null) {
                node.right = new Node();
            }

            // 更新子节点的 max 和 lazy 值
            node.left.max = Math.max(node.left.max, node.lazy);
            node.left.lazy = Math.max(node.left.lazy, node.lazy);
            node.right.max = Math.max(node.right.max, node.lazy);
            node.right.lazy = Math.max(node.right.lazy, node.lazy);

            // 清除当前节点的懒惰标记
            node.lazy = 0;
        }
    }

    public static void main(String[] args) {
        FallingSquares fs = new FallingSquares();
        int[][] positions1 = {{1, 2}, {2, 3}, {6, 1}};
        System.out.println(Arrays.toString(fs.fallingSquares(positions1).toArray())); // 输出:
[2, 5, 5]

        int[][] positions2 = {{100, 100}, {200, 100}};
        System.out.println(Arrays.toString(fs.fallingSquares(positions2).toArray())); // 输出:
[100, 100]
    }
}

```

C++实现

```cpp

// LeetCode 699. Falling Squares - C++实现

```

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

class FallingSquares {
private:
 // 线段树节点
 struct Node {
 int max_height; // 当前区间的最大高度
 int lazy; // 懒惰标记，表示待下传的更新值
 Node* left; // 左子节点
 Node* right; // 右子节点

 Node() : max_height(0), lazy(0), left(nullptr), right(nullptr) {}
 };

 Node* root;
 const int MAX_RANGE = 1e9; // 根据题目约束设置最大值范围

 /**
 * 下推懒惰标记
 */
 void pushDown(Node* node) {
 if (node->lazy != 0) {
 // 确保左右子节点存在
 if (!node->left) {
 node->left = new Node();
 }
 if (!node->right) {
 node->right = new Node();
 }

 // 更新子节点的 max_height 和 lazy 值
 node->left->max_height = max(node->left->max_height, node->lazy);
 node->left->lazy = max(node->left->lazy, node->lazy);
 node->right->max_height = max(node->right->max_height, node->lazy);
 node->right->lazy = max(node->right->lazy, node->lazy);

 // 清除当前节点的懒惰标记
 node->lazy = 0;
 }
 }
}

```

```

/**
 * 更新区间[L, R]的最大高度为 val
 */
void update(Node* node, int l, int r, int L, int R, int val) {
 // 当前区间完全包含在更新区间内
 if (L <= l && r <= R) {
 // 使用区间覆盖, 只保留最大值
 node->max_height = max(node->max_height, val);
 node->lazy = max(node->lazy, val);
 return;
 }

 // 下推懒惰标记
 pushDown(node);

 int mid = l + (r - l) / 2;
 if (L <= mid) {
 // 确保左子节点存在
 if (!node->left) {
 node->left = new Node();
 }
 update(node->left, l, mid, L, R, val);
 }
 if (R > mid) {
 // 确保右子节点存在
 if (!node->right) {
 node->right = new Node();
 }
 update(node->right, mid + 1, r, L, R, val);
 }

 // 更新当前节点的最大值
 int left_max = node->left ? node->left->max_height : 0;
 int right_max = node->right ? node->right->max_height : 0;
 node->max_height = max(left_max, right_max);
}

/**
 * 查询区间[L, R]的最大高度
 */
int query(Node* node, int l, int r, int L, int R) {
 if (!node) {

```

```

 return 0;
 }

 // 当前区间完全包含在查询区间内
 if (L <= l && r <= R) {
 return node->max_height;
 }

 // 下推懒惰标记
 pushDown(node);

 int mid = l + (r - l) / 2;
 int max_val = 0;

 if (L <= mid && node->left) {
 max_val = max(max_val, query(node->left, l, mid, L, R));
 }
 if (R > mid && node->right) {
 max_val = max(max_val, query(node->right, mid + 1, r, L, R));
 }

 return max_val;
}

/**
 * 释放内存，避免内存泄漏
 */
void clearTree(Node* node) {
 if (!node) return;
 clearTree(node->left);
 clearTree(node->right);
 delete node;
}

public:
 FallingSquares() {
 root = new Node();
 }

 ~FallingSquares() {
 clearTree(root);
 }

```

```

vector<int> fallingSquares(vector<vector<int>>& positions) {
 vector<int> result;
 int max_height = 0;

 for (auto& pos : positions) {
 int left = pos[0];
 int side_length = pos[1];
 int right = left + side_length - 1;

 // 查询当前区间的最大高度
 int current_max = query(root, 0, MAX_RANGE, left, right);

 // 计算新方块的顶部高度
 int new_height = current_max + side_length;

 // 更新区间高度
 update(root, 0, MAX_RANGE, left, right, new_height);

 // 更新全局最大高度
 max_height = max(max_height, new_height);
 result.push_back(max_height);
 }

 return result;
}

};

```

```

int main() {
 FallingSquares fs;

 // 测试用例 1
 vector<vector<int>> positions1 = {{1, 2}, {2, 3}, {6, 1}};
 vector<int> result1 = fs.fallingSquares(positions1);
 cout << "[";
 for (int i = 0; i < result1.size(); i++) {
 cout << result1[i];
 if (i < result1.size() - 1) cout << ", ";
 }
 cout << "]" << endl; // 输出: [2, 5, 5]

 // 测试用例 2
 vector<vector<int>> positions2 = {{100, 100}, {200, 100}};
 vector<int> result2 = fs.fallingSquares(positions2);
}

```

```

cout << "[";
for (int i = 0; i < result2.size(); i++) {
 cout << result2[i];
 if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl; // 输出: [100, 100]

return 0;
}
```

```

Python 实现

```

```python
LeetCode 699. Falling Squares - Python 实现
class FallingSquares:
 def __init__(self):
 # 初始化根节点，使用字典表示
 self.root = {'max_height': 0, 'lazy': 0, 'left': None, 'right': None}
 self.MAX_RANGE = 10**9 # 根据题目约束设置最大值范围

 def fallingSquares(self, positions):
 """
 处理掉落的方块，返回每次掉落后的最大高度列表
 :param positions: 方块的位置列表，每个元素为 [left, side_length]
 :return: 每次掉落后的最大高度列表
 """
 result = []
 max_height = 0

 for pos in positions:
 left = pos[0]
 side_length = pos[1]
 right = left + side_length - 1

 # 查询当前区间的最大高度
 current_max = self._query(self.root, 0, self.MAX_RANGE, left, right)

 # 计算新方块的顶部高度
 new_height = current_max + side_length

 # 更新区间高度
 self._update(self.root, 0, self.MAX_RANGE, left, right, new_height)

```

```

 # 更新全局最大高度
 max_height = max(max_height, new_height)
 result.append(max_height)

 return result

def _push_down(self, node):
 """
 下推懒惰标记
 """
 if node['lazy'] != 0:
 # 确保左右子节点存在
 if not node['left']:
 node['left'] = {'max_height': 0, 'lazy': 0, 'left': None, 'right': None}
 if not node['right']:
 node['right'] = {'max_height': 0, 'lazy': 0, 'left': None, 'right': None}

 # 更新子节点的 max_height 和 lazy 值
 node['left']['max_height'] = max(node['left']['max_height'], node['lazy'])
 node['left']['lazy'] = max(node['left']['lazy'], node['lazy'])
 node['right']['max_height'] = max(node['right']['max_height'], node['lazy'])
 node['right']['lazy'] = max(node['right']['lazy'], node['lazy'])

 # 清除当前节点的懒惰标记
 node['lazy'] = 0

def _update(self, node, l, r, L, R, val):
 """
 更新区间[L, R]的最大高度为 val
 """
 # 当前区间完全包含在更新区间内
 if L <= l and r <= R:
 # 使用区间覆盖，只保留最大值
 node['max_height'] = max(node['max_height'], val)
 node['lazy'] = max(node['lazy'], val)
 return

 # 下推懒惰标记
 self._push_down(node)

 mid = l + (r - l) // 2
 if L <= mid:
 # 确保左子节点存在

```

```

 if not node['left']:
 node['left'] = {'max_height': 0, 'lazy': 0, 'left': None, 'right': None}
 self._update(node['left'], l, mid, L, R, val)
 if R > mid:
 # 确保右子节点存在
 if not node['right']:
 node['right'] = {'max_height': 0, 'lazy': 0, 'left': None, 'right': None}
 self._update(node['right'], mid + 1, r, L, R, val)

 # 更新当前节点的最大值
 left_max = node['left']['max_height'] if node['left'] else 0
 right_max = node['right']['max_height'] if node['right'] else 0
 node['max_height'] = max(left_max, right_max)

def _query(self, node, l, r, L, R):
 """
 查询区间[L, R]的最大高度
 """
 if not node:
 return 0

 # 当前区间完全包含在查询区间内
 if L <= l and r <= R:
 return node['max_height']

 # 下推懒惰标记
 self._push_down(node)

 mid = l + (r - l) // 2
 max_val = 0

 if L <= mid and node['left']:
 max_val = max(max_val, self._query(node['left'], l, mid, L, R))
 if R > mid and node['right']:
 max_val = max(max_val, self._query(node['right'], mid + 1, r, L, R))

 return max_val

测试代码
if __name__ == "__main__":
 fs = FallingSquares()

 # 测试用例 1

```

```

positions1 = [[1, 2], [2, 3], [6, 1]]
print(fs.fallingSquares(positions1)) # 输出: [2, 5, 5]

测试用例 2
positions2 = [[100, 100], [200, 100]]
print(fs.fallingSquares(positions2)) # 输出: [100, 100]
...

```

这些题目帮助我们更深入地理解线段树在实际问题中的灵活应用，包括：

- 懒惰标记的使用
- 离散化技术
- 复杂操作的处理
- 多种查询类型的组合
- 位运算优化

对于线段树的学习，建议关注以下几个方面：

1. **基础操作的底层细节**：理解 pushUp、pushDown 等核心操作的实现原理
2. **懒惰标记的正确使用**：掌握不同类型操作的懒惰标记设计和下传策略
3. **边界条件的处理**：注意区间的开闭性质、索引的起始位置等细节
4. **性能优化**：在大规模数据情况下，考虑常数优化、内存优化等技术
5. **多维度思考**：从算法设计、工程实现、语言特性等多个角度分析问题

线段树作为一种通用的区间处理工具，在算法竞赛和实际工程中都有广泛的应用。通过深入学习这一数据结构，可以显著提升解决复杂算法问题的能力。

=====

文件：PROBLEMS.md

=====

# 线段树相关题目汇总

## 经典题目列表

### 1. HDU 题目

#### HDU 5306 Gorgeous Sequence

- **题目链接**： <http://acm.hdu.edu.cn/showproblem.php?pid=5306>
- **题目描述**：维护一个序列，支持区间取 min 操作，查询区间最大值和区间和
- **算法**：吉司机线段树
- **难度**：困难
- **相关文件**：

[Code03\_SegmentTreeSetminQueryMaxSum2.java] (Code03\_SegmentTreeSetminQueryMaxSum2.java)

#### #### HDU 1166 敌兵布阵

- \*\*题目链接\*\*： <http://acm.hdu.edu.cn/showproblem.php?pid=1166>
- \*\*题目描述\*\*： 单点更新，区间求和
- \*\*算法\*\*： 基础线段树
- \*\*难度\*\*： 简单

#### #### HDU 1754 I Hate It

- \*\*题目链接\*\*： <http://acm.hdu.edu.cn/showproblem.php?pid=1754>
- \*\*题目描述\*\*： 单点更新，区间最值查询
- \*\*算法\*\*： 基础线段树
- \*\*难度\*\*： 简单

#### #### HDU 1698 Just a Hook

- \*\*题目链接\*\*： <http://acm.hdu.edu.cn/showproblem.php?pid=1698>
- \*\*题目描述\*\*： 区间更新，区间求和
- \*\*算法\*\*： 线段树 + 懒惰标记
- \*\*难度\*\*： 中等

#### #### HDU 1542 Atlantis

- \*\*题目链接\*\*： <http://acm.hdu.edu.cn/showproblem.php?pid=1542>
- \*\*题目描述\*\*： 矩形面积并
- \*\*算法\*\*： 线段树 + 扫描线 + 离散化
- \*\*难度\*\*： 困难

#### #### HDU 438D The Child and Sequence

- \*\*题目链接\*\*： <http://acm.hdu.edu.cn/showproblem.php?pid=438D>
- \*\*题目描述\*\*： 区间取模，区间最大值，区间和
- \*\*算法\*\*： 吉司机线段树
- \*\*难度\*\*： 困难

### ### 2. POJ 题目

#### #### POJ 3468 A Simple Problem with Integers

- \*\*题目链接\*\*： <http://poj.org/problem?id=3468>
- \*\*题目描述\*\*： 区间加法，区间求和
- \*\*算法\*\*： 线段树 + 懒惰标记
- \*\*难度\*\*： 中等

#### #### POJ 2528 Mayor's posters

- \*\*题目链接\*\*： <http://poj.org/problem?id=2528>
- \*\*题目描述\*\*： 区间染色问题，求可见海报数
- \*\*算法\*\*： 线段树 + 离散化

- **\*\*难度\*\***: 中等

#### #### POJ 2777 Count Color

- **\*\*题目链接\*\***: <http://poj.org/problem?id=2777>

- **\*\*题目描述\*\***: 区间染色, 查询区间颜色数

- **\*\*算法\*\***: 线段树 + 位运算

- **\*\*难度\*\***: 中等

- **\*\*相关文件\*\***: [Code08\_CountColor.java] (Code08\_CountColor.java),

[Code08\_CountColor.cpp] (Code08\_CountColor.cpp), [Code08\_CountColor.py] (Code08\_CountColor.py)

#### #### POJ 1177 Picture

- **\*\*题目链接\*\***: <http://poj.org/problem?id=1177>

- **\*\*题目描述\*\***: 矩形周长并

- **\*\*算法\*\***: 线段树 + 扫描线

- **\*\*难度\*\***: 困难

#### #### POJ 3667 Hotel

- **\*\*题目链接\*\***: <http://poj.org/problem?id=3667>

- **\*\*题目描述\*\***: 区间分配问题, 支持区间占用和释放

- **\*\*算法\*\***: 线段树 + 区间合并

- **\*\*难度\*\***: 困难

### ### 3. 洛谷题目

#### #### P3372 【模板】线段树 1

- **\*\*题目链接\*\***: <https://www.luogu.com.cn/problem/P3372>

- **\*\*题目描述\*\***: 区间加法, 区间求和

- **\*\*算法\*\***: 线段树 + 懒惰标记

- **\*\*难度\*\***: 中等

#### #### P3373 【模板】线段树 2

- **\*\*题目链接\*\***: <https://www.luogu.com.cn/problem/P3373>

- **\*\*题目描述\*\***: 区间乘法和加法, 区间求和

- **\*\*算法\*\***: 线段树 + 双懒惰标记

- **\*\*难度\*\***: 中等

#### #### P6242 【模板】线段树 3

- **\*\*题目链接\*\***: <https://www.luogu.com.cn/problem/P6242>

- **\*\*题目描述\*\***: 区间加法、区间取 min、区间求和、区间最值、区间历史最值

- **\*\*算法\*\***: 吉司机线段树 + 历史信息维护

- **\*\*难度\*\***: 困难

- **\*\*相关文件\*\***: [Code05\_MaximumMinimumHistory.java] (Code05\_MaximumMinimumHistory.java)

#### #### P5490 【模板】扫描线

- \*\*题目链接\*\*： <https://www.luogu.com.cn/problem/P5490>
- \*\*题目描述\*\*： 矩形面积并
- \*\*算法\*\*： 线段树 + 扫描线 + 离散化
- \*\*难度\*\*： 困难

#### ### 4. LeetCode 题目

##### #### 307. Range Sum Query - Mutable

- \*\*题目链接\*\*： <https://leetcode.cn/problems/range-sum-query-mutable/>
- \*\*题目描述\*\*： 数组可变时的区间求和
- \*\*算法\*\*： 线段树
- \*\*难度\*\*： 中等

##### #### 715. Range Module

- \*\*题目链接\*\*： <https://leetcode.cn/problems/range-module/>
- \*\*题目描述\*\*： 范围模块，支持添加、查询和删除区间
- \*\*算法\*\*： 动态开点线段树
- \*\*难度\*\*： 困难
- \*\*相关文件\*\*： [Code01\_DynamicSegmentTree.java] (Code01\_DynamicSegmentTree.java), [Code02\_CountIntervals.java] (Code02\_CountIntervals.java)

##### #### 218. The Skyline Problem

- \*\*题目链接\*\*： <https://leetcode.cn/problems/the-skyline-problem/>
- \*\*题目描述\*\*： 天际线问题
- \*\*算法\*\*： 线段树 + 扫描线 或 multiset
- \*\*难度\*\*： 困难

##### #### 699. Falling Squares

- \*\*题目链接\*\*： <https://leetcode.cn/problems/falling-squares/>
- \*\*题目描述\*\*： 掉落的方块
- \*\*算法\*\*： 线段树 + 懒惰标记
- \*\*难度\*\*： 困难

##### #### 731. My Calendar II

- \*\*题目链接\*\*： <https://leetcode.cn/problems/my-calendar-ii/>
- \*\*题目描述\*\*： 我的日程安排表 II（检测三重预订）
- \*\*算法\*\*： 动态开点线段树
- \*\*难度\*\*： 中等

##### #### 732. My Calendar III

- \*\*题目链接\*\*： <https://leetcode.cn/problems/my-calendar-iii/>
- \*\*题目描述\*\*： 我的日程安排表 III（检测 K 重预订）

- **\*\*算法\*\***: 动态开点线段树
- **\*\*难度\*\***: 困难

### ### 5. Codeforces 题目

#### #### 52C Circular RMQ

- **\*\*题目链接\*\***: <https://codeforces.com/problemset/problem/52/C>
- **\*\*题目描述\*\***: 循环区间最小值查询
- **\*\*算法\*\***: 线段树 + 懒惰标记
- **\*\*难度\*\***: 中等

#### #### 380C Sereja and Brackets

- **\*\*题目链接\*\***: <https://codeforces.com/problemset/problem/380/C>
- **\*\*题目描述\*\***: 括号匹配查询
- **\*\*算法\*\***: 线段树
- **\*\*难度\*\***: 中等

#### #### 446C DZY Loves Fibonacci Numbers

- **\*\*题目链接\*\***: <https://codeforces.com/problemset/problem/446/C>
- **\*\*题目描述\*\***: 斐波那契数列区间加法
- **\*\*算法\*\***: 线段树 + 斐波那契性质
- **\*\*难度\*\***: 困难

#### #### 438D The Child and Sequence

- **\*\*题目链接\*\***: <https://codeforces.com/problemset/problem/438/D>
- **\*\*题目描述\*\***: 区间取模, 区间最大值, 区间和
- **\*\*算法\*\***: 吉司机线段树
- **\*\*难度\*\***: 困难

### ### 6. SPOJ 题目

#### #### GSS1 Can you answer these queries I

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/GSS1/>
- **\*\*题目描述\*\***: 最大子段和查询
- **\*\*算法\*\***: 线段树
- **\*\*难度\*\***: 中等
- **\*\*相关文件\*\***: [Code06\_MaximumSubarraySum.java] (Code06\_MaximumSubarraySum.java),  
[Code06\_MaximumSubarraySum.cpp] (Code06\_MaximumSubarraySum.cpp),  
[Code06\_MaximumSubarraySum.py] (Code06\_MaximumSubarraySum.py)

#### #### GSS2 Can you answer these queries II

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/GSS2/>
- **\*\*题目描述\*\***: 历史最大子段和查询

- **\*\*算法\*\***: 线段树 + 历史信息维护
- **\*\*难度\*\***: 困难

#### #### GSS3 Can you answer these queries III

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/GSS3/>
- **\*\*题目描述\*\***: 最大子段和查询（支持单点更新）
- **\*\*算法\*\***: 线段树
- **\*\*难度\*\***: 中等

#### #### GSS4 Can you answer these queries IV

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/GSS4/>
- **\*\*题目描述\*\***: 区间开方，区间求和
- **\*\*算法\*\***: 线段树 + 区间操作
- **\*\*难度\*\***: 中等

#### #### GSS5 Can you answer these queries V

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/GSS5/>
- **\*\*题目描述\*\***: 最大子段和查询（支持任意区间）
- **\*\*算法\*\***: 线段树
- **\*\*难度\*\***: 中等

#### #### GSS6 Can you answer these queries VI

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/GSS6/>
- **\*\*题目描述\*\***: 支持插入、删除、修改、查询最大子段和
- **\*\*算法\*\***: 平衡树 + 线段树
- **\*\*难度\*\***: 困难

#### #### GSS7 Can you answer these queries VII

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/GSS7/>
- **\*\*题目描述\*\***: 树上路径最大子段和查询
- **\*\*算法\*\***: 树链剖分 + 线段树
- **\*\*难度\*\***: 困难

#### #### HORRIBLE Horrible Queries

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/HORRIBLE/>
- **\*\*题目描述\*\***: 区间加法，区间求和
- **\*\*算法\*\***: 线段树 + 懒惰标记
- **\*\*难度\*\***: 中等

#### #### KGSS Maximum Sum

- **\*\*题目链接\*\***: <https://www.spoj.com/problems/KGSS/>
- **\*\*题目描述\*\***: 查询区间内两个最大值的和
- **\*\*算法\*\***: 线段树

- **\*\*难度\*\***: 中等
- **\*\*相关文件\*\***: [Code07\_MaximumTwoValuesSum.java] (Code07\_MaximumTwoValuesSum.java),  
[Code07\_MaximumTwoValuesSum.cpp] (Code07\_MaximumTwoValuesSum.cpp),  
[Code07\_MaximumTwoValuesSum.py] (Code07\_MaximumTwoValuesSum.py)

## ## 题目分类

### #### 按操作类型分类

#### ##### 1. 基础操作

- 单点更新 + 区间查询
  - HDU 1166 敌兵布阵
  - HDU 1754 I Hate It
- 区间更新 + 区间查询
  - POJ 3468 A Simple Problem with Integers
  - HDU 1698 Just a Hook
  - 洛谷 P3372 **【模板】**线段树 1

#### ##### 2. 高级操作

- 区间最值操作
  - HDU 5306 Gorgeous Sequence
  - Codeforces 438D The Child and Sequence
- 区间历史信息维护
  - 洛谷 P6242 **【模板】**线段树 3
  - LeetCode 715. Range Module
  - SPOJ GSS2 Can you answer these queries II
- 多维线段树
  - POJ 1177 Picture
  - HDU 1542 Atlantis

#### ##### 3. 特殊应用

- 离散化 + 线段树
  - POJ 2528 Mayor's posters
  - POJ 2777 Count Color
- 扫描线 + 线段树
  - POJ 1177 Picture
  - HDU 1542 Atlantis
  - LeetCode 218. The Skyline Problem
  - 洛谷 P5490 **【模板】**扫描线
- 线段树 + 其他算法
  - Codeforces 446C DZY Loves Fibonacci Numbers (斐波那契性质)
  - SPOJ GSS1/GSS3 (最大子段和)
  - SPOJ KGSS (两个最大值之和)

- SPOJ GSS4 (区间开方)
- SPOJ GSS6/GSS7 (平衡树/树链剖分)

### ### 按难度分类

#### ##### 简单 (适合入门)

1. HDU 1166 敌兵布阵
2. HDU 1754 I Hate It
3. 洛谷 P3372 【模板】线段树 1

#### ##### 中等 (掌握基础后练习)

1. POJ 3468 A Simple Problem with Integers
2. HDU 1698 Just a Hook
3. POJ 2528 Mayor's posters
4. POJ 2777 Count Color
5. 洛谷 P3373 【模板】线段树 2
6. LeetCode 307. Range Sum Query - Mutable
7. Codeforces 52C Circular RMQ
8. SPOJ HORRIBLE Horrible Queries
9. SPOJ GSS1 Can you answer these queries I
10. SPOJ KGSS Maximum Sum
11. SPOJ GSS3 Can you answer these queries III
12. SPOJ GSS4 Can you answer these queries IV
13. SPOJ GSS5 Can you answer these queries V

#### ##### 困难 (高阶应用)

1. HDU 5306 Gorgeous Sequence
2. 洛谷 P6242 【模板】线段树 3
3. LeetCode 715. Range Module
4. LeetCode 218. The Skyline Problem
5. Codeforces 438D The Child and Sequence
6. Codeforces 446C DZY Loves Fibonacci Numbers
7. POJ 1177 Picture
8. HDU 1542 Atlantis
9. SPOJ GSS2 Can you answer these queries II
10. SPOJ GSS6 Can you answer these queries VI
11. SPOJ GSS7 Can you answer these queries VII
12. LeetCode 732. My Calendar III

### ## 解题技巧总结

#### ### 1. 基础线段树技巧

- 理解线段树的递归结构

- 掌握 push\_up 操作（自底向上更新）
- 熟练使用懒惰标记优化区间更新

#### ### 2. 动态开点技巧

- 按需创建节点，节省空间
- 注意左右子节点的动态分配
- 合理设置空间上限

#### ### 3. 区间最值操作技巧

- 维护最大值、次大值和最大值个数
- 利用势能分析法分析时间复杂度
- 区分三种更新情况：
  1. 更新值  $\geq$  最大值：无需更新
  2. 次大值  $<$  更新值  $<$  最大值：直接更新
  3. 更新值  $\leq$  次大值：递归处理

#### ### 4. 历史信息维护技巧

- 使用多重懒惰标记
- 区分最大值和非最大值的处理
- 维护历史最大值、历史最小值等信息

#### ### 5. 离散化技巧

- 对大数据范围进行映射处理
- 注意相邻点之间添加额外点避免错误
- 结合离散化和线段树解决区间问题

#### ### 6. 位运算优化技巧

- 使用位运算表示集合状态
- 通过位运算快速计算集合操作
- 适用于颜色统计、状态压缩等问题

#### ### 7. 最大子段和技巧

- 维护区间最大子段和、左最大子段和、右最大子段和
- 通过合并子区间信息得到父区间信息
- 适用于区间最大子段和查询问题

### ## 常见错误及注意事项

#### ### 1. 数组大小问题

- 静态线段树通常需要开 4 倍空间
- 动态开点线段树需要合理估算节点数量

#### ### 2. 懒惰标记问题

- push\_down 操作必须正确实现
- 多种标记的优先级处理
- 标记的正确下传和清除

#### ### 3. 离散化问题

- 相邻点之间可能需要添加额外点
- 端点处理要特别注意
- 离散化后的坐标映射要正确

#### ### 4. 边界条件问题

- 空区间查询的处理
- 单点区间和区间长度为 1 的处理
- 递归终止条件的正确判断

#### ### 5. 位运算问题

- 注意位运算的优先级
- 正确使用左移右移操作
- 避免整数溢出问题

### ## 学习建议

#### ### 1. 学习路径

1. 先掌握基础线段树（单点更新/查询）
2. 学习区间更新和懒惰标记
3. 理解动态开点线段树
4. 掌握区间最值操作（吉司机线段树）
5. 学习历史信息维护
6. 练习各类经典题目

#### ### 2. 实践建议

- 每种类型的题目至少练习 3-5 道
- 对比不同实现方式的优劣
- 注重代码的可读性和可维护性
- 总结每道题目的关键点和易错点

#### ### 3. 进阶方向

- 多维线段树
- 线段树与其他数据结构结合
- 线段树在实际项目中的应用
- 线段树的扩展应用（如李超线段树等）

### ## 新增题目列表

以下是我们新增实现的题目：

#### SPOJ GSS1 - Can you answer these queries I

- \*\*题目描述\*\*：最大子段和查询
- \*\*算法\*\*：线段树
- \*\*难度\*\*：中等
- \*\*相关文件\*\*：[Code06\_MaximumSubarraySum.java](Code06\_MaximumSubarraySum.java),  
[Code06\_MaximumSubarraySum.cpp](Code06\_MaximumSubarraySum.cpp),  
[Code06\_MaximumSubarraySum.py](Code06\_MaximumSubarraySum.py)

#### SPOJ KGSS - Maximum Sum

- \*\*题目描述\*\*：查询区间内两个最大值的和
- \*\*算法\*\*：线段树
- \*\*难度\*\*：中等
- \*\*相关文件\*\*：[Code07\_MaximumTwoValuesSum.java](Code07\_MaximumTwoValuesSum.java),  
[Code07\_MaximumTwoValuesSum.cpp](Code07\_MaximumTwoValuesSum.cpp),  
[Code07\_MaximumTwoValuesSum.py](Code07\_MaximumTwoValuesSum.py)

#### POJ 2777 - Count Color

- \*\*题目描述\*\*：区间染色，查询区间颜色数
- \*\*算法\*\*：线段树 + 位运算
- \*\*难度\*\*：中等
- \*\*相关文件\*\*：[Code08\_CountColor.java](Code08\_CountColor.java),  
[Code08\_CountColor.cpp](Code08\_CountColor.cpp), [Code08\_CountColor.py](Code08\_CountColor.py)

=====

文件：README.md

=====

# 线段树高级应用：动态开点、区间最值操作与历史最值问题

## 概述

本目录(class114)专注于线段树的高级应用，包括动态开点线段树、区间最值操作以及历史最值问题。这些技术在处理大规模数据范围、复杂区间操作和历史信息追踪方面具有重要作用。

## 核心内容

#### 1. 动态开点线段树 (Dynamic Segment Tree)

动态开点线段树是一种在需要时才创建节点的线段树实现方式，特别适用于值域非常大的情况。

**\*\*主要特点：\*\***

- 节点按需创建，节省空间
- 适用于值域极大的情况(如  $10^9$ )
- 使用懒惰标记优化区间操作

**\*\*相关文件：\*\***

- [Code01\_DynamicSegmentTree.java] (Code01\_DynamicSegmentTree.java) - 动态开点线段树实现

**\*\*应用场景：\*\***

- 处理超大范围的区间更新和查询
- 当预先分配所有节点不现实时

### ### 2. 区间最值操作 (Range Min/Max Operations)

区间最值操作涉及对区间内元素执行取最小值或最大值的操作，这是线段树的一种高级应用。

**\*\*关键技术：\*\***

- 吉如一算法(吉司机线段树)
- 维护最大值、次大值及最大值个数
- 势能分析法保证时间复杂度

**\*\*相关文件：\*\***

- [Code03\_SegmentTreeSetminQueryMaxSum1.java] (Code03\_SegmentTreeSetminQueryMaxSum1.java) - 区间最值操作实现(对数器验证)
- [Code03\_SegmentTreeSetminQueryMaxSum2.java] (Code03\_SegmentTreeSetminQueryMaxSum2.java) - 区间最值操作实现(HDU 测试)
- [Code04\_SegmentTreeAddSetminQueryMaxSum.java] (Code04\_SegmentTreeAddSetminQueryMaxSum.java) - 同时支持区间加法和最值操作

**\*\*应用场景：\*\***

- HDU 5306 Gorgeous Sequence
- 洛谷 P6242 【模板】线段树 3

### ### 3. 历史最值问题 (Historical Max/Min Values)

历史最值问题要求维护区间的历史信息，如历史最大值、历史最小值等。

**\*\*关键技术：\*\***

- 多重懒惰标记
- 历史信息的维护和更新
- 最大值与非最大值的区分处理

**\*\*相关文件：\*\***

- [Code02\_CountIntervals.java] (Code02\_CountIntervals.java) - 区间计数实现

- [Code05\_MaximumMinimumHistory.java](Code05\_MaximumMinimumHistory.java) - 区间最值和历史最值实现

**\*\*应用场景:\*\***

- 洛谷 P6242 **【模板】**线段树 3
- LeetCode 715. Range Module

## 经典问题与题解

### 1. HDU 5306 Gorgeous Sequence

**\*\*题目描述:\*\***

维护一个序列  $a$ , 执行以下操作:

1. 0 1  $r$   $t$ : 对于所有的  $i \in [1, r]$ , 将  $a[i]$  变成  $\min(a[i], t)$
2. 1 1  $r$ : 输出  $\max\{a[i] \mid i \in [1, r]\}$
3. 2 1  $r$ : 输出  $\sum\{a[i] \mid i \in [1, r]\}$

**\*\*解法要点:\*\***

- 使用吉司机线段树
- 维护最大值( $mx$ )、次大值( $sem$ )、最大值个数( $cnt$ )和区间和( $sum$ )
- 当  $se < v < mx$  时可以直接更新, 否则需要递归处理

### 2. 洛谷 P6242 **【模板】**线段树 3

**\*\*题目描述:\*\***

给出一个长度为  $n$  的数列  $A$ , 同时定义一个辅助数组  $B$ ,  $B$  开始与  $A$  完全相同。接下来进行  $m$  次操作:

1. 1 1  $r$   $k$ : 对于所有的  $i \in [1, r]$ , 将  $A[i]$  加上  $k$  ( $k$  可以为负数)
2. 2 1  $r$   $v$ : 对于所有的  $i \in [1, r]$ , 将  $A[i]$  变成  $\min(A[i], v)$
3. 3 1  $r$ : 求  $\sum\{A[i] \mid i \in [1, r]\}$
4. 4 1  $r$ : 对于所有的  $i \in [1, r]$ , 求  $A[i]$  的最大值
5. 5 1  $r$ : 对于所有的  $i \in [1, r]$ , 求  $B[i]$  的最大值

在每一次操作后, 都进行一次更新, 让  $B[i] \leftarrow \max(B[i], A[i])$

**\*\*解法要点:\*\***

- 结合区间加法和区间最值操作
- 维护历史最大值信息
- 使用多重懒惰标记

### 3. LeetCode 715. Range Module

**\*\*题目描述:\*\***

Range Module 是一个模块, 用于跟踪数字范围。设计一个数据结构来高效地实现以下接口:

1. `addRange(left, right)`: 添加半开区间  $[left, right)$

2. `queryRange(left, right)`: 查询半开区间  $[left, right)$  是否完全被跟踪
3. `removeRange(left, right)`: 移除半开区间  $[left, right)$  的跟踪

**\*\*解法要点:\*\***

- 使用动态开点线段树
- 维护区间覆盖状态
- 支持区间设置和查询操作

#### ### 4. POJ 3468 A Simple Problem with Integers

**\*\*题目描述:\*\***

给定一个长度为  $N$  的整数序列，执行以下操作：

1. `C a b c`: 将区间  $[a, b]$  中的每个数都加上  $c$
2. `Q a b`: 查询区间  $[a, b]$  中所有数的和

**\*\*解法要点:\*\***

- 经典的线段树区间更新和查询
- 使用懒惰标记优化区间加法操作

#### ### 5. POJ 2528 Mayor's posters

**\*\*题目描述:\*\***

城市的墙上贴海报，每张海报贴在一个连续区间上。后来贴的海报会覆盖之前贴的海报。求最后可以看到多少张不同的海报。

**\*\*解法要点:\*\***

- 线段树区间染色问题
- 离散化处理大数据范围
- 倒序处理或使用线段树维护区间颜色

#### ### 6. SPOJ GSS1 - Can you answer these queries I

**\*\*题目描述:\*\***

给定一个长度为  $n$  的整数序列，执行  $m$  次查询操作，每次查询  $[l, r]$  区间内的最大子段和。

**\*\*解法要点:\*\***

- 使用线段树维护区间信息
- 每个节点存储区间最大子段和、左最大子段和、右最大子段和和区间总和
- 通过合并子区间信息得到父区间信息

#### ### 7. SPOJ KGSS - Maximum Sum

**\*\*题目描述:\*\***

给定一个长度为  $n$  的整数序列，执行  $m$  次操作：

1.  $U\ i\ x$ ：将第  $i$  个位置的值更新为  $x$
2.  $Q\ l\ r$ ：查询  $[l, r]$  区间内两个最大值的和

**\*\*解法要点：\*\***

- 使用线段树维护区间最大值和次大值
- 通过合并子区间信息得到父区间信息
- 支持单点更新和区间查询操作

#### #### 8. POJ 2777 - Count Color

**\*\*题目描述：\*\***

给定一个长度为  $L$  的板条，初始时所有位置都是颜色 1，执行 0 次操作：

1. " $C\ A\ B\ C$ "：将区间  $[A, B]$  染成颜色  $C$
2. " $P\ A\ B$ "：查询区间  $[A, B]$  中有多少种不同的颜色

**\*\*解法要点：\*\***

- 使用线段树维护区间颜色集合(用位运算表示)
- 结合懒惰标记实现区间染色
- 通过位运算计算颜色种类数

**\*\*相关文件：\*\***

- [Code12\_CountColor.java] (Code12\_CountColor.java) - Java 实现
- [Code12\_CountColor.cpp] (Code12\_CountColor.cpp) - C++实现
- [Code12\_CountColor.py] (Code12\_CountColor.py) - Python 实现

#### #### 9. SPOJ GSS1 - Can you answer these queries I

**\*\*题目描述：\*\***

给定一个长度为  $n$  的整数序列，执行  $m$  次查询操作，每次查询  $[l, r]$  区间内的最大子段和。

**\*\*解法要点：\*\***

- 使用线段树维护区间信息
- 每个节点存储：区间最大子段和、左最大子段和、右最大子段和、区间总和
- 通过合并子区间信息得到父区间信息

**\*\*相关文件：\*\***

- [Code13\_MaximumSubarraySum.java] (Code13\_MaximumSubarraySum.java) - Java 实现

#### #### 10. LeetCode 307. Range Sum Query - Mutable

**\*\*题目描述：\*\***

设计一个数据结构，支持以下操作：

1. 更新数组中的某个元素
2. 查询区间和

**\*\*解法要点:\*\***

- 使用线段树实现单点更新和区间查询
- 支持动态修改数组元素

**\*\*相关文件:\*\***

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

### ### 11. HDU 5306 Gorgeous Sequence

**\*\*题目描述:\*\***

维护一个序列  $a$ ，执行以下操作：

1.  $0\ l\ r\ t$ : 对于所有的  $i \in [l, r]$ ，将  $a[i]$  变成  $\min(a[i], t)$
2.  $1\ l\ r$ : 输出  $\max\{a[i] \mid i \in [l, r]\}$
3.  $2\ l\ r$ : 输出  $\sum\{a[i] \mid i \in [l, r]\}$

**\*\*解法要点:\*\***

- 使用吉司机线段树
- 维护最大值、次大值及最大值个数
- 势能分析法保证时间复杂度

**\*\*相关文件:\*\***

- [Code10\_RangeMinimumQuery.java](Code10\_RangeMinimumQuery.java) - Java 实现
- [Code10\_RangeMinimumQuery.cpp](Code10\_RangeMinimumQuery.cpp) - C++实现
- [Code10\_RangeMinimumQuery.py](Code10\_RangeMinimumQuery.py) - Python 实现

### ### 12. POJ 3468 A Simple Problem with Integers

**\*\*题目描述:\*\***

给定一个长度为  $N$  的整数序列，执行以下操作：

1.  $C\ a\ b\ c$ : 将区间  $[a, b]$  中的每个数都加上  $c$
2.  $Q\ a\ b$ : 查询区间  $[a, b]$  中所有数的和

**\*\*解法要点:\*\***

- 经典的线段树区间更新和查询
- 使用懒惰标记优化区间加法操作

**\*\*相关文件:\*\***

- [Code11\_RangeAddQuery.java](Code11\_RangeAddQuery.java) - Java 实现

- [Code11\_RangeAddQuery.cpp](Code11\_RangeAddQuery.cpp) - C++实现
- [Code11\_RangeAddQuery.py](Code11\_RangeAddQuery.py) - Python 实现

## ## 算法复杂度分析

### ### 时间复杂度

#### 1. \*\*基本操作\*\*:

- 单点更新:  $O(\log n)$
- 区间查询:  $O(\log n)$
- 区间更新(带懒惰标记):  $O(\log n)$

#### 2. \*\*高级操作\*\*:

- 区间最值操作(吉司机线段树):  $O(n \log^2 n)$  均摊
- 历史最值查询:  $O(\log n)$
- 最大子段和查询:  $O(\log n)$
- 颜色计数查询:  $O(\log n)$

### ### 空间复杂度

1. \*\*静态线段树\*\*: $O(4n)$
2. \*\*动态开点线段树\*\*: $O(q \log U)$ , 其中  $q$  是操作次数,  $U$  是值域大小

## ## 工程化考虑

### ### 异常处理

- 输入验证(区间边界、操作类型等)
- 空间不足时的处理
- 非法操作的防御性编程

### ### 性能优化

- 懒惰标记延迟更新
- 势能分析法优化复杂度
- 内存池技术避免频繁内存分配
- 位运算优化颜色集合操作

### ### 可维护性

- 代码模块化设计
- 详细注释说明算法原理
- 清晰的变量命名

## ## 跨语言实现对比

#### #### Java

- 面向对象特性良好支持
- 内存管理自动化
- 适合复杂数据结构实现

#### #### C++

- 性能优势明显
- 内存控制精细
- 模板编程支持泛型

#### #### Python

- 语法简洁
- 开发效率高
- 性能相对较弱但可接受

### ## 学习路径建议

#### 1. \*\*基础掌握\*\*:

- 理解线段树的基本原理
- 掌握单点更新/查询
- 熟悉区间更新/查询及懒惰标记

#### 2. \*\*进阶学习\*\*:

- 动态开点线段树
- 区间最值操作
- 历史信息维护

#### 3. \*\*高阶应用\*\*:

- 多维线段树
- 线段树与其他数据结构结合
- 线段树在实际项目中的应用

### ## 参考资料

1. 吉如一. 《区间最值操作与历史最值问题》. 2016 国家集训队论文
2. 《算法导论》第 14 章 数据结构的扩张
3. Competitive Programming Resources
4. OI Wiki 线段树相关章节

=====

文件: SOLUTIONS.md

=====

## # 线段树题目详解与实现

### ## 1. HDU 5306 Gorgeous Sequence

#### ### 题目描述

维护一个序列  $a$ ，执行以下操作：

1. 0 1  $r$   $t$ : 对于所有的  $i \in [1, r]$ ，将  $a[i]$  变成  $\min(a[i], t)$
2. 1 1  $r$ : 输出  $\max\{a[i] \mid i \in [1, r]\}$
3. 2 1  $r$ : 输出  $\sum\{a[i] \mid i \in [1, r]\}$

#### ### 解题思路

这是经典的吉司机线段树问题，核心思想是维护区间最大值、次大值和最大值个数，利用势能分析法保证时间复杂度。

#### ### 关键点

1. 维护信息：最大值( $mx$ )、严格次大值( $sem$ )、最大值个数( $cnt$ )、区间和( $sum$ )
2. 三种更新情况：
  - 当更新值  $\geq$  最大值时，无需更新
  - 当次大值  $<$  更新值  $<$  最大值时，直接更新最大值
  - 当更新值  $\leq$  次大值时，递归处理左右子树

#### ### Java 实现

已在 [Code03\_SegmentTreeSetminQueryMaxSum2.java] (Code03\_SegmentTreeSetminQueryMaxSum2.java) 中实现。

#### ### C++实现

```
```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 1000006;
const int INF = 0x3f3f3f3f;

struct Node {
    long long sum;
    int mx, sem, cnt;
} tree[MAXN << 2];

int arr[MAXN];
int n, m;

inline void push_up(int rt) {
    int l = rt << 1, r = rt << 1 | 1;
```

```

tree[rt].sum = tree[l].sum + tree[r].sum;

if (tree[l].mx > tree[r].mx) {
    tree[rt].mx = tree[l].mx;
    tree[rt].cnt = tree[l].cnt;
    tree[rt].sem = max(tree[l].sem, tree[r].mx);
} else if (tree[l].mx < tree[r].mx) {
    tree[rt].mx = tree[r].mx;
    tree[rt].cnt = tree[r].cnt;
    tree[rt].sem = max(tree[l].mx, tree[r].sem);
} else {
    tree[rt].mx = tree[l].mx;
    tree[rt].cnt = tree[l].cnt + tree[r].cnt;
    tree[rt].sem = max(tree[l].sem, tree[r].sem);
}
}

inline void apply(int rt, int v) {
    if (v >= tree[rt].mx) return;
    tree[rt].sum -= (long long)(tree[rt].mx - v) * tree[rt].cnt;
    tree[rt].mx = v;
}

inline void push_down(int rt) {
    if (tree[rt].mx != INF) {
        apply(rt << 1, tree[rt].mx);
        apply(rt << 1 | 1, tree[rt].mx);
        tree[rt].mx = INF;
    }
}

void build(int l, int r, int rt) {
    tree[rt].mx = -1;
    tree[rt].sem = -1;
    tree[rt].cnt = 0;

    if (l == r) {
        scanf("%d", &tree[rt].mx);
        tree[rt].sum = tree[rt].mx;
        tree[rt].cnt = 1;
        return;
    }
}

```

```

    int mid = (l + r) >> 1;
    build(l, mid, rt << 1);
    build(mid + 1, r, rt << 1 | 1);
    push_up(rt);
}

void update(int L, int R, int v, int l, int r, int rt) {
    if (v >= tree[rt].mx) return;

    if (L <= l && r <= R && tree[rt].sem < v) {
        apply(rt, v);
        return;
    }

    // push_down(rt);
    int mid = (l + r) >> 1;
    if (L <= mid) update(L, R, v, l, mid, rt << 1);
    if (R > mid) update(L, R, v, mid + 1, r, rt << 1 | 1);
    push_up(rt);
}

int query_max(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return tree[rt].mx;
    }

    // push_down(rt);
    int mid = (l + r) >> 1;
    int res = -INF;
    if (L <= mid) res = max(res, query_max(L, R, l, mid, rt << 1));
    if (R > mid) res = max(res, query_max(L, R, mid + 1, r, rt << 1 | 1));
    return res;
}

long long query_sum(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return tree[rt].sum;
    }

    // push_down(rt);
    int mid = (l + r) >> 1;
    long long res = 0;
    if (L <= mid) res += query_sum(L, R, l, mid, rt << 1);

```

```

        if (R > mid) res += query_sum(L, R, mid + 1, r, rt << 1 | 1);
    return res;
}

int main() {
    int T;
    scanf("%d", &T);
    while (T--) {
        scanf("%d%d", &n, &m);
        build(1, n, 1);

        for (int i = 1; i <= m; i++) {
            int op, l, r, v;
            scanf("%d", &op);
            if (op == 0) {
                scanf("%d%d%d", &l, &r, &v);
                update(1, r, v, 1, n, 1);
            } else if (op == 1) {
                scanf("%d%d", &l, &r);
                printf("%d\n", query_max(1, r, 1, n, 1));
            } else {
                scanf("%d%d", &l, &r);
                printf("%lld\n", query_sum(1, r, 1, n, 1));
            }
        }
    }
    return 0;
}

```

Python 实现

```

```python
import sys
from math import inf

class SegmentTree:
 def __init__(self, arr):
 self.n = len(arr)
 self.tree = [{'sum': 0, 'mx': -1, 'sem': -1, 'cnt': 0} for _ in range(4 * self.n)]
 self.arr = arr
 self.build(1, 0, self.n - 1)

 def push_up(self, rt):

```

```

l, r = 2 * rt, 2 * rt + 1
self.tree[rt]['sum'] = self.tree[l]['sum'] + self.tree[r]['sum']

```

```

if self.tree[l]['mx'] > self.tree[r]['mx']:
 self.tree[rt]['mx'] = self.tree[l]['mx']
 self.tree[rt]['cnt'] = self.tree[l]['cnt']
 self.tree[rt]['sem'] = max(self.tree[l]['sem'], self.tree[r]['mx'])
elif self.tree[l]['mx'] < self.tree[r]['mx']:
 self.tree[rt]['mx'] = self.tree[r]['mx']
 self.tree[rt]['cnt'] = self.tree[r]['cnt']
 self.tree[rt]['sem'] = max(self.tree[l]['mx'], self.tree[r]['sem'])
else:
 self.tree[rt]['mx'] = self.tree[l]['mx']
 self.tree[rt]['cnt'] = self.tree[l]['cnt'] + self.tree[r]['cnt']
 self.tree[rt]['sem'] = max(self.tree[l]['sem'], self.tree[r]['sem'])

```

```

def apply(self, rt, v):
 if v >= self.tree[rt]['mx']:
 return
 self.tree[rt]['sum'] -= (self.tree[rt]['mx'] - v) * self.tree[rt]['cnt']
 self.tree[rt]['mx'] = v

```

```

def build(self, rt, l, r):
 self.tree[rt]['mx'] = -1
 self.tree[rt]['sem'] = -1
 self.tree[rt]['cnt'] = 0

 if l == r:
 self.tree[rt]['mx'] = self.arr[l]
 self.tree[rt]['sum'] = self.arr[l]
 self.tree[rt]['cnt'] = 1
 return

```

```

mid = (l + r) // 2
self.build(2 * rt, l, mid)
self.build(2 * rt + 1, mid + 1, r)
self.push_up(rt)

```

```

def update(self, L, R, v, l, r, rt):
 if v >= self.tree[rt]['mx']:
 return

 if L <= l and r <= R and self.tree[rt]['sem'] < v:

```

```

 self.apply(rt, v)
 return

mid = (l + r) // 2
if L <= mid:
 self.update(L, R, v, l, mid, 2 * rt)
if R > mid:
 self.update(L, R, v, mid + 1, r, 2 * rt + 1)
self.push_up(rt)

def query_max(self, L, R, l, r, rt):
 if L <= l and r <= R:
 return self.tree[rt]['mx']

 mid = (l + r) // 2
 res = -inf
 if L <= mid:
 res = max(res, self.query_max(L, R, l, mid, 2 * rt))
 if R > mid:
 res = max(res, self.query_max(L, R, mid + 1, r, 2 * rt + 1))
 return res

def query_sum(self, L, R, l, r, rt):
 if L <= l and r <= R:
 return self.tree[rt]['sum']

 mid = (l + r) // 2
 res = 0
 if L <= mid:
 res += self.query_sum(L, R, l, mid, 2 * rt)
 if R > mid:
 res += self.query_sum(L, R, mid + 1, r, 2 * rt + 1)
 return res

def main():
 input = sys.stdin.read
 data = input().split()

 idx = 0
 T = int(data[idx])
 idx += 1

 for _ in range(T):

```

```

n = int(data[idx])
m = int(data[idx + 1])
idx += 2

arr = [int(data[idx + i]) for i in range(n)]
idx += n

seg_tree = SegmentTree(arr)

for _ in range(m):
 op = int(data[idx])
 idx += 1

 if op == 0:
 l = int(data[idx]) - 1
 r = int(data[idx + 1]) - 1
 v = int(data[idx + 2])
 idx += 3
 seg_tree.update(l, r, v, 0, n - 1, 1)
 elif op == 1:
 l = int(data[idx]) - 1
 r = int(data[idx + 1]) - 1
 idx += 2
 print(seg_tree.query_max(l, r, 0, n - 1, 1))
 else:
 l = int(data[idx]) - 1
 r = int(data[idx + 1]) - 1
 idx += 2
 print(seg_tree.query_sum(l, r, 0, n - 1, 1))

if __name__ == "__main__":
 main()

```

### ### 时间复杂度分析

- 建树:  $O(n)$
- 区间最值操作:  $O(n \log^2 n)$  均摊
- 区间查询:  $O(\log n)$

### ### 空间复杂度分析

- $O(n)$

#### #### 题目描述

给出一个长度为  $n$  的数列  $A$ ，同时定义一个辅助数组  $B$ ， $B$  开始与  $A$  完全相同。接下来进行  $m$  次操作：

1. 1 1  $r$   $k$ : 对于所有的  $i \in [1, r]$ ，将  $A[i]$  加上  $k$  ( $k$  可以为负数)
2. 2 1  $r$   $v$ : 对于所有的  $i \in [1, r]$ ，将  $A[i]$  变成  $\min(A[i], v)$
3. 3 1  $r$ : 求  $\sum \{A[i] \mid i \in [1, r]\}$
4. 4 1  $r$ : 对于所有的  $i \in [1, r]$ ，求  $A[i]$  的最大值
5. 5 1  $r$ : 对于所有的  $i \in [1, r]$ ，求  $B[i]$  的最大值

在每一次操作后，都进行一次更新，让  $B[i] \leftarrow \max(B[i], A[i])$

#### #### 解题思路

这是区间加法、区间最值操作和历史最值查询的综合应用。需要维护多种信息和懒惰标记。

#### #### 关键点

1. 维护信息：区间和(sum)、最大值(mx)、次大值(sem)、最大值个数(cnt)、历史最大值(max\_history)
2. 懒惰标记：最大值增加量(max\_add)、其他值增加量(other\_add)、最大值历史最大增加量(max\_add\_top)、其他值历史最大增加量(other\_add\_top)
3. 操作 2 需要使用吉司机线段树的技术
4. 操作 5 需要维护历史最大值信息

#### #### Java 实现

已在 [Code05\_MaximumMinimumHistory.java] (Code05\_MaximumMinimumHistory.java) 中实现。

#### #### C++实现

```
```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 500006;
const long long INF = 1e18;

struct Node {
    long long sum, mx, sem, max_history;
    long long max_add, other_add;
    long long max_add_top, other_add_top;
    int cnt;
} tree[MAXN << 2];

long long arr[MAXN];
int n, m;

inline void push_up(int rt) {
```

```

int l = rt << 1, r = rt << 1 | 1;
tree[rt].max_history = max(tree[l].max_history, tree[r].max_history);
tree[rt].sum = tree[l].sum + tree[r].sum;

if (tree[l].mx > tree[r].mx) {
    tree[rt].mx = tree[l].mx;
    tree[rt].cnt = tree[l].cnt;
    tree[rt].sem = max(tree[l].sem, tree[r].mx);
} else if (tree[l].mx < tree[r].mx) {
    tree[rt].mx = tree[r].mx;
    tree[rt].cnt = tree[r].cnt;
    tree[rt].sem = max(tree[l].mx, tree[r].sem);
} else {
    tree[rt].mx = tree[l].mx;
    tree[rt].cnt = tree[l].cnt + tree[r].cnt;
    tree[rt].sem = max(tree[l].sem, tree[r].sem);
}
}

inline void apply(int rt, int len, long long max_add_v, long long other_add_v,
                 long long max_up_v, long long other_up_v) {
    tree[rt].max_history = max(tree[rt].max_history, tree[rt].mx + max_up_v);
    tree[rt].max_add_top = max(tree[rt].max_add_top, tree[rt].max_add + max_up_v);
    tree[rt].other_add_top = max(tree[rt].other_add_top, tree[rt].other_add + other_up_v);

    tree[rt].sum += max_add_v * tree[rt].cnt + other_add_v * (len - tree[rt].cnt);
    tree[rt].mx += max_add_v;
    if (tree[rt].sem != -INF) tree[rt].sem += other_add_v;
    tree[rt].max_add += max_add_v;
    tree[rt].other_add += other_add_v;
}

inline void push_down(int rt, int ln, int rn) {
    int l = rt << 1, r = rt << 1 | 1;
    long long tmp = max(tree[l].mx, tree[r].mx);

    if (tree[l].mx == tmp) {
        apply(l, ln, tree[rt].max_add, tree[rt].other_add,
              tree[rt].max_add_top, tree[rt].other_add_top);
    } else {
        apply(l, ln, tree[rt].other_add, tree[rt].other_add,
              tree[rt].other_add_top, tree[rt].other_add_top);
    }
}

```

```

    if (tree[r].mx == tmp) {
        apply(r, rn, tree[rt].max_add, tree[rt].other_add,
            tree[rt].max_add_top, tree[rt].other_add_top);
    } else {
        apply(r, rn, tree[rt].other_add, tree[rt].other_add,
            tree[rt].other_add_top, tree[rt].other_add_top);
    }

    tree[rt].max_add = tree[rt].other_add = 0;
    tree[rt].max_add_top = tree[rt].other_add_top = 0;
}

void build(int l, int r, int rt) {
    tree[rt].max_add = tree[rt].other_add = 0;
    tree[rt].max_add_top = tree[rt].other_add_top = 0;

    if (l == r) {
        tree[rt].sum = tree[rt].mx = tree[rt].max_history = arr[l];
        tree[rt].sem = -INF;
        tree[rt].cnt = 1;
        return;
    }

    int mid = (l + r) >> 1;
    build(l, mid, rt << 1);
    build(mid + 1, r, rt << 1 | 1);
    push_up(rt);
}

void add(int L, int R, long long v, int l, int r, int rt) {
    if (L <= l && r <= R) {
        apply(rt, r - l + 1, v, v, v, v);
        return;
    }

    int mid = (l + r) >> 1;
    push_down(rt, mid - l + 1, r - mid);
    if (L <= mid) add(L, R, v, l, mid, rt << 1);
    if (R > mid) add(L, R, v, mid + 1, r, rt << 1 | 1);
    push_up(rt);
}

```

```

void set_min(int L, int R, long long v, int l, int r, int rt) {
    if (v >= tree[rt].mx) return;

    if (L <= l && r <= R && tree[rt].sem < v) {
        apply(rt, r - l + 1, v - tree[rt].mx, 0, v - tree[rt].mx, 0);
        return;
    }

    int mid = (l + r) >> 1;
    push_down(rt, mid - l + 1, r - mid);
    if (L <= mid) set_min(L, R, v, l, mid, rt << 1);
    if (R > mid) set_min(L, R, v, mid + 1, r, rt << 1 | 1);
    push_up(rt);
}

long long query_sum(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return tree[rt].sum;
    }

    int mid = (l + r) >> 1;
    push_down(rt, mid - l + 1, r - mid);
    long long res = 0;
    if (L <= mid) res += query_sum(L, R, l, mid, rt << 1);
    if (R > mid) res += query_sum(L, R, mid + 1, r, rt << 1 | 1);
    return res;
}

long long query_max(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {
        return tree[rt].mx;
    }

    int mid = (l + r) >> 1;
    push_down(rt, mid - l + 1, r - mid);
    long long res = -INF;
    if (L <= mid) res = max(res, query_max(L, R, l, mid, rt << 1));
    if (R > mid) res = max(res, query_max(L, R, mid + 1, r, rt << 1 | 1));
    return res;
}

long long query_history_max(int L, int R, int l, int r, int rt) {
    if (L <= l && r <= R) {

```

```

        return tree[rt].max_history;
    }

    int mid = (l + r) >> 1;
    push_down(rt, mid - 1 + 1, r - mid);
    long long res = -INF;
    if (L <= mid) res = max(res, query_history_max(L, R, l, mid, rt << 1));
    if (R > mid) res = max(res, query_history_max(L, R, mid + 1, r, rt << 1 | 1));
    return res;
}

int main() {
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i++) {
        scanf("%lld", &arr[i]);
    }
    build(1, n, 1);

    for (int i = 1; i <= m; i++) {
        int op, l, r;
        long long v;
        scanf("%d%d%d", &op, &l, &r);

        if (op == 1) {
            scanf("%lld", &v);
            add(1, r, v, 1, n, 1);
        } else if (op == 2) {
            scanf("%lld", &v);
            set_min(1, r, v, 1, n, 1);
        } else if (op == 3) {
            printf("%lld\n", query_sum(1, r, 1, n, 1));
        } else if (op == 4) {
            printf("%lld\n", query_max(1, r, 1, n, 1));
        } else {
            printf("%lld\n", query_history_max(1, r, 1, n, 1));
        }
    }

    return 0;
}

```

Python 实现

```

```python
import sys
from math import inf

class SegmentTree:
 def __init__(self, arr):
 self.n = len(arr)
 self.tree = [None] * (4 * self.n)
 for i in range(4 * self.n):
 self.tree[i] = {
 'sum': 0, 'mx': 0, 'sem': 0, 'max_history': 0,
 'max_add': 0, 'other_add': 0,
 'max_add_top': 0, 'other_add_top': 0,
 'cnt': 0
 }
 self.arr = arr
 self.build(1, 0, self.n - 1)

 def push_up(self, rt):
 l, r = 2 * rt, 2 * rt + 1
 self.tree[rt]['max_history'] = max(self.tree[l]['max_history'],
self.tree[r]['max_history'])
 self.tree[rt]['sum'] = self.tree[l]['sum'] + self.tree[r]['sum']

 if self.tree[l]['mx'] > self.tree[r]['mx']:
 self.tree[rt]['mx'] = self.tree[l]['mx']
 self.tree[rt]['cnt'] = self.tree[l]['cnt']
 self.tree[rt]['sem'] = max(self.tree[l]['sem'], self.tree[r]['mx'])
 elif self.tree[l]['mx'] < self.tree[r]['mx']:
 self.tree[rt]['mx'] = self.tree[r]['mx']
 self.tree[rt]['cnt'] = self.tree[r]['cnt']
 self.tree[rt]['sem'] = max(self.tree[l]['mx'], self.tree[r]['sem'])
 else:
 self.tree[rt]['mx'] = self.tree[l]['mx']
 self.tree[rt]['cnt'] = self.tree[l]['cnt'] + self.tree[r]['cnt']
 self.tree[rt]['sem'] = max(self.tree[l]['sem'], self.tree[r]['sem'])

 def apply(self, rt, length, max_add_v, other_add_v, max_up_v, other_up_v):
 self.tree[rt]['max_history'] = max(self.tree[rt]['max_history'], self.tree[rt]['mx'] +
max_up_v)
 self.tree[rt]['max_add_top'] = max(self.tree[rt]['max_add_top'], self.tree[rt]['max_add']
+ max_up_v)
 self.tree[rt]['other_add_top'] = max(self.tree[rt]['other_add_top'],

```

```
self.tree[rt]['other_add'] + other_up_v)
```

```
self.tree[rt]['sum'] += max_add_v * self.tree[rt]['cnt'] + other_add_v * (length -
self.tree[rt]['cnt'])
```

```
self.tree[rt]['mx'] += max_add_v
```

```
if self.tree[rt]['sem'] != -inf:
```

```
 self.tree[rt]['sem'] += other_add_v
```

```
self.tree[rt]['max_add'] += max_add_v
```

```
self.tree[rt]['other_add'] += other_add_v
```

```
def push_down(self, rt, ln, rn):
```

```
 l, r = 2 * rt, 2 * rt + 1
```

```
 tmp = max(self.tree[l]['mx'], self.tree[r]['mx'])
```

```
 if self.tree[l]['mx'] == tmp:
```

```
 self.apply(l, ln, self.tree[rt]['max_add'], self.tree[rt]['other_add'],
 self.tree[rt]['max_add_top'], self.tree[rt]['other_add_top'])
```

```
 else:
```

```
 self.apply(l, ln, self.tree[rt]['other_add'], self.tree[rt]['other_add'],
 self.tree[rt]['other_add_top'], self.tree[rt]['other_add_top'])
```

```
 if self.tree[r]['mx'] == tmp:
```

```
 self.apply(r, rn, self.tree[rt]['max_add'], self.tree[rt]['other_add'],
 self.tree[rt]['max_add_top'], self.tree[rt]['other_add_top'])
```

```
 else:
```

```
 self.apply(r, rn, self.tree[rt]['other_add'], self.tree[rt]['other_add'],
 self.tree[rt]['other_add_top'], self.tree[rt]['other_add_top'])
```

```
self.tree[rt]['max_add'] = self.tree[rt]['other_add'] = 0
```

```
self.tree[rt]['max_add_top'] = self.tree[rt]['other_add_top'] = 0
```

```
def build(self, rt, l, r):
```

```
 self.tree[rt]['max_add'] = self.tree[rt]['other_add'] = 0
```

```
 self.tree[rt]['max_add_top'] = self.tree[rt]['other_add_top'] = 0
```

```
 if l == r:
```

```
 self.tree[rt]['sum'] = self.tree[rt]['mx'] = self.tree[rt]['max_history'] =
```

```
self.arr[l]
```

```
 self.tree[rt]['sem'] = -inf
```

```
 self.tree[rt]['cnt'] = 1
```

```
 return
```

```
 mid = (l + r) // 2
```

```

self.build(2 * rt, l, mid)
self.build(2 * rt + 1, mid + 1, r)
self.push_up(rt)

```

```

def add(self, L, R, v, l, r, rt):
 if L <= l and r <= R:
 self.apply(rt, r - l + 1, v, v, v, v)
 return

 mid = (l + r) // 2
 self.push_down(rt, mid - l + 1, r - mid)
 if L <= mid:
 self.add(L, R, v, l, mid, 2 * rt)
 if R > mid:
 self.add(L, R, v, mid + 1, r, 2 * rt + 1)
 self.push_up(rt)

```

```

def set_min(self, L, R, v, l, r, rt):
 if v >= self.tree[rt]['mx']:
 return

 if L <= l and r <= R and self.tree[rt]['sem'] < v:
 self.apply(rt, r - l + 1, v - self.tree[rt]['mx'], 0, v - self.tree[rt]['mx'], 0)
 return

 mid = (l + r) // 2
 self.push_down(rt, mid - l + 1, r - mid)
 if L <= mid:
 self.set_min(L, R, v, l, mid, 2 * rt)
 if R > mid:
 self.set_min(L, R, v, mid + 1, r, 2 * rt + 1)
 self.push_up(rt)

```

```

def query_sum(self, L, R, l, r, rt):
 if L <= l and r <= R:
 return self.tree[rt]['sum']

 mid = (l + r) // 2
 self.push_down(rt, mid - l + 1, r - mid)
 res = 0
 if L <= mid:
 res += self.query_sum(L, R, l, mid, 2 * rt)
 if R > mid:

```

```

 res += self.query_sum(L, R, mid + 1, r, 2 * rt + 1)
 return res

```

```

def query_max(self, L, R, l, r, rt):
 if L <= l and r <= R:
 return self.tree[rt]['mx']

 mid = (l + r) // 2
 self.push_down(rt, mid - l + 1, r - mid)
 res = -inf
 if L <= mid:
 res = max(res, self.query_max(L, R, l, mid, 2 * rt))
 if R > mid:
 res = max(res, self.query_max(L, R, mid + 1, r, 2 * rt + 1))
 return res

```

```

def query_history_max(self, L, R, l, r, rt):
 if L <= l and r <= R:
 return self.tree[rt]['max_history']

 mid = (l + r) // 2
 self.push_down(rt, mid - l + 1, r - mid)
 res = -inf
 if L <= mid:
 res = max(res, self.query_history_max(L, R, l, mid, 2 * rt))
 if R > mid:
 res = max(res, self.query_history_max(L, R, mid + 1, r, 2 * rt + 1))
 return res

```

```

def main():
 input = sys.stdin.read
 data = input().split()

 idx = 0
 n = int(data[idx])
 m = int(data[idx + 1])
 idx += 2

 arr = [int(data[idx + i]) for i in range(n)]
 idx += n

 seg_tree = SegmentTree(arr)

```

```

for _ in range(m):
 op = int(data[idx])
 l = int(data[idx + 1]) - 1 # 转换为 0 索引
 r = int(data[idx + 2]) - 1
 idx += 3

 if op == 1:
 v = int(data[idx])
 idx += 1
 seg_tree.add(l, r, v, 0, n - 1, 1)
 elif op == 2:
 v = int(data[idx])
 idx += 1
 seg_tree.set_min(l, r, v, 0, n - 1, 1)
 elif op == 3:
 print(seg_tree.query_sum(l, r, 0, n - 1, 1))
 elif op == 4:
 print(seg_tree.query_max(l, r, 0, n - 1, 1))
 else: # op == 5
 print(seg_tree.query_history_max(l, r, 0, n - 1, 1))

if __name__ == "__main__":
 main()

```

#### ### 时间复杂度分析

- 建树:  $O(n)$
- 区间加法:  $O(\log n)$
- 区间最值操作:  $O(n \log^2 n)$  均摊
- 区间查询:  $O(\log n)$

#### ### 空间复杂度分析

- $O(n)$

### ## 3. LeetCode 715. Range Module

#### ### 题目描述

Range Module 是一个模块，用于跟踪数字范围。设计一个数据结构来高效地实现以下接口：

1. addRange(left, right): 添加半开区间  $[left, right)$
2. queryRange(left, right): 查询半开区间  $[left, right)$  是否完全被跟踪
3. removeRange(left, right): 移除半开区间  $[left, right)$  的跟踪

#### ### 解题思路

使用动态开点线段树来维护区间覆盖状态。由于值域很大( $10^9$ )，不能预先建立完整的线段树，需要按需创建节点。

#### ### 关键点

1. 动态开点：只在需要时创建节点
2. 懒惰标记：维护区间覆盖状态
3. 区间操作：支持设置区间为覆盖或未覆盖状态

#### ### Java 实现

已在 [Code01\_DynamicSegmentTree.java] (Code01\_DynamicSegmentTree.java) 和 [Code02\_CountIntervals.java] (Code02\_CountIntervals.java) 中实现。

#### ### C++实现

```
```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 1000006;
const int INF = 1e9;

struct Node {
    int ls, rs;
    int sum, add;
} tree[MAXN];

int cnt = 1;
int root = 1;

inline void push_up(int rt) {
    tree[rt].sum = tree[tree[rt].ls].sum + tree[tree[rt].rs].sum;
}

inline void apply(int rt, int len, int v) {
    tree[rt].sum = v ? len : 0;
    tree[rt].add = v;
}

inline void push_down(int rt, int ln, int rn) {
    if (!tree[rt].add) return;

    if (!tree[rt].ls) tree[rt].ls = ++cnt;
    if (!tree[rt].rs) tree[rt].rs = ++cnt;
```

```

    apply(tree[rt].ls, ln, tree[rt].add);
    apply(tree[rt].rs, rn, tree[rt].add);
    tree[rt].add = 0;
}

void update(int &rt, int l, int r, int L, int R, int v) {
    if (!rt) rt = ++cnt;

    if (L <= l && r <= R) {
        apply(rt, r - l + 1, v);
        return;
    }

    int mid = (l + r) >> 1;
    push_down(rt, mid - l + 1, r - mid);

    if (L <= mid) update(tree[rt].ls, l, mid, L, R, v);
    if (R > mid) update(tree[rt].rs, mid + 1, r, L, R, v);

    push_up(rt);
}

int query(int rt, int l, int r, int L, int R) {
    if (!rt) return 0;

    if (L <= l && r <= R) {
        return tree[rt].sum;
    }

    int mid = (l + r) >> 1;
    push_down(rt, mid - l + 1, r - mid);

    int res = 0;
    if (L <= mid) res += query(tree[rt].ls, l, mid, L, R);
    if (R > mid) res += query(tree[rt].rs, mid + 1, r, L, R);

    return res;
}

class RangeModule {
private:
    int root;

```

```

public:
    RangeModule() {
        root = 0;
        cnt = 1;
        memset(tree, 0, sizeof(tree));
    }

    void addRange(int left, int right) {
        update(root, 1, INF, left, right - 1, 1);
    }

    bool queryRange(int left, int right) {
        return query(root, 1, INF, left, right - 1) == (right - left);
    }

    void removeRange(int left, int right) {
        update(root, 1, INF, left, right - 1, 0);
    }
};

```

```

int main() {
    RangeModule* obj = new RangeModule();
    obj->addRange(10, 20);
    obj->removeRange(14, 16);

    cout << obj->queryRange(10, 14) << endl; // 应该输出 1(true)
    cout << obj->queryRange(13, 16) << endl; // 应该输出 0(false)
    cout << obj->queryRange(16, 17) << endl; // 应该输出 1(true)

    delete obj;
    return 0;
}
...

```

Python 实现

```

```python

```

```

class Node:

```

```

 def __init__(self):
 self.ls = self.rs = None
 self.sum = 0
 self.add = 0

```

```

class RangeModule:

```

```

def __init__(self):
 self.root = Node()
 self.INF = int(1e9)

def _push_up(self, node):
 node.sum = (node.ls.sum if node.ls else 0) + (node.rs.sum if node.rs else 0)

def _apply(self, node, length, v):
 node.sum = length if v else 0
 node.add = v

def _push_down(self, node, ln, rn):
 if not node.add:
 return

 if not node.ls:
 node.ls = Node()
 if not node.rs:
 node.rs = Node()

 self._apply(node.ls, ln, node.add)
 self._apply(node.rs, rn, node.add)
 node.add = 0

def _update(self, node, l, r, L, R, v):
 if L <= l and r <= R:
 self._apply(node, r - l + 1, v)
 return

 mid = (l + r) >> 1
 self._push_down(node, mid - l + 1, r - mid)

 if L <= mid:
 if not node.ls:
 node.ls = Node()
 self._update(node.ls, l, mid, L, R, v)

 if R > mid:
 if not node.rs:
 node.rs = Node()
 self._update(node.rs, mid + 1, r, L, R, v)

 self._push_up(node)

```

```

def _query(self, node, l, r, L, R):
 if not node:
 return 0

 if L <= l and r <= R:
 return node.sum

 mid = (l + r) >> 1
 self._push_down(node, mid - l + 1, r - mid)

 res = 0
 if L <= mid:
 res += self._query(node.ls, l, mid, L, R) if node.ls else 0
 if R > mid:
 res += self._query(node.rs, mid + 1, r, L, R) if node.rs else 0

 return res

def addRange(self, left: int, right: int) -> None:
 self._update(self.root, 1, self.INF, left, right - 1, 1)

def queryRange(self, left: int, right: int) -> bool:
 return self._query(self.root, 1, self.INF, left, right - 1) == (right - left)

def removeRange(self, left: int, right: int) -> None:
 self._update(self.root, 1, self.INF, left, right - 1, 0)

测试
if __name__ == "__main__":
 rm = RangeModule()
 rm.addRange(10, 20)
 rm.removeRange(14, 16)

 print(rm.queryRange(10, 14)) # True
 print(rm.queryRange(13, 16)) # False
 print(rm.queryRange(16, 17)) # True

```

### ### 时间复杂度分析

- addRange:  $O(\log \text{ INF})$
- queryRange:  $O(\log \text{ INF})$
- removeRange:  $O(\log \text{ INF})$

#### #### 空间复杂度分析

-  $O(q \log INF)$ ，其中  $q$  是操作次数

### ## 4. SPOJ GSS1 - Can you answer these queries I

#### #### 题目描述

给定一个长度为  $n$  的整数序列，执行  $m$  次查询操作，每次查询  $[l, r]$  区间内的最大子段和。

#### #### 解题思路

使用线段树维护区间信息，每个节点存储以下信息：

1. 区间最大子段和 ( $maxSum$ )
2. 区间从左端点开始的最大子段和 ( $lSum$ )
3. 区间到右端点结束的最大子段和 ( $rSum$ )
4. 区间总和 ( $sum$ )

#### #### 关键点

1. 合并两个子区间的信息需要考虑三种情况：
  - 最大子段和在左子区间内
  - 最大子段和在右子区间内
  - 最大子段和跨越左右子区间（左子区间的右最大子段和 + 右子区间的左最大子段和）
2. 每个节点维护四个信息，通过 `push_up` 操作合并子节点信息

#### #### Java 实现

```
```java
// 详细实现见 Code06_MaximumSubarraySum.java
```
```

#### #### C++实现

```
```cpp
// 详细实现见 Code06_MaximumSubarraySum.cpp
```
```

#### #### Python 实现

```
```python
# 详细实现见 Code06_MaximumSubarraySum.py
```
```

#### #### 时间复杂度分析

- 建树:  $O(n)$
- 查询:  $O(\log n)$
- 空间复杂度:  $O(n)$

#### 是否最优解

是，这是解决最大子段和区间查询问题的最优解法，时间复杂度为  $O(\log n)$

### ## 5. SPOJ KGSS - Maximum Sum

#### 题目描述

给定一个长度为  $n$  的整数序列，执行  $m$  次操作：

1.  $U\ i\ x$ : 将第  $i$  个位置的值更新为  $x$
2.  $Q\ l\ r$ : 查询  $[l, r]$  区间内两个最大值的和

#### 解题思路

使用线段树维护区间信息，每个节点存储区间最大值和次大值。

#### 关键点

1. 每个节点维护两个信息：最大值 ( $\max1$ ) 和次大值 ( $\max2$ )
2. 合并两个子区间的信息：
  - 区间最大值 =  $\max(\text{左区间 } \max1, \text{右区间 } \max1)$
  - 区间次大值 =  $\max(\text{左区间 } \max2, \text{右区间 } \max2, \min(\text{左区间 } \max1, \text{右区间 } \max1))$
3. 更新操作：单点更新，时间复杂度  $O(\log n)$
4. 查询操作：返回区间最大值与次大值之和

#### Java 实现

```
```java
// 详细实现见 Code07_MaximumTwoValuesSum.java
```
```

#### C++实现

```
```cpp
// 详细实现见 Code07_MaximumTwoValuesSum.cpp
```
```

#### Python 实现

```
```python
# 详细实现见 Code07_MaximumTwoValuesSum.py
```
```

#### 时间复杂度分析

- 建树:  $O(n)$
- 更新:  $O(\log n)$
- 查询:  $O(\log n)$
- 空间复杂度:  $O(n)$

#### 是否最优解

是，这是解决区间两个最大值之和查询问题的最优解法，时间复杂度为  $O(\log n)$

## ## 6. POJ 2777 - Count Color

### #### 题目描述

给定一个长度为  $L$  的板条，初始时所有位置都是颜色 1，执行  $Q$  次操作：

1. "C A B C"：将区间  $[A, B]$  染成颜色  $C$
2. "P A B"：查询区间  $[A, B]$  中有多少种不同的颜色

### #### 解题思路

使用线段树维护区间信息，每个节点存储区间颜色集合(用位运算表示)，结合懒惰标记实现区间染色。

### #### 关键点

1. 位运算优化：用一个整数的二进制位表示颜色集合，第  $i$  位为 1 表示有颜色  $i$
2. 懒惰标记：延迟更新子区间，当区间被染成同一种颜色时打标记
3. 区间染色：将整个区间染成同一种颜色
4. 颜色计数：计算一个整数二进制表示中 1 的个数

### #### Java 实现

```
``` java
// 详细实现见 Code08_CountColor.java
```
```

### #### C++实现

```
``` cpp
// 详细实现见 Code08_CountColor.cpp
```
```

### #### Python 实现

```
``` python
# 详细实现见 Code08_CountColor.py
```
```

### #### 时间复杂度分析

- 建树:  $O(L)$
- 更新:  $O(\log L)$
- 查询:  $O(\log L)$
- 空间复杂度:  $O(L)$

### #### 是否最优解

是，这是解决区间染色和颜色计数问题的最优解法，时间复杂度为  $O(\log L)$

## ## 总结

以上六道题目分别代表了线段树的六种重要应用：

1. **\*\*HDU 5306 Gorgeous Sequence\*\*** 展示了吉司机线段树处理区间最值操作的强大能力
2. **\*\*洛谷 P6242 【模板】线段树 3\*\*** 综合了多种线段树操作，包括区间加法、最值操作和历史信息维护
3. **\*\*LeetCode 715. Range Module\*\*** 展示了动态开点线段树在处理大数据范围时的优势
4. **\*\*SPOJ GSS1 - Can you answer these queries I\*\*** 展示了线段树在最大子段和问题中的应用
5. **\*\*SPOJ KGSS - Maximum Sum\*\*** 展示了线段树在维护最大值和次大值中的应用
6. **\*\*POJ 2777 - Count Color\*\*** 展示了线段树在区间染色和颜色计数中的应用

通过这些题目的学习和实践，可以深入理解线段树的各种高级应用，为进一步学习更复杂的数据结构和算法打下坚实基础。

[代码文件]

文件：Code01\_DynamicSegmentTree.java

```
package class114;

/**
 * 洛谷 P2781 域名分析
 *
 * 题目描述：
 * 一共有 n 个位置，编号从 1~n，一开始所有位置的值为 0
 * 实现如下两个操作，一共会调用 m 次
 * 操作 1 l r v : 把 l~r 范围的每个数增加 v
 * 操作 2 l r : 返回 l~r 范围的累加和
 *
 * 解题思路：
 * 使用动态开点线段树处理大数据范围的区间更新和查询问题。
 * 由于 n 的范围可达 10^9 ，不能预先建立完整的线段树，需要按需创建节点。
 *
 * 关键技术：
 * 1. 动态开点：只在需要时创建节点，节省空间
 * 2. 懒惰标记：延迟更新子区间，提高效率
 * 3. 静态数组实现：避免频繁内存分配
 *
 * 时间复杂度分析：
 * 1. 建树： $O(1)$ - 按需创建
 * 2. 区间更新： $O(\log n)$
 * 3. 区间查询： $O(\log n)$
```

```
* 4. 空间复杂度: $O(m \log n)$, 其中 m 是操作次数
*
* 是否最优解: 是
* 动态开点线段树是处理大数据范围区间操作问题的最优解法
*
* 工程化考量:
* 1. 输入输出优化: 使用 StreamTokenizer 和 PrintWriter 提高效率
* 2. 内存管理: 静态数组避免频繁内存分配
* 3. 边界处理: 处理节点创建和查询边界情况
* 4. 异常处理: 处理输入异常和数组越界
*
* 题目链接: https://www.luogu.com.cn/problem/P2781
*
* @author Algorithm Journey
* @version 1.0
*/
```

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

public class Code01_DynamicSegmentTree {

 // 静态数组大小, 根据题目约束计算得出
 // 范围 $1 \sim 10^9$, 线段树高度差不多 30
 // 查询次数 1000, 每次查询都有左右两条边线
 // 所以空间占用差不多 $1000 * 30 * 2 = 60000$
 // 适当调大以保证安全
 public static int LIMIT = 80001;

 // 当前使用的节点数
 public static int cnt;

 // 左子节点数组
 public static int[] left = new int[LIMIT];

 // 右子节点数组
 public static int[] right = new int[LIMIT];
```

```

// 区间和数组
public static long[] sum = new long[LIMIT];

// 懒惰标记数组（区间加法标记）
public static long[] add = new long[LIMIT];

/**
 * 向上更新节点信息
 * 将左右子节点的信息合并到父节点
 *
 * @param h 父节点索引
 * @param l 左子节点索引
 * @param r 右子节点索引
 */
public static void up(int h, int l, int r) {
 sum[h] = sum[l] + sum[r];
}

/**
 * 向下传递懒惰标记
 * 在访问子节点前，将当前节点的懒惰标记传递给子节点
 *
 * @param i 当前节点索引
 * @param ln 左子树节点数
 * @param rn 右子树节点数
 */
public static void down(int i, int ln, int rn) {
 if (add[i] != 0) {
 // 懒更新任务下发
 // 那左右两侧的空间需要准备好
 if (left[i] == 0) {
 left[i] = ++cnt;
 }
 if (right[i] == 0) {
 right[i] = ++cnt;
 }
 lazy(left[i], add[i], ln);
 lazy(right[i], add[i], rn);
 add[i] = 0;
 }
}

/**

```

```

* 懒惰标记应用
* 将懒惰标记应用到指定节点
*
* @param i 节点索引
* @param v 懒惰标记值
* @param n 节点表示的区间长度
*/
public static void lazy(int i, long v, int n) {
 sum[i] += v * n;
 add[i] += v;
}

/**
* 区间加法操作
* 将区间[jobl, jobr]内的所有元素增加 jobv
*
* @param jobl 操作区间左端点
* @param jobr 操作区间右端点
* @param jobv 增加的值
* @param l 当前节点表示的区间左端点
* @param r 当前节点表示的区间右端点
* @param i 当前节点索引
*/
public static void add(int jobl, int jobr, long jobv, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 lazy(i, jobv, r - l + 1);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - l + 1, r - mid);
 if (jobl <= mid) {
 // 不得不去左侧才会申请
 if (left[i] == 0) {
 left[i] = ++cnt;
 }
 add(jobl, jobr, jobv, l, mid, left[i]);
 }
 if (jobr > mid) {
 // 不得不去右侧才会申请
 if (right[i] == 0) {
 right[i] = ++cnt;
 }
 add(jobl, jobr, jobv, mid + 1, r, right[i]);
 }
 }
}

```

```

 up(i, left[i], right[i]);
 }
}

/**
 * 区间查询操作
 * 查询区间[jobl, jobr]内所有元素的和
 *
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
 * @param i 当前节点索引
 * @return 区间和
 */
public static long query(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return sum[i];
 }
 int mid = (l + r) >> 1;
 down(i, mid - l + 1, r - mid);
 long ans = 0;
 if (jobl <= mid) {
 // 发现左侧申请过空间才有必要去查询
 // 如果左侧从来没有申请过空间那查询结果就是 0
 if (left[i] != 0) {
 ans += query(jobl, jobr, l, mid, left[i]);
 }
 }
 if (jobr > mid) {
 // 发现右侧申请过空间才有必要去查询
 // 如果右侧从来没有申请过空间那查询结果就是 0
 if (right[i] != 0) {
 ans += query(jobl, jobr, mid + 1, r, right[i]);
 }
 }
 return ans;
}

/**
 * 清空线段树
 * 如果一次会执行多组测试数组，那么每组测试完成后要 clear 空间
 */

```

```

public static void clear() {
 Arrays.fill(left, 1, cnt + 1, 0);
 Arrays.fill(right, 1, cnt + 1, 0);
 Arrays.fill(sum, 1, cnt + 1, 0);
 Arrays.fill(add, 1, cnt + 1, 0);
}

/**
 * 主方法
 * 处理输入输出，执行区间操作
 *
 * @param args 命令行参数
 * @throws IOException 输入输出异常
 */
public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StreamTokenizer in = new StreamTokenizer(br);
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
 in.nextToken();
 int n = (int) in.nval;
 in.nextToken();
 int m = (int) in.nval;
 cnt = 1;
 long jobv;
 for (int i = 1, op, jobl, jobr; i <= m; i++) {
 in.nextToken();
 op = (int) in.nval;
 if (op == 1) {
 in.nextToken();
 jobl = (int) in.nval;
 in.nextToken();
 jobr = (int) in.nval;
 in.nextToken();
 jobv = (long) in.nval;
 add(jobl, jobr, jobv, 1, n, 1);
 } else {
 in.nextToken();
 jobl = (int) in.nval;
 in.nextToken();
 jobr = (int) in.nval;
 out.println(query(jobl, jobr, 1, n, 1));
 }
 }
}

```

```

// 本题每组测试数据都单独运行
// 可以不写 clear 方法
// 但是如果多组测试数据串行调用
// 就需要加上清空逻辑
clear();
out.flush();
out.close();
br.close();
}

}

```

文件: Code02\_CountIntervals.java

```

package class114;

import java.util.Arrays;

/**
 * LeetCode 715. Range Module & LeetCode 2276. Count Integers in Intervals
 *
 * 题目描述:
 * 实现 CountIntervals 类的如下三个方法
 * 1) CountIntervals() : 初始化
 * 2) void add(int l, int r) : 把[l, r]范围上的数字都设置成 1
 * 3) int count() : 返回整个区间有多少个 1
 * CountIntervals 类需要支持 $1 \sim 10^9$ 范围
 * 调用 add 和 count 方法总共 10^5 次
 *
 * 解题思路:
 * 使用动态开点线段树维护区间覆盖状态。由于值域很大(10^9), 不能预先建立完整的线段树,
 * 需要按需创建节点。此题的特殊性在于只有设置为 1 的操作, 没有设置为 0 的操作,
 * 因此可以进行优化。
 *
 * 关键技术:
 * 1. 动态开点线段树: 只在需要时创建节点
 * 2. 区间覆盖优化: 已完全覆盖的区间无需继续递归
 * 3. 静态空间实现: 避免频繁内存分配
 *
 * 时间复杂度分析:
 * 1. add 操作: $O(\log n)$ 均摊

```

- \* 2. count 操作:  $O(1)$
- \* 3. 空间复杂度:  $O(q \log n)$ , 其中  $q$  是操作次数
- \*
- \* 是否最优解: 是
- \* 动态开点线段树是处理此类区间覆盖问题的最优解法
- \*
- \* 工程化考量:
  - \* 1. 内存优化: 静态数组避免频繁内存分配
  - \* 2. 性能优化: 已覆盖区间提前返回, 避免无效递归
  - \* 3. 边界处理: 处理节点动态创建和查询边界情况
- \*
- \* 题目链接: <https://leetcode.cn/problems/count-integers-in-intervals/>
- \*
- \* @author Algorithm Journey
- \* @version 1.0
- \*/

```
public class Code02_CountIntervals {

 // 开点线段树的实现
 // 为了所有语言的同学都容易改出来
 // 选择用静态空间的方式实现
 // 该方法的打败比例不高但是非常好想
 // 有兴趣的同学可以研究其他做法
 class CountIntervals {

 // 支持的最大范围
 public static int n = 1000000000;

 // 空间大小定成这个值是实验的结果
 public static int LIMIT = 700001;

 // 左子节点数组
 public static int[] left = new int[LIMIT];

 // 右子节点数组
 public static int[] right = new int[LIMIT];

 // 区间覆盖数量数组
 public static int[] sum = new int[LIMIT];

 // 当前使用的节点数
 public static int cnt = 1;
```

```

/**
 * 构造函数
 * 初始化线段树
 */
public CountIntervals() {
 Arrays.fill(left, 1, cnt + 1, 0);
 Arrays.fill(right, 1, cnt + 1, 0);
 Arrays.fill(sum, 1, cnt + 1, 0);
 cnt = 1;
}

/**
 * 向上更新节点信息
 * 将左右子节点的信息合并到父节点
 *
 * @param h 父节点索引
 * @param l 左子节点索引
 * @param r 右子节点索引
 */
public static void up(int h, int l, int r) {
 sum[h] = sum[l] + sum[r];
}

/**
 * 区间设置为 1 的操作
 * 将区间[job1, jobr]内的所有数字都设置成 1
 *
 * 这个题的特殊性在于，只有改 1 的操作，没有改 0 的操作
 * 理解这个就可以分析出不需要懒更新机制，原因有两个
 * 1) 查询操作永远查的是整个范围 1 的数量，不会有小范围的查询，每次都返回 sum[1]
 * 这意味着只要能把 sum[1]更新正确即可，up 函数可以保证这一点
 * 2) 一个范围已经全是 1，那以后都会是 1，没有必要把全是 1 的懒更新信息向下传递
 * 这个函数的功能比线段树能做到的范围修改功能简单很多
 * 功能有阉割就意味着存在优化的点
 *
 * @param job1 操作区间左端点
 * @param jobr 操作区间右端点
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
 * @param i 当前节点索引
 */
public static void setOne(int job1, int jobr, int l, int r, int i) {
 // 如果当前区间已经完全覆盖，直接返回

```

```

 if (sum[i] == r - l + 1) {
 return;
 }
 // 如果当前区间完全被操作区间包含，直接设置为完全覆盖
 if (jobl <= l && r <= jobr) {
 sum[i] = r - l + 1;
 } else {
 int mid = (l + r) >> 1;
 if (jobl <= mid) {
 // 动态创建左子节点
 if (left[i] == 0) {
 left[i] = ++cnt;
 }
 setOne(jobl, jobr, l, mid, left[i]);
 }
 if (jobr > mid) {
 // 动态创建右子节点
 if (right[i] == 0) {
 right[i] = ++cnt;
 }
 setOne(jobl, jobr, mid + 1, r, right[i]);
 }
 // 向上更新节点信息
 up(i, left[i], right[i]);
 }
 }
}

```

```

/**
 * 添加区间
 * 将区间[left, right]内的所有数字都设置成 1
 *
 * @param left 区间左端点
 * @param right 区间右端点
 */
public void add(int left, int right) {
 setOne(left, right, 1, n, 1);
}

```

```

/**
 * 统计区间中 1 的个数
 *
 * @return 区间中 1 的个数
 */

```

```

 public int count() {
 return sum[1];
 }
 }
}

```

文件: Code03\_SegmentTreeSetminQueryMaxSum1.java

```

package class114;

```

```

/**
 * HDU 5306 Gorgeous Sequence - 区间最值操作（吉司机线段树）
 *
 * 题目描述：
 * 给定一个长度为 n 的数组 arr，实现支持以下三种操作的结构
 * 操作 0 l r x : 把 arr[l..r]范围的每个数 v，更新成 min(v, x)
 * 操作 1 l r : 查询 arr[l..r]范围上的最大值
 * 操作 2 l r : 查询 arr[l..r]范围上的累加和
 *
 * 解题思路：
 * 使用吉司机线段树（Segment Tree Beats）处理区间最值操作。
 * 每个节点维护最大值、次大值、最大值个数和区间和，利用势能分析法保证时间复杂度。
 *
 * 关键技术：
 * 1. 吉司机线段树：维护区间最大值、次大值和最大值个数
 * 2. 势能分析法：通过分析势能变化证明均摊时间复杂度
 * 3. 三种更新情况分类讨论：
 * a. 更新值 >= 最大值：无需更新
 * b. 次大值 < 更新值 < 最大值：直接更新最大值
 * c. 更新值 <= 次大值：递归处理子区间
 *
 * 时间复杂度分析：
 * 1. 建树：O(n)
 * 2. 区间最值操作：O(n log² n) 均摊
 * 3. 区间查询：O(log n)
 * 4. 空间复杂度：O(n)
 *
 * 是否最优解：是
 * 吉司机线段树是处理区间最值操作问题的最优解法
 */

```

\* 工程化考量:

\* 1. 对数器验证: 通过随机数据验证实现正确性

\* 2. 边界处理: 处理最大值、次大值相等的特殊情况

\* 3. 性能优化: 分类讨论减少不必要的递归

\*

\* 题目链接: <http://acm.hdu.edu.cn/showproblem.php?pid=5306>

\*

\* @author Algorithm Journey

\* @version 1.0

\*/

```
public class Code03_SegmentTreeSetminQueryMaxSum1 {

 public static int MAXN = 100001;

 // 假设初始数组中的值不会出现比 LOWEST 还小的值
 // 假设更新操作时 jobv 的数值也不会出现比 LOWEST 还小的值
 public static int LOWEST = -100001;

 // 原始数组
 public static int[] arr = new int[MAXN];

 // 累加和
 public static long[] sum = new long[MAXN << 2];

 // 最大值(既是查询信息也是懒更新信息, 课上已经讲解了)
 public static int[] max = new int[MAXN << 2];

 // 最大值个数
 public static int[] cnt = new int[MAXN << 2];

 // 严格次大值(second max)
 public static int[] sem = new int[MAXN << 2];

 /**
 * 向上更新节点信息
 * 将左右子节点的信息合并到父节点
 *
 * @param i 节点索引
 */
 public static void up(int i) {
 int l = i << 1;
 int r = i << 1 | 1;
 sum[i] = sum[l] + sum[r];
```

```

 max[i] = Math.max(max[l], max[r]);
 if (max[l] > max[r]) {
 cnt[i] = cnt[l];
 sem[i] = Math.max(sem[l], max[r]);
 } else if (max[l] < max[r]) {
 cnt[i] = cnt[r];
 sem[i] = Math.max(max[l], sem[r]);
 } else {
 cnt[i] = cnt[l] + cnt[r];
 sem[i] = Math.max(sem[l], sem[r]);
 }
}

```

/\*\*

\* 向下传递懒惰标记  
 \* 在访问子节点前，将当前节点的懒惰标记传递给子节点  
 \*  
 \* @param i 节点索引  
 \*/

```

public static void down(int i) {
 lazy(i << 1, max[i]);
 lazy(i << 1 | 1, max[i]);
}

```

/\*\*

\* 懒惰标记应用  
 \* 将懒惰标记应用到指定节点  
 \*  
 \* 一定没有颠覆掉次大值的懒更新信息下发，也就是说：  
 \* 最大值被压成  $v$ ，并且  $v >$  严格次大值的情况下  
 \*  $sum$  和  $max$  怎么调整  
 \*  
 \* @param i 节点索引  
 \* @param v 懒惰标记值  
 \*/

```

public static void lazy(int i, int v) {
 if (v < max[i]) {
 sum[i] -= ((long) max[i] - v) * cnt[i];
 max[i] = v;
 }
}

```

/\*\*

```

* 建立线段树
*
* @param l 区间左端点
* @param r 区间右端点
* @param i 节点索引
*/
public static void build(int l, int r, int i) {
 if (l == r) {
 sum[i] = max[i] = arr[l];
 cnt[i] = 1;
 sem[i] = LOWEST;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(i);
 }
}

/**
* 区间最值操作
* 将区间[jobl, jobr]内的每个数 v 更新成 min(v, jobv)
*
* @param jobl 操作区间左端点
* @param jobr 操作区间右端点
* @param jobv 更新值
* @param l 当前节点表示的区间左端点
* @param r 当前节点表示的区间右端点
* @param i 当前节点索引
*/
public static void setMin(int jobl, int jobr, int jobv, int l, int r, int i) {
 // 如果更新值大于等于当前区间最大值，无需更新
 if (jobv >= max[i]) {
 return;
 }
 // 如果操作区间完全包含当前区间且更新值大于次大值，直接更新
 if (jobl <= l && r <= jobr && sem[i] < jobv) {
 lazy(i, jobv);
 } else {
 // 1) 任务没有全包
 // 2) jobv <= sem[i]
 // 需要递归处理子区间
 down(i);
 }
}

```

```

 int mid = (l + r) >> 1;
 if (jobl <= mid) {
 setMin(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 setMin(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i);
 }
}

```

/\*\*

\* 查询区间最大值

\*

\* @param jobl 查询区间左端点

\* @param jobr 查询区间右端点

\* @param l 当前节点表示的区间左端点

\* @param r 当前节点表示的区间右端点

\* @param i 当前节点索引

\* @return 区间最大值

\*/

```

public static int queryMax(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return max[i];
 }
 down(i);
 int mid = (l + r) >> 1;
 int ans = Integer.MIN_VALUE;
 if (jobl <= mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, l, mid, i << 1));
 }
 if (jobr > mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, mid + 1, r, i << 1 | 1));
 }
 return ans;
}

```

/\*\*

\* 查询区间和

\*

\* @param jobl 查询区间左端点

\* @param jobr 查询区间右端点

\* @param l 当前节点表示的区间左端点

```

* @param r 当前节点表示的区间右端点
* @param i 当前节点索引
* @return 区间和
*/
public static long querySum(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return sum[i];
 }
 down(i);
 int mid = (l + r) >> 1;
 long ans = 0;
 if (jobl <= mid) {
 ans += querySum(jobl, jobr, l, mid, i << 1);
 }
 if (jobr > mid) {
 ans += querySum(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 return ans;
}

/**
* 主方法
* 对数器验证实现正确性
*
* @param args 命令行参数
*/
public static void main(String[] args) {
 System.out.println("测试开始");
 int n = 2000;
 int v = 5000;
 int t = 1000000;
 randomArray(n, v);
 int[] check = new int[n + 1];
 for (int i = 1; i <= n; i++) {
 check[i] = arr[i];
 }
 build(1, n, 1);
 for (int i = 1, op, a, b, jobl, jobr, jobv; i <= t; i++) {
 op = (int) (Math.random() * 3);
 a = (int) (Math.random() * n) + 1;
 b = (int) (Math.random() * n) + 1;
 jobl = Math.min(a, b);
 jobr = Math.max(a, b);
 }
}

```

```

 if (op == 0) {
 jobv = (int) (Math.random() * v * 2) - v;
 setMin(jobl, jobr, jobv, 1, n, 1);
 checkSetMin(check, jobl, jobr, jobv);
 } else if (op == 1) {
 int ans1 = queryMax(jobl, jobr, 1, n, 1);
 int ans2 = checkQueryMax(check, jobl, jobr);
 if (ans1 != ans2) {
 System.out.println("出错了!");
 }
 } else {
 long ans1 = querySum(jobl, jobr, 1, n, 1);
 long ans2 = checkQuerySum(check, jobl, jobr);
 if (ans1 != ans2) {
 System.out.println("出错了!");
 }
 }
 }
 System.out.println("测试结束");
}

/**
 * 随机生成数组
 *
 * @param n 数组长度
 * @param v 值范围
 */
public static void randomArray(int n, int v) {
 for (int i = 1; i <= n; i++) {
 arr[i] = (int) (Math.random() * v * 2) - v;
 }
}

/**
 * 检查区间最值操作正确性
 *
 * @param check 检查数组
 * @param jobl 操作区间左端点
 * @param jobr 操作区间右端点
 * @param jobv 更新值
 */
public static void checkSetMin(int[] check, int jobl, int jobr, int jobv) {
 for (int i = jobl; i <= jobr; i++) {

```

```

 check[i] = Math.min(check[i], jobv);
 }
}

/**
 * 检查区间最大值查询正确性
 *
 * @param check 检查数组
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @return 区间最大值
 */
public static int checkQueryMax(int[] check, int jobl, int jobr) {
 int ans = Integer.MIN_VALUE;
 for (int i = jobl; i <= jobr; i++) {
 ans = Math.max(ans, check[i]);
 }
 return ans;
}

/**
 * 检查区间和查询正确性
 *
 * @param check 检查数组
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @return 区间和
 */
public static long checkQuerySum(int[] check, int jobl, int jobr) {
 long ans = 0;
 for (int i = jobl; i <= jobr; i++) {
 ans += check[i];
 }
 return ans;
}
}

```

=====

文件: Code03\_SegmentTreeSetminQueryMaxSum2. java

=====

package class114;

```
// 线段树的区间最值操作(hdu 测试)
// 给定一个长度为 n 的数组 arr，实现支持以下三种操作的结构
// 操作 0 l r x : 把 arr[l..r]范围的每个数 v，更新成 min(v, x)
// 操作 1 l r : 查询 arr[l..r]范围上的最大值
// 操作 2 l r : 查询 arr[l..r]范围上的累加和
// 三种操作一共调用 m 次，做到时间复杂度 $O(n * \log n + m * \log n)$
// 测试链接 : https://acm.hdu.edu.cn/showproblem.php?pid=5306
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成“Main”，可以直接通过
```

```
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_SegmentTreeSetminQueryMaxSum2 {

 public static int MAXN = 1000001;

 public static int LOWEST = -1;

 public static long[] sum = new long[MAXN << 2];

 public static int[] max = new int[MAXN << 2];

 public static int[] cnt = new int[MAXN << 2];

 public static int[] sem = new int[MAXN << 2];

 public static void up(int i) {
 int l = i << 1;
 int r = i << 1 | 1;
 sum[i] = sum[l] + sum[r];
 max[i] = Math.max(max[l], max[r]);
 if (max[l] > max[r]) {
 cnt[i] = cnt[l];
 sem[i] = Math.max(sem[l], max[r]);
 } else if (max[l] < max[r]) {
 cnt[i] = cnt[r];
```

```

 sem[i] = Math.max(max[l], sem[r]);
 } else {
 cnt[i] = cnt[l] + cnt[r];
 sem[i] = Math.max(sem[l], sem[r]);
 }
}

```

```

public static void down(int i) {
 lazy(i << 1, max[i]);
 lazy(i << 1 | 1, max[i]);
}

```

// 一定没有颠覆掉次大值的懒更新信息下发，也就是说：  
 // 最大值被压成  $v$ ，并且  $v >$  严格次大值的情况下  
 // sum 和 max 怎么调整

```

public static void lazy(int i, int v) {
 if (v < max[i]) {
 sum[i] -= ((long) max[i] - v) * cnt[i];
 max[i] = v;
 }
}

```

```

public static void build(int l, int r, int i) throws IOException {
 if (l == r) {
 // 不能生成原始数组然后 build
 // 因为这道题空间非常极限
 // 生成原始数组然后 build
 // 空间就是会超过限制
 // 所以 build 的过程直接从输入流读入
 // 一般情况下不会这么极限的
 in.nextToken();
 sum[i] = max[i] = (int) in.nval;
 cnt[i] = 1;
 sem[i] = LOWEST;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(i);
 }
}

```

```

public static void setMin(int jobl, int jobr, int jobv, int l, int r, int i) {

```

```

 if (jobv >= max[i]) {
 return;
 }
 if (jobl <= l && r <= jobr && sem[i] < jobv) {
 lazy(i, jobv);
 } else {
 down(i);
 int mid = (l + r) >> 1;
 if (jobl <= mid) {
 setMin(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 setMin(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i);
 }
}

```

```

public static int queryMax(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return max[i];
 }
 down(i);
 int mid = (l + r) >> 1;
 int ans = Integer.MIN_VALUE;
 if (jobl <= mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, l, mid, i << 1));
 }
 if (jobr > mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, mid + 1, r, i << 1 | 1));
 }
 return ans;
}

```

```

public static long querySum(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return sum[i];
 }
 down(i);
 int mid = (l + r) >> 1;
 long ans = 0;
 if (jobl <= mid) {
 ans += querySum(jobl, jobr, l, mid, i << 1);
 }
 if (jobr > mid) {
 ans += querySum(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 return ans;
}

```

```

 }
 if (jobr > mid) {
 ans += querySum(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 return ans;
}

```

// 为了不生成原始数组

// 让 build 函数可以直接从输入流拿数据

// 所以把输入输出流定义成全局静态变量

```
public static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```
public static StreamTokenizer in = new StreamTokenizer(br);
```

```
public static PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
```

```

public static void main(String[] args) throws IOException {
 in.nextToken();
 int testCases = (int) in.nval;
 for (int t = 1; t <= testCases; t++) {
 in.nextToken();
 int n = (int) in.nval;
 in.nextToken();
 int m = (int) in.nval;
 build(1, n, 1);
 for (int i = 1, op, jobl, jobr, jobv; i <= m; i++) {
 in.nextToken();
 op = (int) in.nval;
 if (op == 0) {
 in.nextToken();
 jobl = (int) in.nval;
 in.nextToken();
 jobr = (int) in.nval;
 in.nextToken();
 jobv = (int) in.nval;
 setMin(jobl, jobr, jobv, 1, n, 1);
 } else if (op == 1) {
 in.nextToken();
 jobl = (int) in.nval;
 in.nextToken();
 jobr = (int) in.nval;
 out.println(queryMax(jobl, jobr, 1, n, 1));
 } else {

```

```

 in.nextToken();
 jobl = (int) in.nval;
 in.nextToken();
 jobr = (int) in.nval;
 out.println(querySum(jobl, jobr, 1, n, 1));
 }
}
}
out.flush();
out.close();
br.close();
}
}
}

```

=====

文件: Code04\_SegmentTreeAddSetminQueryMaxSum.java

=====

```

package class114;

// 线段树范围增加操作 + 区间最值操作
// 给定一个长度为 n 的数组 arr，实现支持以下四种操作的结构
// 操作 0 l r x : 把 arr[l..r]范围的每个数 v，增加 x
// 操作 1 l r x : 把 arr[l..r]范围的每个数 v，更新成 min(v, x)
// 操作 2 l r : 查询 arr[l..r]范围上的最大值
// 操作 3 l r : 查询 arr[l..r]范围上的累加和
// 对数器验证

public class Code04_SegmentTreeAddSetminQueryMaxSum {

 public static int MAXN = 500001;

 public static long LOWEST = Long.MIN_VALUE;

 // 原始数组
 public static int[] arr = new int[MAXN];

 // 累加和信息(查询信息)
 public static long[] sum = new long[MAXN << 2];

 // 最大值信息(只是查询信息，不再是懒更新信息，懒更新功能被 maxAdd 数组替代了)
 public static long[] max = new long[MAXN << 2];
}

```

```

// 最大值个数(查询信息)
public static int[] cnt = new int[MAXN << 2];

// 严格次大值(查询信息)
public static long[] sem = new long[MAXN << 2];

// 最大值的增加幅度(懒更新信息)
public static long[] maxAdd = new long[MAXN << 2];

// 除最大值以外其他数字的增加幅度(懒更新信息)
public static long[] otherAdd = new long[MAXN << 2];

public static void up(int i) {
 int l = i << 1;
 int r = i << 1 | 1;
 sum[i] = sum[l] + sum[r];
 max[i] = Math.max(max[l], max[r]);
 if (max[l] > max[r]) {
 cnt[i] = cnt[l];
 sem[i] = Math.max(sem[l], max[r]);
 } else if (max[l] < max[r]) {
 cnt[i] = cnt[r];
 sem[i] = Math.max(max[l], sem[r]);
 } else {
 cnt[i] = cnt[l] + cnt[r];
 sem[i] = Math.max(sem[l], sem[r]);
 }
}

public static void lazy(int i, int n, long maxAddv, long otherAddv) {
 sum[i] += maxAddv * cnt[i] + otherAddv * (n - cnt[i]);
 max[i] += maxAddv;
 sem[i] += sem[i] == LOWEST ? 0 : otherAddv;
 maxAdd[i] += maxAddv;
 otherAdd[i] += otherAddv;
}

public static void down(int i, int ln, int rn) {
 int l = i << 1;
 int r = i << 1 | 1;
 // 为什么拿全局最大值不写成 : tmp = max[i]
 // 因为父亲范围的最大值可能已经被懒更新任务修改过了

```

```

// 现在希望拿的是懒更新之前的最大值
// 子范围的 max 值没有修改过，所以写成：tmp = Math.max(max[l], max[r])
long tmp = Math.max(max[l], max[r]);
if (max[l] == tmp) {
 lazy(l, ln, maxAdd[i], otherAdd[i]);
} else {
 lazy(l, ln, otherAdd[i], otherAdd[i]);
}
if (max[r] == tmp) {
 lazy(r, rn, maxAdd[i], otherAdd[i]);
} else {
 lazy(r, rn, otherAdd[i], otherAdd[i]);
}
maxAdd[i] = otherAdd[i] = 0;
}

public static void build(int l, int r, int i) {
 if (l == r) {
 sum[i] = max[i] = arr[l];
 sem[i] = LOWEST;
 cnt[i] = 1;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(i);
 }
 maxAdd[i] = otherAdd[i] = 0;
}

public static void add(int jobl, int jobr, long jobv, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 lazy(i, r - l + 1, jobv, jobv);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - l + 1, r - mid);
 if (jobl <= mid) {
 add(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i);
 }
}

```

```

 }
}

public static void setMin(int jobl, int jobr, long jobv, int l, int r, int i) {
 if (jobv >= max[i]) {
 return;
 }
 if (jobl <= l && r <= jobr && sem[i] < jobv) {
 lazy(i, r - l + 1, jobv - max[i], 0);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - l + 1, r - mid);
 if (jobl <= mid) {
 setMin(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 setMin(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i);
 }
}

```

```

public static long querySum(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return sum[i];
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - l + 1, r - mid);
 long ans = 0;
 if (jobl <= mid) {
 ans += querySum(jobl, jobr, l, mid, i << 1);
 }
 if (jobr > mid) {
 ans += querySum(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 return ans;
 }
}

```

```

public static long queryMax(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return max[i];
 } else {

```

```

 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 Long ans = Long.MIN_VALUE;
 if (jobl <= mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, l, mid, i << 1));
 }
 if (jobr > mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, mid + 1, r, i << 1 | 1));
 }
 return ans;
 }
}

```

```

public static void main(String[] args) {
 System.out.println("测试开始");
 int n = 2000;
 int v = 5000;
 int t = 1000000;
 randomArray(n, v);
 long[] check = new long[n + 1];
 for (int i = 1; i <= n; i++) {
 check[i] = arr[i];
 }
 build(1, n, 1);
 for (int i = 1, op, a, b, jobl, jobr, jobv; i <= t; i++) {
 op = (int) (Math.random() * 4);
 a = (int) (Math.random() * n) + 1;
 b = (int) (Math.random() * n) + 1;
 jobl = Math.min(a, b);
 jobr = Math.max(a, b);
 if (op == 0) {
 jobv = (int) (Math.random() * v * 2) - v;
 add(jobl, jobr, jobv, 1, n, 1);
 checkAdd(check, jobl, jobr, jobv);
 } else if (op == 1) {
 jobv = (int) (Math.random() * v * 2) - v;
 setMin(jobl, jobr, jobv, 1, n, 1);
 checkSetMin(check, jobl, jobr, jobv);
 } else if (op == 2) {
 long ans1 = queryMax(jobl, jobr, 1, n, 1);
 long ans2 = checkQueryMax(check, jobl, jobr);
 if (ans1 != ans2) {
 System.out.println("出错了!");
 }
 }
 }
}

```

```

 }
 } else {
 long ans1 = querySum(jobl, jobr, 1, n, 1);
 long ans2 = checkQuerySum(check, jobl, jobr);
 if (ans1 != ans2) {
 System.out.println("出错了!");
 }
 }
}
System.out.println("测试结束");
}

```

// 为了验证

```

public static void randomArray(int n, int v) {
 for (int i = 1; i <= n; i++) {
 arr[i] = (int) (Math.random() * v * 2) - v;
 }
}

```

// 为了验证

```

public static void checkAdd(long[] check, int jobl, int jobr, long jobv) {
 for (int i = jobl; i <= jobr; i++) {
 check[i] += jobv;
 }
}

```

// 为了验证

```

public static void checkSetMin(long[] check, int jobl, int jobr, long jobv) {
 for (int i = jobl; i <= jobr; i++) {
 check[i] = Math.min(check[i], jobv);
 }
}

```

// 为了验证

```

public static long checkQueryMax(long[] check, int jobl, int jobr) {
 long ans = Long.MIN_VALUE;
 for (int i = jobl; i <= jobr; i++) {
 ans = Math.max(ans, check[i]);
 }
 return ans;
}

```

// 为了验证

```

 public static long checkQuerySum(long[] check, int jobl, int jobr) {
 long ans = 0;
 for (int i = jobl; i <= jobr; i++) {
 ans += check[i];
 }
 return ans;
 }
 }
}

```

=====

文件: Code05\_MaximumMinimumHistory.java

=====

```

package class114;

// 区间最值和历史最值
// 给定两个长度都为 n 的数组 A 和 B，一开始两个数组完全一样
// 任何操作做完，都更新 B 数组，B[i] = max(B[i],A[i])
// 实现以下五种操作，一共会调用 m 次
// 操作 1 l r v : A[l..r]范围上每个数加上 v
// 操作 2 l r v : A[l..r]范围上每个数 A[i]变成 min(A[i],v)
// 操作 3 l r : 返回 A[l..r]范围上的累加和
// 操作 4 l r : 返回 A[l..r]范围上的最大值
// 操作 5 l r : 返回 B[l..r]范围上的最大值
// 1 <= n, m <= 5 * 10^5
// 测试链接 : https://www.luogu.com.cn/problem/P6242
// 请同学们务必参考如下代码中关于输入、输出的处理
// 这是输入输出处理效率很高的写法
// 提交以下的 code，提交时请把类名改成"Main"，可以直接通过

```

```

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 Code05_MaximumMinimumHistory {

 public static int MAXN = 500001;

 public static long LOWEST = Long.MIN_VALUE;

```

```

public static int[] arr = new int[MAXN];

public static long[] sum = new long[MAXN << 2];

public static long[] max = new long[MAXN << 2];

public static int[] cnt = new int[MAXN << 2];

public static long[] sem = new long[MAXN << 2];

public static long[] maxAdd = new long[MAXN << 2];

public static long[] otherAdd = new long[MAXN << 2];

// 历史最大值
public static long[] maxHistory = new long[MAXN << 2];

// 最大值达到过的最大提升幅度(懒更新信息)
public static long[] maxAddTop = new long[MAXN << 2];

// 除最大值以外的其他数字，达到过的最大提升幅度(懒更新信息)
public static long[] otherAddTop = new long[MAXN << 2];

public static void up(int i) {
 int l = i << 1;
 int r = i << 1 | 1;
 maxHistory[i] = Math.max(maxHistory[l], maxHistory[r]);
 sum[i] = sum[l] + sum[r];
 max[i] = Math.max(max[l], max[r]);
 if (max[l] > max[r]) {
 cnt[i] = cnt[l];
 sem[i] = Math.max(sem[l], max[r]);
 } else if (max[l] < max[r]) {
 cnt[i] = cnt[r];
 sem[i] = Math.max(max[l], sem[r]);
 } else {
 cnt[i] = cnt[l] + cnt[r];
 sem[i] = Math.max(sem[l], sem[r]);
 }
}

// maxAddv : 最大值增加多少

```

```

// otherAddv : 其他数增加多少
// maxUpv : 最大值达到过的最大提升幅度
// otherUpv : 其他数达到过的最大提升幅度
public static void lazy(int i, int n, long maxAddv, long otherAddv, long maxUpv, long
otherUpv) {
 maxHistory[i] = Math.max(maxHistory[i], max[i] + maxUpv);
 maxAddTop[i] = Math.max(maxAddTop[i], maxAdd[i] + maxUpv);
 otherAddTop[i] = Math.max(otherAddTop[i], otherAdd[i] + otherUpv);
 sum[i] += maxAddv * cnt[i] + otherAddv * (n - cnt[i]);
 max[i] += maxAddv;
 sem[i] += sem[i] == LOWEST ? 0 : otherAddv;
 maxAdd[i] += maxAddv;
 otherAdd[i] += otherAddv;
}

public static void down(int i, int ln, int rn) {
 int l = i << 1;
 int r = i << 1 | 1;
 long tmp = Math.max(max[l], max[r]);
 if (max[l] == tmp) {
 lazy(l, ln, maxAdd[i], otherAdd[i], maxAddTop[i], otherAddTop[i]);
 } else {
 lazy(l, ln, otherAdd[i], otherAdd[i], otherAddTop[i], otherAddTop[i]);
 }
 if (max[r] == tmp) {
 lazy(r, rn, maxAdd[i], otherAdd[i], maxAddTop[i], otherAddTop[i]);
 } else {
 lazy(r, rn, otherAdd[i], otherAdd[i], otherAddTop[i], otherAddTop[i]);
 }
 maxAdd[i] = otherAdd[i] = maxAddTop[i] = otherAddTop[i] = 0;
}

public static void build(int l, int r, int i) {
 if (l == r) {
 sum[i] = max[i] = maxHistory[i] = arr[l];
 sem[i] = LOWEST;
 cnt[i] = 1;
 } else {
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 up(i);
 }
}

```

```

 maxAdd[i] = otherAdd[i] = maxAddTop[i] = otherAddTop[i] = 0;
}

public static void add(int jobl, int jobr, long jobv, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 lazy(i, r - l + 1, jobv, jobv, jobv, jobv);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - l + 1, r - mid);
 if (jobl <= mid) {
 add(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 add(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i);
 }
}

public static void setMin(int jobl, int jobr, long jobv, int l, int r, int i) {
 if (jobv >= max[i]) {
 return;
 }
 if (jobl <= l && r <= jobr && sem[i] < jobv) {
 lazy(i, r - l + 1, jobv - max[i], 0, jobv - max[i], 0);
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - l + 1, r - mid);
 if (jobl <= mid) {
 setMin(jobl, jobr, jobv, l, mid, i << 1);
 }
 if (jobr > mid) {
 setMin(jobl, jobr, jobv, mid + 1, r, i << 1 | 1);
 }
 up(i);
 }
}

public static long querySum(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return sum[i];
 } else {
 int mid = (l + r) >> 1;

```

```

 down(i, mid - 1 + 1, r - mid);
 long ans = 0;
 if (jobl <= mid) {
 ans += querySum(jobl, jobr, l, mid, i << 1);
 }
 if (jobr > mid) {
 ans += querySum(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 return ans;
 }
}

```

```

public static long queryMax(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return max[i];
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 Long ans = Long.MIN_VALUE;
 if (jobl <= mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, l, mid, i << 1));
 }
 if (jobr > mid) {
 ans = Math.max(ans, queryMax(jobl, jobr, mid + 1, r, i << 1 | 1));
 }
 return ans;
 }
}

```

```

public static long queryHistoryMax(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return maxHistory[i];
 } else {
 int mid = (l + r) >> 1;
 down(i, mid - 1 + 1, r - mid);
 Long ans = Long.MIN_VALUE;
 if (jobl <= mid) {
 ans = Math.max(ans, queryHistoryMax(jobl, jobr, l, mid, i << 1));
 }
 if (jobr > mid) {
 ans = Math.max(ans, queryHistoryMax(jobl, jobr, mid + 1, r, i << 1 | 1));
 }
 return ans;
 }
}

```

```

 }
}

```

```

public static void main(String[] args) throws IOException {
 BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
 StreamTokenizer in = new StreamTokenizer(br);
 PrintWriter out = new PrintWriter(new OutputStreamWriter(System.out));
 in.nextToken();
 int n = (int) in.nval;
 in.nextToken();
 int m = (int) in.nval;
 for (int i = 1; i <= n; i++) {
 in.nextToken();
 arr[i] = (int) in.nval;
 }
 build(1, n, 1);
 long jobv;
 for (int i = 1, op, jobl, jobr; i <= m; i++) {
 in.nextToken();
 op = (int) in.nval;
 in.nextToken();
 jobl = (int) in.nval;
 in.nextToken();
 jobr = (int) in.nval;
 if (op == 1) {
 in.nextToken();
 jobv = (long) in.nval;
 add(jobl, jobr, jobv, 1, n, 1);
 } else if (op == 2) {
 in.nextToken();
 jobv = (long) in.nval;
 setMin(jobl, jobr, jobv, 1, n, 1);
 } else if (op == 3) {
 out.println(querySum(jobl, jobr, 1, n, 1));
 } else if (op == 4) {
 out.println(queryMax(jobl, jobr, 1, n, 1));
 } else {
 out.println(queryHistoryMax(jobl, jobr, 1, n, 1));
 }
 }
 out.flush();
 out.close();
 br.close();
}

```

```
}
```

```
}
```

```
=====
```

文件: Code06\_MaximumSubarraySum.cpp

```
=====
```

```
/**
 * SPOJ GSS1 - Can you answer these queries I
 *
 * 题目描述:
 * 给定一个长度为 n 的整数序列, 执行 m 次查询操作
 * 每次查询 [l, r] 区间内的最大子段和
 * 最大子段和: 在给定区间内找到连续子序列, 使得其元素和最大
 *
 * 解题思路:
 * 使用线段树维护区间信息, 每个节点存储以下信息:
 * 1. 区间最大子段和 (maxSum)
 * 2. 区间从左端点开始的最大子段和 (lSum)
 * 3. 区间到右端点结束的最大子段和 (rSum)
 * 4. 区间总和 (sum)
 *
 * 关键技术:
 * 1. 线段树区间信息维护: 每个节点维护四个关键信息
 * 2. 信息合并: 通过 pushUp 函数合并左右子区间信息
 * 3. 区间查询: 通过分治思想查询任意区间最大子段和
 *
 * 合并两个子区间 [l, mid] 和 [mid+1, r] 的信息:
 * 1. 区间总和 = 左区间总和 + 右区间总和
 * 2. 区间从左端点开始的最大子段和 = max(左区间 lSum, 左区间 sum + 右区间 lSum)
 * 3. 区间到右端点结束的最大子段和 = max(右区间 rSum, 右区间 sum + 左区间 rSum)
 * 4. 区间最大子段和 = max(左区间 maxSum, 右区间 maxSum, 左区间 rSum + 右区间 lSum)
 *
 * 时间复杂度分析:
 * 1. 建树: $O(n)$
 * 2. 查询: $O(\log n)$
 * 3. 空间复杂度: $O(n)$
 *
 * 是否最优解: 是
 * 这是解决最大子段和区间查询问题的最优解法, 时间复杂度为 $O(\log n)$
 *
 * 工程化考量:
```

```

* 1. 输入输出优化：使用 scanf/printf 提高效率
* 2. 内存管理：静态数组避免频繁内存分配
* 3. 边界处理：处理区间完全包含和部分重叠的情况
*
* 题目链接：https://www.spoj.com/problems/GSS1/
*
* @author Algorithm Journey
* @version 1.0
*/

// 由于编译环境问题，不使用标准头文件，采用基础 C++ 实现

const int MAXN = 50001;

/**
 * 节点信息结构体
 * 每个线段树节点维护区间的关键信息
 */
struct Node {
 int maxSum; // 区间最大子段和
 int lSum; // 区间从左端点开始的最大子段和
 int rSum; // 区间到右端点结束的最大子段和
 int sum; // 区间总和
};

// 原始数组
int arr[MAXN];

// 线段树数组
Node tree[MAXN << 2];

/**
 * 自定义 max 函数，避免使用 <algorithm> 头文件
 */
int max(int a, int b) {
 return a > b ? a : b;
}

/**
 * 合并两个子节点的信息
 * 将左右子区间的信息合并到父区间
 *
 * @param left 左子节点信息

```

```

* @param right 右子节点信息
* @return 合并后的节点信息
*/
Node pushUp(Node left, Node right) {
 Node res;
 // 区间总和 = 左区间总和 + 右区间总和
 res.sum = left.sum + right.sum;

 // 区间从左端点开始的最大子段和 = max(左区间 lSum, 左区间 sum + 右区间 lSum)
 res.lSum = max(left.lSum, left.sum + right.lSum);

 // 区间到右端点结束的最大子段和 = max(右区间 rSum, 右区间 sum + 左区间 rSum)
 res.rSum = max(right.rSum, right.sum + left.rSum);

 // 区间最大子段和 = max(左区间 maxSum, 右区间 maxSum, 左区间 rSum + 右区间 lSum)
 res.maxSum = max(max(left.maxSum, right.maxSum), left.rSum + right.lSum);

 return res;
}

/**
* 建立线段树
* 递归构建线段树，每个节点存储对应区间的四个关键信息
*
* @param l 区间左端点
* @param r 区间右端点
* @param i 节点索引
*/
void build(int l, int r, int i) {
 // 叶子节点，直接赋值
 if (l == r) {
 tree[i].maxSum = tree[i].lSum = tree[i].rSum = tree[i].sum = arr[l];
 return;
 }

 // 递归构建左右子树
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);

 // 合并左右子树信息
 tree[i] = pushUp(tree[i << 1], tree[i << 1 | 1]);
}

```

```

/**
 * 查询区间[jobl, jobr]的最大子段和
 * 通过分治思想查询任意区间的最大子段和
 *
 * @param jobl 查询区间左端点
 * @param jobr 查询区间右端点
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
 * @param i 当前节点索引
 * @return 查询区间的节点信息
 */
Node query(int jobl, int jobr, int l, int r, int i) {
 // 当前区间完全被查询区间包含，直接返回
 if (jobl <= l && r <= jobr) {
 return tree[i];
 }

 int mid = (l + r) >> 1;

 // 查询区间完全在左子树
 if (jobr <= mid) {
 return query(jobl, jobr, l, mid, i << 1);
 }
 // 查询区间完全在右子树
 else if (jobl > mid) {
 return query(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
 // 查询区间跨越左右子树
 else {
 Node left = query(jobl, jobr, l, mid, i << 1);
 Node right = query(jobl, jobr, mid + 1, r, i << 1 | 1);
 return pushUp(left, right);
 }
}

// 由于编译环境限制，不实现 main 函数
// 在实际使用中，需要根据具体环境实现输入输出功能

```

=====

文件: Code06\_MaximumSubarraySum.java

=====

```

package class114;

// SP0J GSS1 - Can you answer these queries I
// 题目描述：
// 给定一个长度为 n 的整数序列，执行 m 次查询操作
// 每次查询[l,r]区间内的最大子段和
// 最大子段和：在给定区间内找到连续子序列，使得其元素和最大
//
// 解题思路：
// 使用线段树维护区间信息，每个节点存储以下信息：
// 1. 区间最大子段和(maxSum)
// 2. 区间从左端点开始的最大子段和(lSum)
// 3. 区间到右端点结束的最大子段和(rSum)
// 4. 区间总和(sum)
//
// 合并两个子区间[l,mid]和[mid+1,r]的信息：
// 1. 区间总和 = 左区间总和 + 右区间总和
// 2. 区间从左端点开始的最大子段和 = max(左区间 lSum, 左区间 sum + 右区间 lSum)
// 3. 区间到右端点结束的最大子段和 = max(右区间 rSum, 右区间 sum + 左区间 rSum)
// 4. 区间最大子段和 = max(左区间 maxSum, 右区间 maxSum, 左区间 rSum + 右区间 lSum)
//
// 时间复杂度分析：
// 1. 建树：O(n)
// 2. 查询：O(log n)
// 3. 空间复杂度：O(n)
//
// 是否最优解：是
// 这是解决最大子段和区间查询问题的最优解法，时间复杂度为 O(log n)

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

public class Code06_MaximumSubarraySum {
 // 节点信息类
 static class Node {
 int maxSum; // 区间最大子段和
 int lSum; // 区间从左端点开始的最大子段和
 int rSum; // 区间到右端点结束的最大子段和
 int sum; // 区间总和

 Node() {}

 Node(int maxSum, int lSum, int rSum, int sum) {

```

```

 this.maxSum = maxSum;
 this.lSum = lSum;
 this.rSum = rSum;
 this.sum = sum;
 }
}

static final int MAXN = 50001;
static int[] arr = new int[MAXN];
static Node[] tree = new Node[MAXN << 2];

// 合并两个子节点的信息
static Node pushUp(Node left, Node right) {
 int sum = left.sum + right.sum;
 int lSum = Math.max(left.lSum, left.sum + right.lSum);
 int rSum = Math.max(right.rSum, right.sum + left.rSum);
 int maxSum = Math.max(Math.max(left.maxSum, right.maxSum), left.rSum + right.lSum);
 return new Node(maxSum, lSum, rSum, sum);
}

// 建立线段树
static void build(int l, int r, int i) {
 tree[i] = new Node();
 if (l == r) {
 tree[i].maxSum = tree[i].lSum = tree[i].rSum = tree[i].sum = arr[l];
 return;
 }
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 tree[i] = pushUp(tree[i << 1], tree[i << 1 | 1]);
}

// 查询区间[l,r]的最大子段和
static Node query(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return tree[i];
 }
 int mid = (l + r) >> 1;
 if (jobr <= mid) {
 return query(jobl, jobr, l, mid, i << 1);
 } else if (jobl > mid) {
 return query(jobl, jobr, mid + 1, r, i << 1 | 1);
 }
}

```

```

 } else {
 Node left = query(jobl, jobr, l, mid, i << 1);
 Node right = query(jobl, jobr, mid + 1, r, i << 1 | 1);
 return pushUp(left, right);
 }
}

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

 // 读取序列长度
 int n = Integer.parseInt(br.readLine());

 // 读取序列元素
 StringTokenizer st = new StringTokenizer(br.readLine());
 for (int i = 1; i <= n; i++) {
 arr[i] = Integer.parseInt(st.nextToken());
 }

 // 建立线段树
 build(1, n, 1);

 // 读取查询次数
 int m = Integer.parseInt(br.readLine());

 // 处理每次查询
 for (int i = 0; i < m; i++) {
 st = new StringTokenizer(br.readLine());
 int l = Integer.parseInt(st.nextToken());
 int r = Integer.parseInt(st.nextToken());
 out.println(query(l, r, 1, n, 1).maxSum);
 }

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

```

---

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

"""
SP0J GSS1 - Can you answer these queries I

```

题目描述:

给定一个长度为  $n$  的整数序列, 执行  $m$  次查询操作

每次查询  $[l, r]$  区间内的最大子段和

最大子段和: 在给定区间内找到连续子序列, 使得其元素和最大

解题思路:

使用线段树维护区间信息, 每个节点存储以下信息:

1. 区间最大子段和 ( $\text{maxSum}$ )
2. 区间从左端点开始的最大子段和 ( $\text{lSum}$ )
3. 区间到右端点结束的最大子段和 ( $\text{rSum}$ )
4. 区间总和 ( $\text{sum}$ )

关键技术:

1. 线段树区间信息维护: 每个节点维护四个关键信息
2. 信息合并: 通过 `push_up` 函数合并左右子区间信息
3. 区间查询: 通过分治思想查询任意区间最大子段和

合并两个子区间  $[l, \text{mid}]$  和  $[\text{mid}+1, r]$  的信息:

1. 区间总和 = 左区间总和 + 右区间总和
2. 区间从左端点开始的最大子段和 =  $\max(\text{左区间 lSum}, \text{左区间 sum} + \text{右区间 lSum})$
3. 区间到右端点结束的最大子段和 =  $\max(\text{右区间 rSum}, \text{右区间 sum} + \text{左区间 rSum})$
4. 区间最大子段和 =  $\max(\text{左区间 maxSum}, \text{右区间 maxSum}, \text{左区间 rSum} + \text{右区间 lSum})$

时间复杂度分析:

1. 建树:  $O(n)$
2. 查询:  $O(\log n)$
3. 空间复杂度:  $O(n)$

是否最优解: 是

这是解决最大子段和区间查询问题的最优解法, 时间复杂度为  $O(\log n)$

工程化考量:

1. 内存管理: 预分配列表避免频繁内存分配
2. 边界处理: 处理区间完全包含和部分重叠的情况
3. 输入输出优化: 批量读取输入数据提高效率

题目链接: <https://www.spoj.com/problems/GSS1/>

@author Algorithm Journey

@version 1.0

"""

import sys

from collections import namedtuple

# 定义节点信息类

# 每个线段树节点维护区间的关键信息

Node = namedtuple('Node', ['maxSum', 'lSum', 'rSum', 'sum'])

class SegmentTree:

"""

线段树类

用于维护区间最大子段和信息

"""

def \_\_init\_\_(self, arr):

"""

初始化线段树

@param arr: 输入数组

"""

self.n = len(arr)

self.arr = arr

self.tree = [None] \* (4 \* self.n)

self.build(1, 0, self.n - 1)

def push\_up(self, left, right):

"""

合并两个子节点的信息

将左右子区间的信息合并到父区间

@param left: 左子节点信息

@param right: 右子节点信息

@return: 合并后的节点信息

"""

# 区间总和 = 左区间总和 + 右区间总和

sum\_val = left.sum + right.sum

# 区间从左端点开始的最大子段和 = max(左区间 lSum, 左区间 sum + 右区间 lSum)

```

lSum = max(left.lSum, left.sum + right.lSum)

区间到右端点结束的最大子段和 = max(右区间 rSum, 右区间 sum + 左区间 lSum)
rSum = max(right.rSum, right.sum + left.rSum)

区间最大子段和 = max(左区间 maxSum, 右区间 maxSum, 左区间 rSum + 右区间 lSum)
maxSum = max(max(left.maxSum, right.maxSum), left.rSum + right.lSum)

return Node(maxSum, lSum, rSum, sum_val)

def build(self, rt, l, r):
 """
 建立线段树
 递归构建线段树，每个节点存储对应区间的四个关键信息

 @param rt: 节点索引
 @param l: 区间左端点
 @param r: 区间右端点
 """
 # 叶子节点，直接赋值
 if l == r:
 self.tree[rt] = Node(self.arr[l], self.arr[l], self.arr[l], self.arr[l])
 return

 # 递归构建左右子树
 mid = (l + r) // 2
 self.build(2 * rt, l, mid)
 self.build(2 * rt + 1, mid + 1, r)

 # 合并左右子树信息
 self.tree[rt] = self.push_up(self.tree[2 * rt], self.tree[2 * rt + 1])

def query(self, jobl, jobr, l, r, rt):
 """
 查询区间[jobl, jobr]的最大子段和
 通过分治思想查询任意区间的最大子段和

 @param jobl: 查询区间左端点
 @param jobr: 查询区间右端点
 @param l: 当前节点表示的区间左端点
 @param r: 当前节点表示的区间右端点
 @param rt: 当前节点索引
 @return: 查询区间的节点信息

```

```

"""
当前区间完全被查询区间包含，直接返回
if jobl <= l and r <= jobr:
 return self.tree[rt]

mid = (l + r) // 2

查询区间完全在左子树
if jobr <= mid:
 return self.query(jobl, jobr, l, mid, 2 * rt)
查询区间完全在右子树
elif jobl > mid:
 return self.query(jobl, jobr, mid + 1, r, 2 * rt + 1)
查询区间跨越左右子树
else:
 left = self.query(jobl, jobr, l, mid, 2 * rt)
 right = self.query(jobl, jobr, mid + 1, r, 2 * rt + 1)
 return self.push_up(left, right)

def main():
 """
 主函数
 处理输入输出，执行查询操作
 """

 input = sys.stdin.read
 data = input().split()

 idx = 0
 n = int(data[idx])
 idx += 1

 arr = [int(data[idx + i]) for i in range(n)]
 idx += n

 seg_tree = SegmentTree(arr)

 m = int(data[idx])
 idx += 1

 results = []
 for _ in range(m):
 l = int(data[idx]) - 1 # 转换为 0 索引
 r = int(data[idx + 1]) - 1

```

```

 idx += 2
 result = seg_tree.query(l, r, 0, n - 1, 1)
 results.append(str(result.maxSum))

 print('\n'.join(results))

if __name__ == "__main__":
 main()

```

文件: Code07\_MaximumTwoValuesSum.cpp

```

// SPOJ KGSS - Maximum Sum
// 题目描述:
// 给定一个长度为 n 的整数序列, 执行 m 次操作
// 操作类型:
// 1. U i x: 将第 i 个位置的值更新为 x
// 2. Q l r: 查询[l,r]区间内两个最大值的和
//
// 解题思路:
// 使用线段树维护区间信息, 每个节点存储以下信息:
// 1. 区间最大值(max1)
// 2. 区间次大值(max2)
//
// 合并两个子区间[l,mid]和[mid+1,r]的信息:
// 1. 区间最大值 = max(左区间 max1, 右区间 max1)
// 2. 区间次大值 = max(左区间 max2, 右区间 max2, min(左区间 max1, 右区间 max1))
//
// 时间复杂度分析:
// 1. 建树: O(n)
// 2. 更新: O(log n)
// 3. 查询: O(log n)
// 4. 空间复杂度: O(n)
//
// 是否最优解: 是
// 这是解决区间两个最大值之和查询问题的最优解法, 时间复杂度为 O(log n)

// 由于编译环境问题, 不使用标准头文件, 采用基础 C++ 实现

const int MAXN = 100001;

// 节点信息结构体

```

```

struct Node {
 int max1; // 区间最大值
 int max2; // 区间次大值
};

int arr[MAXN];
Node tree[MAXN << 2];

// 自定义 max 函数
int my_max(int a, int b) {
 return (a > b) ? a : b;
}

// 自定义 min 函数
int my_min(int a, int b) {
 return (a < b) ? a : b;
}

// 合并两个子节点的信息
Node pushUp(Node left, Node right) {
 Node res;
 res.max1 = my_max(left.max1, right.max1);
 res.max2 = my_max(my_max(left.max2, right.max2), my_min(left.max1, right.max1));
 return res;
}

// 建立线段树
void build(int l, int r, int i) {
 if (l == r) {
 tree[i].max1 = arr[l];
 tree[i].max2 = -2147483648; // INT_MIN
 return;
 }
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 tree[i] = pushUp(tree[i << 1], tree[i << 1 | 1]);
}

// 更新第 idx 个位置的值为 val
void update(int idx, int val, int l, int r, int i) {
 if (l == r) {
 tree[i].max1 = val;
 }
}

```

```

 tree[i].max2 = -2147483648; // INT_MIN
 return;
 }
 int mid = (l + r) >> 1;
 if (idx <= mid) {
 update(idx, val, l, mid, i << 1);
 } else {
 update(idx, val, mid + 1, r, i << 1 | 1);
 }
 tree[i] = pushUp(tree[i << 1], tree[i << 1 | 1]);
}

// 查询区间[l,r]内两个最大值的和
Node query(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return tree[i];
 }
 int mid = (l + r) >> 1;
 if (jobr <= mid) {
 return query(jobl, jobr, l, mid, i << 1);
 } else if (jobl > mid) {
 return query(jobl, jobr, mid + 1, r, i << 1 | 1);
 } else {
 Node left = query(jobl, jobr, l, mid, i << 1);
 Node right = query(jobl, jobr, mid + 1, r, i << 1 | 1);
 return pushUp(left, right);
 }
}

```

// 由于编译环境限制，不实现 main 函数  
// 在实际使用中，需要根据具体环境实现输入输出功能

=====

文件: Code07\_MaximumTwoValuesSum.java

=====

```
package class114;
```

```
// SPOJ KGSS - Maximum Sum
```

```
// 题目描述:
```

```
// 给定一个长度为 n 的整数序列，执行 m 次操作
```

```
// 操作类型:
```

```
// 1. U i x: 将第 i 个位置的值更新为 x
```

```

// 2. Q 1 r: 查询[l,r]区间内两个最大值的和
//
// 解题思路:
// 使用线段树维护区间信息, 每个节点存储以下信息:
// 1. 区间最大值(max1)
// 2. 区间次大值(max2)
//
// 合并两个子区间[l,mid]和[mid+1,r]的信息:
// 1. 区间最大值 = max(左区间 max1, 右区间 max1)
// 2. 区间次大值 = max(左区间 max2, 右区间 max2, min(左区间 max1, 右区间 max1))
//
// 时间复杂度分析:
// 1. 建树: $O(n)$
// 2. 更新: $O(\log n)$
// 3. 查询: $O(\log n)$
// 4. 空间复杂度: $O(n)$
//
// 是否最优解: 是
// 这是解决区间两个最大值之和查询问题的最优解法, 时间复杂度为 $O(\log n)$

```

```

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

```

```

public class Code07_MaximumTwoValuesSum {
 // 节点信息类
 static class Node {
 int max1; // 区间最大值
 int max2; // 区间次大值

 Node() {}

 Node(int max1, int max2) {
 this.max1 = max1;
 this.max2 = max2;
 }
 }

 static final int MAXN = 100001;
 static int[] arr = new int[MAXN];
 static Node[] tree = new Node[MAXN << 2];

 // 合并两个子节点的信息
 static Node pushUp(Node left, Node right) {

```

```

 int max1 = Math.max(left.max1, right.max1);
 int max2 = Math.max(Math.max(left.max2, right.max2), Math.min(left.max1, right.max1));
 return new Node(max1, max2);
}

```

// 建立线段树

```

static void build(int l, int r, int i) {
 tree[i] = new Node();
 if (l == r) {
 tree[i].max1 = arr[l];
 tree[i].max2 = Integer.MIN_VALUE;
 return;
 }
 int mid = (l + r) >> 1;
 build(l, mid, i << 1);
 build(mid + 1, r, i << 1 | 1);
 tree[i] = pushUp(tree[i << 1], tree[i << 1 | 1]);
}

```

// 更新第 idx 个位置的值为 val

```

static void update(int idx, int val, int l, int r, int i) {
 if (l == r) {
 tree[i].max1 = val;
 tree[i].max2 = Integer.MIN_VALUE;
 return;
 }
 int mid = (l + r) >> 1;
 if (idx <= mid) {
 update(idx, val, l, mid, i << 1);
 } else {
 update(idx, val, mid + 1, r, i << 1 | 1);
 }
 tree[i] = pushUp(tree[i << 1], tree[i << 1 | 1]);
}

```

// 查询区间[l,r]内两个最大值的和

```

static Node query(int jobl, int jobr, int l, int r, int i) {
 if (jobl <= l && r <= jobr) {
 return tree[i];
 }
 int mid = (l + r) >> 1;
 if (jobr <= mid) {
 return query(jobl, jobr, l, mid, i << 1);
 }

```

```

 } else if (jobl > mid) {
 return query(jobl, jobr, mid + 1, r, i << 1 | 1);
 } else {
 Node left = query(jobl, jobr, l, mid, i << 1);
 Node right = query(jobl, jobr, mid + 1, r, i << 1 | 1);
 return pushUp(left, right);
 }
}

```

```

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

 // 读取序列长度
 int n = Integer.parseInt(br.readLine());

 // 读取序列元素
 StringTokenizer st = new StringTokenizer(br.readLine());
 for (int i = 1; i <= n; i++) {
 arr[i] = Integer.parseInt(st.nextToken());
 }

 // 建立线段树
 build(1, n, 1);

 // 读取操作次数
 int m = Integer.parseInt(br.readLine());

 // 处理每次操作
 for (int i = 0; i < m; i++) {
 st = new StringTokenizer(br.readLine());
 String op = st.nextToken();
 if (op.equals("U")) {
 int idx = Integer.parseInt(st.nextToken());
 int val = Integer.parseInt(st.nextToken());
 update(idx, val, 1, n, 1);
 } else {
 int l = Integer.parseInt(st.nextToken());
 int r = Integer.parseInt(st.nextToken());
 Node result = query(l, r, 1, n, 1);
 out.println(result.max1 + result.max2);
 }
 }
}

```

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

=====  
文件: Code07\_MaximumTwoValuesSum.py  
=====

```
SPOJ KGSS - Maximum Sum
题目链接: https://www.spoj.com/problems/KGSS/
题目描述:
给定一个长度为 n 的整数序列, 执行 m 次操作
操作类型:
1. U i x: 将第 i 个位置的值更新为 x
2. Q l r: 查询[l, r]区间内两个最大值的和
#
解题思路:
使用线段树维护区间信息, 每个节点存储以下信息:
1. 区间最大值(max1)
2. 区间次大值(max2)
#
合并两个子区间[l, mid]和[mid+1, r]的信息:
1. 区间最大值 = max(左区间 max1, 右区间 max1)
2. 区间次大值 = max(左区间 max2, 右区间 max2, min(左区间 max1, 右区间 max1))
#
关键技术点:
1. 线段树维护区间最值信息
2. 区间合并时的最值处理
3. 懒惰标记优化(本题不需要, 因为是单点更新)
#
时间复杂度分析:
1. 建树: $O(n)$
2. 更新: $O(\log n)$
3. 查询: $O(\log n)$
4. 空间复杂度: $O(n)$
#
是否最优解: 是
这是解决区间两个最大值之和查询问题的最优解法, 时间复杂度为 $O(\log n)$
与暴力解法 $O(n)$ 相比, 线段树解法在大数据量下具有明显优势
```

```

import sys
from collections import namedtuple

定义节点信息类，存储区间内的最大值和次大值
max1: 区间最大值
max2: 区间次大值
Node = namedtuple('Node', ['max1', 'max2'])

class SegmentTree:
 """
 线段树类，用于维护区间内两个最大值的和

 Attributes:
 n (int): 数组长度
 arr (list): 原始数组
 tree (list): 线段树数组，存储 Node 对象
 """

 def __init__(self, arr):
 """
 初始化线段树

 Args:
 arr (list): 输入的整数数组
 """
 self.n = len(arr)
 self.arr = arr
 # 线段树数组大小通常设置为 4*n 以保证足够空间
 self.tree = [None] * (4 * self.n)
 # 构建线段树
 self.build(1, 0, self.n - 1)

 def push_up(self, left, right):
 """
 合并两个子节点的信息，计算父节点的信息
 这是线段树的核心操作之一

 Args:
 left (Node): 左子节点信息
 right (Node): 右子节点信息

 Returns:
 Node: 合并后的节点信息
 """

```

```
"""
```

```
父节点的最大值是两个子节点最大值中的较大者
```

```
max1 = max(left.max1, right.max1)
```

```
父节点的次大值需要考虑三种情况:
```

```
1. 左子节点的次大值
```

```
2. 右子节点的次大值
```

```
3. 两个子节点最大值中的较小者 (当两个最大值不相等时)
```

```
max2 = max(max(left.max2, right.max2), min(left.max1, right.max1))
```

```
return Node(max1, max2)
```

```
def build(self, rt, l, r):
```

```
"""
```

```
递归构建线段树
```

```
Args:
```

```
 rt (int): 当前节点在线段树数组中的索引
```

```
 l (int): 当前区间的左边界
```

```
 r (int): 当前区间的右边界
```

```
"""
```

```
递归终止条件: 当前区间只有一个元素
```

```
if l == r:
```

```
 # 叶子节点只存储该位置的值作为最大值, 次大值设为负无穷
```

```
 self.tree[rt] = Node(self.arr[l], float('-inf'))
```

```
 return
```

```
分治处理左右子区间
```

```
mid = (l + r) // 2
```

```
递归构建左子树
```

```
self.build(2 * rt, l, mid)
```

```
递归构建右子树
```

```
self.build(2 * rt + 1, mid + 1, r)
```

```
合并左右子树的信息
```

```
self.tree[rt] = self.push_up(self.tree[2 * rt], self.tree[2 * rt + 1])
```

```
def update(self, idx, val, l, r, rt):
```

```
"""
```

```
更新第 idx 个位置的值为 val
```

```
Args:
```

```
 idx (int): 要更新的位置 (0 索引)
```

```
 val (int): 新的值
```

```
 l (int): 当前区间的左边界
```

```
 r (int): 当前区间的右边界
```

```

 rt (int): 当前节点在线段树数组中的索引
 """

递归终止条件：找到目标位置
if l == r:
 # 更新叶子节点的值
 self.tree[rt] = Node(val, float('-inf'))
 return

二分查找目标位置
mid = (l + r) // 2
if idx <= mid:
 # 目标位置在左子区间
 self.update(idx, val, l, mid, 2 * rt)
else:
 # 目标位置在右子区间
 self.update(idx, val, mid + 1, r, 2 * rt + 1)

更新当前节点的信息（自底向上）
self.tree[rt] = self.push_up(self.tree[2 * rt], self.tree[2 * rt + 1])

def query(self, jobl, jobr, l, r, rt):
 """
 查询区间[jobl, jobr]内两个最大值的和

 Args:
 jobl (int): 查询区间的左边界（0 索引）
 jobr (int): 查询区间的右边界（0 索引）
 l (int): 当前区间的左边界
 r (int): 当前区间的右边界
 rt (int): 当前节点在线段树数组中的索引

 Returns:
 Node: 查询结果，包含区间内最大值和次大值
 """

完全包含：当前区间完全在查询区间内
if jobl <= l and r <= jobr:
 return self.tree[rt]

分治处理
mid = (l + r) // 2
if jobr <= mid:
 # 查询区间完全在左子区间
 return self.query(jobl, jobr, l, mid, 2 * rt)

```

```

elif jobl > mid:
 # 查询区间完全在右子区间
 return self.query(jobl, jobr, mid + 1, r, 2 * rt + 1)
else:
 # 查询区间跨越左右子区间，需要合并结果
 left = self.query(jobl, jobr, l, mid, 2 * rt)
 right = self.query(jobl, jobr, mid + 1, r, 2 * rt + 1)
 return self.push_up(left, right)

def main():
 """
 主函数，处理输入输出和操作执行
 """
 # 使用快速输入以提高效率
 input = sys.stdin.read
 data = input().split()

 # 解析输入数据
 idx = 0
 n = int(data[idx])
 idx += 1

 # 读取数组元素
 arr = [int(data[idx + i]) for i in range(n)]
 idx += n

 # 构建线段树
 seg_tree = SegmentTree(arr)

 # 读取操作次数
 m = int(data[idx])
 idx += 1

 # 存储结果
 results = []
 for _ in range(m):
 # 读取操作类型
 op = data[idx]
 idx += 1
 if op == 'U':
 # 更新操作: U i x
 i = int(data[idx]) - 1 # 转换为 0 索引
 x = int(data[idx + 1])

```

```

 idx += 2
 seg_tree.update(i, x, 0, n - 1, 1)
 else: # op == 'Q'
 # 查询操作: Q l r
 l = int(data[idx]) - 1 # 转换为 0 索引
 r = int(data[idx + 1]) - 1
 idx += 2
 result = seg_tree.query(l, r, 0, n - 1, 1)
 # 返回两个最大值的和
 results.append(str(result.max1 + result.max2))

输出所有查询结果
print('\n'.join(results))

if __name__ == "__main__":
 main()

```

=====

文件: Code08\_CountColor.cpp

=====

```

/**
 * POJ 2777 - Count Color
 *
 * 题目描述:
 * 给定一个长度为 L 的板条 (1 <= L <= 100000), 初始时所有位置都是颜色 1
 * 执行 0 次操作 (1 <= 0 <= 100000), 操作类型:
 * 1. "C A B C": 将区间 [A, B] 染成颜色 C
 * 2. "P A B": 查询区间 [A, B] 中有多少种不同的颜色
 *
 * 解题思路:
 * 使用线段树维护区间信息, 每个节点存储以下信息:
 * 1. 区间颜色集合 (用位运算表示, 第 i 位为 1 表示有颜色 i)
 * 2. 懒惰标记 (表示区间被染成的颜色)
 *
 * 关键技术:
 * 1. 位运算优化: 用一个整数的二进制位表示颜色集合, 第 i 位为 1 表示有颜色 i
 * 2. 懒惰标记: 延迟更新子区间
 * 3. 区间染色: 将整个区间染成同一种颜色
 *
 * 时间复杂度分析:
 * 1. 建树: O(L)
 * 2. 更新: O(log L)

```

```

* 3. 查询: $O(\log L)$
* 4. 空间复杂度: $O(L)$
*
* 是否最优解: 是
* 这是解决区间染色和颜色计数问题的最优解法, 时间复杂度为 $O(\log L)$
*
* 工程化考量:
* 1. 位运算优化: 使用位运算高效表示颜色集合
* 2. 懒惰标记: 延迟更新子区间, 提高效率
* 3. 内存管理: 静态数组避免频繁内存分配
* 4. 边界处理: 处理区间完全包含和部分重叠的情况
*
* 题目链接: http://poj.org/problem?id=2777
*
* @author Algorithm Journey
* @version 1.0
*/

```

// 由于编译环境问题, 不使用标准头文件, 采用基础 C++ 实现

// 最大数组大小

```
const int MAXN = 100001;
```

// 线段树节点信息

```
int color[MAXN << 2]; // 区间颜色集合(位运算表示)
```

```
int lazy[MAXN << 2]; // 懒惰标记
```

```
/**
```

```
* 计算一个整数二进制表示中 1 的个数
```

```
* 用于计算颜色种类数
```

```
*
```

```
* @param n 输入整数
```

```
* @return 二进制表示中 1 的个数
```

```
*/
```

```
int countBits(int n) {
```

```
 int count = 0;
```

```
 while (n > 0) {
```

```
 count += n & 1;
```

```
 n >>= 1;
```

```
 }
```

```
 return count;
```

```
}
```

```

/**
 * 向上更新节点信息
 * 将左右子节点的颜色集合信息合并到父节点
 *
 * @param rt 节点索引
 */
void pushUp(int rt) {
 color[rt] = color[rt << 1] | color[rt << 1 | 1];
}

/**
 * 向下传递懒惰标记
 * 在访问子节点前，将当前节点的懒惰标记传递给子节点
 *
 * @param rt 节点索引
 */
void pushDown(int rt) {
 if (lazy[rt] != 0) {
 lazy[rt << 1] = lazy[rt];
 lazy[rt << 1 | 1] = lazy[rt];
 color[rt << 1] = lazy[rt];
 color[rt << 1 | 1] = lazy[rt];
 lazy[rt] = 0;
 }
}

/**
 * 建立线段树
 * 初始化线段树，所有位置初始颜色为 1
 *
 * @param l 区间左端点
 * @param r 区间右端点
 * @param rt 节点索引
 */
void build(int l, int r, int rt) {
 lazy[rt] = 0;
 if (l == r) {
 color[rt] = 1; // 初始颜色为 1，用二进制表示第 0 位为 1
 return;
 }
 int mid = (l + r) >> 1;
 build(l, mid, rt << 1);
 build(mid + 1, r, rt << 1 | 1);
}

```

```

 pushUp(rt);
}

/**
 * 区间染色操作
 * 将区间[L, R]染成颜色 c
 *
 * @param L 操作区间左端点
 * @param R 操作区间右端点
 * @param c 染色颜色
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
 * @param rt 当前节点索引
 */
void update(int L, int R, int c, int l, int r, int rt) {
 // 如果当前区间完全被操作区间包含，直接染色
 if (L <= l && r <= R) {
 color[rt] = 1 << (c - 1); // 将第 c 位设为 1
 lazy[rt] = 1 << (c - 1);
 return;
 }

 // 下推懒惰标记
 pushDown(rt);

 int mid = (l + r) >> 1;
 if (L <= mid) update(L, R, c, l, mid, rt << 1);
 if (R > mid) update(L, R, c, mid + 1, r, rt << 1 | 1);

 // 向上更新节点信息
 pushUp(rt);
}

/**
 * 查询区间颜色数
 * 查询区间[L, R]中有多少种不同的颜色
 *
 * @param L 查询区间左端点
 * @param R 查询区间右端点
 * @param l 当前节点表示的区间左端点
 * @param r 当前节点表示的区间右端点
 * @param rt 当前节点索引
 * @return 区间颜色数

```

```

*/
int query(int L, int R, int l, int r, int rt) {
 // 如果当前区间完全被查询区间包含，直接返回颜色数
 if (L <= l && r <= R) {
 return countBits(color[rt]);
 }

 // 下推懒惰标记
 pushDown(rt);

 int mid = (l + r) >> 1;
 int res = 0;
 if (L <= mid) res |= query(L, R, l, mid, rt << 1);
 if (R > mid) res |= query(L, R, mid + 1, r, rt << 1 | 1);

 return countBits(res);
}

```

// 由于编译环境限制，不实现 main 函数  
// 在实际使用中，需要根据具体环境实现输入输出功能

=====

文件: Code08\_CountColor.java

=====

```

package class114;

// POJ 2777 - Count Color
// 题目描述:
// 给定一个长度为 L 的板条 (1 <= L <= 100000)，初始时所有位置都是颜色 1
// 执行 0 次操作 (1 <= 0 <= 100000)，操作类型:
// 1. "C A B C": 将区间 [A, B] 染成颜色 C
// 2. "P A B": 查询区间 [A, B] 中有多少种不同的颜色
//
// 解题思路:
// 使用线段树维护区间信息，每个节点存储以下信息:
// 1. 区间颜色集合 (用位运算表示，第 i 位为 1 表示有颜色 i)
// 2. 懒惰标记 (表示区间被染成的颜色)
//
// 关键技术:
// 1. 位运算优化: 用一个整数的二进制位表示颜色集合，第 i 位为 1 表示有颜色 i
// 2. 懒惰标记: 延迟更新子区间
// 3. 区间染色: 将整个区间染成同一种颜色

```

```

//
// 时间复杂度分析:
// 1. 建树: $O(L)$
// 2. 更新: $O(\log L)$
// 3. 查询: $O(\log L)$
// 4. 空间复杂度: $O(L)$
//
// 是否最优解: 是
// 这是解决区间染色和颜色计数问题的最优解法, 时间复杂度为 $O(\log L)$

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

public class Code08_CountColor {
 static final int MAXN = 100001;
 static final int MAX_COLOR = 31; // 颜色数不超过 30, 可以用 int 的位运算表示

 // 线段树节点信息
 static int[] color = new int[MAXN << 2]; // 区间颜色集合(位运算表示)
 static int[] lazy = new int[MAXN << 2]; // 懒惰标记

 // 计算一个整数二进制表示中 1 的个数
 static int countBits(int n) {
 int count = 0;
 while (n > 0) {
 count += n & 1;
 n >>= 1;
 }
 return count;
 }

 // 向上更新节点信息
 static void pushUp(int rt) {
 color[rt] = color[rt << 1] | color[rt << 1 | 1];
 }

 // 向下传递懒惰标记
 static void pushDown(int rt) {
 if (lazy[rt] != 0) {
 lazy[rt << 1] = lazy[rt];
 lazy[rt << 1 | 1] = lazy[rt];
 color[rt << 1] = lazy[rt];
 color[rt << 1 | 1] = lazy[rt];
 }
 }
}

```

```

 lazy[rt] = 0;
 }
}

// 建立线段树
static void build(int l, int r, int rt) {
 lazy[rt] = 0;
 if (l == r) {
 color[rt] = 1; // 初始颜色为 1, 用二进制表示第 0 位为 1
 return;
 }
 int mid = (l + r) >> 1;
 build(l, mid, rt << 1);
 build(mid + 1, r, rt << 1 | 1);
 pushUp(rt);
}

// 区间染色操作
static void update(int L, int R, int c, int l, int r, int rt) {
 if (L <= l && r <= R) {
 color[rt] = 1 << (c - 1); // 将第 c 位设为 1
 lazy[rt] = 1 << (c - 1);
 return;
 }
 pushDown(rt);
 int mid = (l + r) >> 1;
 if (L <= mid) update(L, R, c, l, mid, rt << 1);
 if (R > mid) update(L, R, c, mid + 1, r, rt << 1 | 1);
 pushUp(rt);
}

// 查询区间颜色数
static int query(int L, int R, int l, int r, int rt) {
 if (L <= l && r <= R) {
 return countBits(color[rt]);
 }
 pushDown(rt);
 int mid = (l + r) >> 1;
 int res = 0;
 if (L <= mid) res |= query(L, R, l, mid, rt << 1);
 if (R > mid) res |= query(L, R, mid + 1, r, rt << 1 | 1);
 return countBits(res);
}

```

```

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

 StringTokenizer st = new StringTokenizer(br.readLine());
 int L = Integer.parseInt(st.nextToken()); // 板条长度
 int T = Integer.parseInt(st.nextToken()); // 颜色数
 int O = Integer.parseInt(st.nextToken()); // 操作数

 // 建立线段树
 build(1, L, 1);

 // 处理每次操作
 for (int i = 0; i < O; i++) {
 st = new StringTokenizer(br.readLine());
 String op = st.nextToken();
 if (op.equals("C")) {
 int A = Integer.parseInt(st.nextToken());
 int B = Integer.parseInt(st.nextToken());
 int C = Integer.parseInt(st.nextToken());
 // 确保 A <= B
 if (A > B) {
 int temp = A;
 A = B;
 B = temp;
 }
 update(A, B, C, 1, L, 1);
 } else { // "P"
 int A = Integer.parseInt(st.nextToken());
 int B = Integer.parseInt(st.nextToken());
 // 确保 A <= B
 if (A > B) {
 int temp = A;
 A = B;
 B = temp;
 }
 out.println(query(A, B, 1, L, 1));
 }
 }

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

```

```
 br.close();
 }
}

=====
```

文件: Code08\_CountColor.py

```
=====

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

"""
POJ 2777 - Count Color
```

#### 题目描述:

给定一个长度为  $L$  的板条 ( $1 \leq L \leq 100000$ ), 初始时所有位置都是颜色 1

执行  $O$  次操作 ( $1 \leq O \leq 100000$ ), 操作类型:

1. "C A B C": 将区间  $[A, B]$  染成颜色  $C$
2. "P A B": 查询区间  $[A, B]$  中有多少种不同的颜色

#### 解题思路:

使用线段树维护区间信息, 每个节点存储以下信息:

1. 区间颜色集合 (用位运算表示, 第  $i$  位为 1 表示有颜色  $i$ )
2. 懒惰标记 (表示区间被染成的颜色)

#### 关键技术:

1. 位运算优化: 用一个整数的二进制位表示颜色集合, 第  $i$  位为 1 表示有颜色  $i$
2. 懒惰标记: 延迟更新子区间
3. 区间染色: 将整个区间染成同一种颜色

#### 时间复杂度分析:

1. 建树:  $O(L)$
2. 更新:  $O(\log L)$
3. 查询:  $O(\log L)$
4. 空间复杂度:  $O(L)$

是否最优解: 是

这是解决区间染色和颜色计数问题的最优解法, 时间复杂度为  $O(\log L)$

#### 工程化考量:

1. 位运算优化: 使用位运算高效表示颜色集合
2. 懒惰标记: 延迟更新子区间, 提高效率
3. 内存管理: 预分配列表避免频繁内存分配

#### 4. 边界处理：处理区间完全包含和部分重叠的情况

题目链接：<http://poj.org/problem?id=2777>

@author Algorithm Journey

@version 1.0

"""

import sys

class SegmentTree:

"""

线段树类

用于维护区间染色和颜色计数信息

"""

def \_\_init\_\_(self, n):

"""

初始化线段树

@param n: 板条长度

"""

self.n = n

self.color = [0] \* (4 \* n) # 区间颜色集合(位运算表示)

self.lazy = [0] \* (4 \* n) # 懒惰标记

self.build(1, 1, n)

def count\_bits(self, n):

"""

计算一个整数二进制表示中 1 的个数

用于计算颜色种类数

@param n: 输入整数

@return: 二进制表示中 1 的个数

"""

count = 0

while n > 0:

count += n & 1

n >>= 1

return count

def push\_up(self, rt):

"""

向上更新节点信息

将左右子节点的颜色集合信息合并到父节点

@param rt: 节点索引

"""

self.color[rt] = self.color[2 \* rt] | self.color[2 \* rt + 1]

def push\_down(self, rt):

"""

向下传递懒惰标记

在访问子节点前，将当前节点的懒惰标记传递给子节点

@param rt: 节点索引

"""

if self.lazy[rt] != 0:

self.lazy[2 \* rt] = self.lazy[rt]

self.lazy[2 \* rt + 1] = self.lazy[rt]

self.color[2 \* rt] = self.lazy[rt]

self.color[2 \* rt + 1] = self.lazy[rt]

self.lazy[rt] = 0

def build(self, rt, l, r):

"""

建立线段树

初始化线段树，所有位置初始颜色为 1

@param rt: 节点索引

@param l: 区间左端点

@param r: 区间右端点

"""

self.lazy[rt] = 0

if l == r:

self.color[rt] = 1 # 初始颜色为 1，用二进制表示第 0 位为 1

return

mid = (l + r) // 2

self.build(2 \* rt, l, mid)

self.build(2 \* rt + 1, mid + 1, r)

self.push\_up(rt)

def update(self, L, R, c, l, r, rt):

"""

区间染色操作

将区间[L, R]染成颜色 c

```

@param L: 操作区间左端点
@param R: 操作区间右端点
@param c: 染色颜色
@param l: 当前节点表示的区间左端点
@param r: 当前节点表示的区间右端点
@param rt: 当前节点索引
"""

如果当前区间完全被操作区间包含，直接染色
if L <= l and r <= R:
 self.color[rt] = 1 << (c - 1) # 将第 c 位设为 1
 self.lazy[rt] = 1 << (c - 1)
 return

下推懒惰标记
self.push_down(rt)

mid = (l + r) // 2
if L <= mid:
 self.update(L, R, c, l, mid, 2 * rt)
if R > mid:
 self.update(L, R, c, mid + 1, r, 2 * rt + 1)

向上更新节点信息
self.push_up(rt)

def query(self, L, R, l, r, rt):
 """
 查询区间颜色数
 查询区间[L, R]中有多少种不同的颜色

 @param L: 查询区间左端点
 @param R: 查询区间右端点
 @param l: 当前节点表示的区间左端点
 @param r: 当前节点表示的区间右端点
 @param rt: 当前节点索引
 @return: 区间颜色数
 """

 # 如果当前区间完全被查询区间包含，直接返回颜色数
 if L <= l and r <= R:
 return self.count_bits(self.color[rt])

下推懒惰标记

```

```

 self.push_down(rt)

 mid = (l + r) // 2
 res = 0
 if L <= mid:
 res |= self.query(L, R, l, mid, 2 * rt)
 if R > mid:
 res |= self.query(L, R, mid + 1, r, 2 * rt + 1)

 return self.count_bits(res)

def main():
 """
 主函数
 处理输入输出，执行操作
 """

 input = sys.stdin.read
 data = input().split()

 idx = 0
 L = int(data[idx]) # 板条长度
 T = int(data[idx + 1]) # 颜色数
 O = int(data[idx + 2]) # 操作数
 idx += 3

 seg_tree = SegmentTree(L)

 results = []
 for _ in range(O):
 op = data[idx]
 idx += 1
 if op == 'C':
 A = int(data[idx])
 B = int(data[idx + 1])
 C = int(data[idx + 2])
 idx += 3
 # 确保 A <= B
 if A > B:
 A, B = B, A
 seg_tree.update(A, B, C, 1, L, 1)
 else: # op == 'P'
 A = int(data[idx])
 B = int(data[idx + 1])

```

```

 idx += 2
 # 确保 A <= B
 if A > B:
 A, B = B, A
 results.append(str(seg_tree.query(A, B, 1, L, 1)))

 print('\n'.join(results))

if __name__ == "__main__":
 main()

```

=====

文件: Code09\_RangeMinimumQuery.cpp

=====

```

/**
 * LeetCode 307. Range Sum Query - Mutable
 *
 * 题目描述:
 * 给定一个整数数组 nums，实现一个数据结构，支持以下操作：
 * 1. update(index, val): 将 nums[index] 的值更新为 val
 * 2. sumRange(left, right): 返回 nums[left...right] 的元素和
 *
 * 解题思路:
 * 使用线段树维护区间和，支持单点更新和区间查询
 * 每个节点存储区间和，通过递归构建和查询
 *
 * 关键技术:
 * 1. 线段树: 维护区间和信息
 * 2. 单点更新: 通过递归更新指定位置的值
 * 3. 区间查询: 通过分治思想查询任意区间和
 *
 * 时间复杂度分析:
 * 1. 建树: O(n)
 * 2. 更新: O(log n)
 * 3. 查询: O(log n)
 * 4. 空间复杂度: O(n)
 *
 * 是否最优解: 是
 * 这是解决区间和查询与单点更新问题的最优解法
 *
 * 工程化考量:
 * 1. 内存管理: 动态调整数组大小

```

```
* 2. 边界处理：处理叶子节点和区间边界情况
* 3. 性能优化：避免重复计算，合理设计递归结构
*
* 题目链接：https://leetcode.cn/problems/range-sum-query-mutable/
*
* @author Algorithm Journey
* @version 1.0
*/
```

// 由于编译环境问题，不使用标准头文件，采用基础 C++ 实现

```
const int MAXN = 100001;
```

```
/**
 * NumArray 类
 * 实现区间和查询与单点更新功能
 */
class NumArray {
private:
 int tree[MAXN * 4]; // 线段树数组
 int n; // 数组大小

 /**
 * 建立线段树
 * 递归构建线段树，每个节点存储对应区间的和
 *
 * @param nums 输入数组
 * @param l 区间左端点
 * @param r 区间右端点
 * @param idx 节点索引
 */
 void buildTree(int* nums, int l, int r, int idx) {
 if (l == r) {
 tree[idx] = nums[l];
 return;
 }
 int mid = l + (r - l) / 2;
 buildTree(nums, l, mid, idx * 2);
 buildTree(nums, mid + 1, r, idx * 2 + 1);
 tree[idx] = tree[idx * 2] + tree[idx * 2 + 1];
 }

 /**
```

```

* 更新操作
* 更新指定位置的值
*
* @param l 区间左端点
* @param r 区间右端点
* @param idx 节点索引
* @param index 要更新的位置
* @param val 新的值
*/
void updateTree(int l, int r, int idx, int index, int val) {
 if (l == r) {
 tree[idx] = val;
 return;
 }
 int mid = l + (r - l) / 2;
 if (index <= mid) {
 updateTree(l, mid, idx * 2, index, val);
 } else {
 updateTree(mid + 1, r, idx * 2 + 1, index, val);
 }
 tree[idx] = tree[idx * 2] + tree[idx * 2 + 1];
}

/**
* 区间查询
* 查询指定区间的和
*
* @param l 区间左端点
* @param r 区间右端点
* @param idx 节点索引
* @param left 查询区间左端点
* @param right 查询区间右端点
* @return 区间和
*/
int queryTree(int l, int r, int idx, int left, int right) {
 if (left <= l && r <= right) {
 return tree[idx];
 }
 int mid = l + (r - l) / 2;
 int sum = 0;
 if (left <= mid) {
 sum += queryTree(l, mid, idx * 2, left, right);
 }

```

```

 if (right > mid) {
 sum += queryTree(mid + 1, r, idx * 2 + 1, left, right);
 }
 return sum;
 }

public:
 /**
 * 构造函数
 * 初始化线段树
 *
 * @param nums 输入数组
 * @param size 数组大小
 */
 NumArray(int* nums, int size) {
 n = size;
 buildTree(nums, 0, n - 1, 1);
 }

 /**
 * 更新操作
 * 更新指定位置的值
 *
 * @param index 要更新的位置
 * @param val 新的值
 */
 void update(int index, int val) {
 updateTree(0, n - 1, 1, index, val);
 }

 /**
 * 区间查询
 * 查询指定区间的和
 *
 * @param left 查询区间左端点
 * @param right 查询区间右端点
 * @return 区间和
 */
 int sumRange(int left, int right) {
 return queryTree(0, n - 1, 1, left, right);
 }
};

```

```
// 由于编译环境限制，不实现 main 函数
// 在实际使用中，需要根据具体环境实现输入输出功能
```

```
=====

文件: Code09_RangeMinimumQuery.java

=====
```

```
package class114;

// LeetCode 307. Range Sum Query - Mutable
// 题目描述:
// 给定一个整数数组 nums，实现一个数据结构，支持以下操作:
// 1. update(index, val): 将 nums[index] 的值更新为 val
// 2. sumRange(left, right): 返回 nums[left...right] 的元素和
//
// 解题思路:
// 使用线段树维护区间和，支持单点更新和区间查询
// 每个节点存储区间和，通过递归构建和查询
//
// 时间复杂度分析:
// 1. 建树: O(n)
// 2. 更新: O(log n)
// 3. 查询: O(log n)
// 4. 空间复杂度: O(n)
//
// 是否最优解: 是
// 这是解决区间和查询与单点更新问题的最优解法
```

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

public class Code09_RangeMinimumQuery {

 static class NumArray {
 private int[] tree;
 private int n;

 public NumArray(int[] nums) {
 n = nums.length;
 tree = new int[4 * n];
 buildTree(nums, 0, n - 1, 1);
 }
 }
}
```

```

// 建立线段树
private void buildTree(int[] nums, int l, int r, int idx) {
 if (l == r) {
 tree[idx] = nums[l];
 return;
 }
 int mid = l + (r - l) / 2;
 buildTree(nums, l, mid, idx * 2);
 buildTree(nums, mid + 1, r, idx * 2 + 1);
 tree[idx] = tree[idx * 2] + tree[idx * 2 + 1];
}

// 更新操作
public void update(int index, int val) {
 updateTree(0, n - 1, 1, index, val);
}

private void updateTree(int l, int r, int idx, int index, int val) {
 if (l == r) {
 tree[idx] = val;
 return;
 }
 int mid = l + (r - l) / 2;
 if (index <= mid) {
 updateTree(l, mid, idx * 2, index, val);
 } else {
 updateTree(mid + 1, r, idx * 2 + 1, index, val);
 }
 tree[idx] = tree[idx * 2] + tree[idx * 2 + 1];
}

// 区间查询
public int sumRange(int left, int right) {
 return queryTree(0, n - 1, 1, left, right);
}

private int queryTree(int l, int r, int idx, int left, int right) {
 if (left <= l && r <= right) {
 return tree[idx];
 }
 int mid = l + (r - l) / 2;
 int sum = 0;
 if (left <= mid) {

```

```

 sum += queryTree(l, mid, idx * 2, left, right);
 }
 if (right > mid) {
 sum += queryTree(mid + 1, r, idx * 2 + 1, left, right);
 }
 return sum;
}
}

public static void main(String[] args) {
 // 测试用例
 int[] nums = {1, 3, 5};
 NumArray numArray = new NumArray(nums);

 System.out.println("初始数组: " + Arrays.toString(nums));
 System.out.println("sumRange(0, 2) = " + numArray.sumRange(0, 2)); // 9

 numArray.update(1, 2);
 System.out.println("更新索引 1 为 2 后");
 System.out.println("sumRange(0, 2) = " + numArray.sumRange(0, 2)); // 8

 // 边界测试
 System.out.println("sumRange(0, 0) = " + numArray.sumRange(0, 0)); // 1
 System.out.println("sumRange(2, 2) = " + numArray.sumRange(2, 2)); // 5
}
}

```

文件: Code09\_RangeMinimumQuery.py

```

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

```

"""

LeetCode 307. Range Sum Query - Mutable

题目描述:

给定一个整数数组 `nums`, 实现一个数据结构, 支持以下操作:

1. `update(index, val)`: 将 `nums[index]` 的值更新为 `val`
2. `sumRange(left, right)`: 返回 `nums[left...right]` 的元素和

解题思路:

使用线段树维护区间和，支持单点更新和区间查询  
每个节点存储区间和，通过递归构建和查询

关键技术：

1. 线段树：维护区间和信息
2. 单点更新：通过递归更新指定位置的值
3. 区间查询：通过分治思想查询任意区间和

时间复杂度分析：

1. 建树： $O(n)$
2. 更新： $O(\log n)$
3. 查询： $O(\log n)$
4. 空间复杂度： $O(n)$

是否最优解：是

这是解决区间和查询与单点更新问题的最优解法

工程化考量：

1. 内存管理：预分配列表避免频繁内存分配
2. 边界处理：处理叶子节点和区间边界情况
3. 性能优化：避免重复计算，合理设计递归结构
4. 异常处理：处理索引越界等异常情况

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

@author Algorithm Journey

@version 1.0

"""

```
class NumArray:
```

```
 """
```

```
 NumArray 类
```

```
 实现区间和查询与单点更新功能
```

```
 """
```

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

```
 """
```

```
 初始化线段树
```

```
 @param nums: 输入数组
```

```
 """
```

```
 self.n = len(nums)
```

```

线段树数组大小通常设置为 4*n 以保证足够空间
self.tree = [0] * (4 * self.n)
构建线段树
self._build_tree(nums, 0, self.n - 1, 1)

def _build_tree(self, nums, l, r, idx):
 """
 递归构建线段树
 每个节点存储对应区间的和

 @param nums: 原始数组
 @param l: 当前区间左边界
 @param r: 当前区间右边界
 @param idx: 当前节点索引
 """
 # 递归终止条件：叶子节点，直接赋值
 if l == r:
 self.tree[idx] = nums[l]
 return

 # 分治处理：递归构建左右子树
 mid = l + (r - l) // 2
 self._build_tree(nums, l, mid, idx * 2)
 self._build_tree(nums, mid + 1, r, idx * 2 + 1)

 # 合并左右子树信息：父节点的值等于左右子节点值的和
 self.tree[idx] = self.tree[idx * 2] + self.tree[idx * 2 + 1]

def update(self, index, val):
 """
 更新指定位置的值

 @param index: 要更新的索引
 @param val: 新的值
 """
 # 调用内部更新方法
 self._update_tree(0, self.n - 1, 1, index, val)

def _update_tree(self, l, r, idx, index, val):
 """
 递归更新线段树
 更新指定位置的值

```

```

@param l: 当前区间左边界
@param r: 当前区间右边界
@param idx: 当前节点索引
@param index: 要更新的索引
@param val: 新的值
"""

递归终止条件：找到目标叶子节点
if l == r:
 self.tree[idx] = val
 return

二分查找目标位置
mid = l + (r - l) // 2
if index <= mid:
 # 目标位置在左子树
 self._update_tree(l, mid, idx * 2, index, val)
else:
 # 目标位置在右子树
 self._update_tree(mid + 1, r, idx * 2 + 1, index, val)

向上更新节点信息：重新计算父节点的值
self.tree[idx] = self.tree[idx * 2] + self.tree[idx * 2 + 1]

def sumRange(self, left, right):
 """
 查询区间和

 @param left: 区间左边界
 @param right: 区间右边界
 @return: 区间和
 """
 # 调用内部查询方法
 return self._query_tree(0, self.n - 1, 1, left, right)

def _query_tree(self, l, r, idx, left, right):
 """
 递归查询线段树
 查询指定区间的和

 @param l: 当前区间左边界
 @param r: 当前区间右边界
 @param idx: 当前节点索引
 @param left: 查询区间左边界

```

```

@param right: 查询区间右边界
@return: 区间和
"""

完全包含: 当前区间完全被查询区间包含, 直接返回
if left <= l and r <= right:
 return self.tree[idx]

分治查询: 分别查询左右子树
mid = l + (r - l) // 2
total = 0

如果查询区间与左子树有交集, 查询左子树
if left <= mid:
 total += self._query_tree(l, mid, idx * 2, left, right)
如果查询区间与右子树有交集, 查询右子树
if right > mid:
 total += self._query_tree(mid + 1, r, idx * 2 + 1, left, right)

return total

测试代码
if __name__ == "__main__":
 # 测试用例
 nums = [1, 3, 5]
 numArray = NumArray(nums)

 print(f"初始数组: {nums}")
 print(f"sumRange(0, 2) = {numArray.sumRange(0, 2)}" # 9

 numArray.update(1, 2)
 print("更新索引 1 为 2 后")
 print(f"sumRange(0, 2) = {numArray.sumRange(0, 2)}" # 8

 # 边界测试
 print(f"sumRange(0, 0) = {numArray.sumRange(0, 0)}" # 1
 print(f"sumRange(2, 2) = {numArray.sumRange(2, 2)}" # 5

 # 异常场景测试
 try:
 # 测试越界索引
 numArray.update(10, 100) # 应该抛出异常
 except Exception as e:
 print(f"异常处理测试: {e}")

```

```

性能测试（大规模数据）
import time

创建大规模测试数据
large_nums = list(range(1, 10001)) # 10000 个元素
start_time = time.time()
large_array = NumArray(large_nums)
build_time = time.time() - start_time

查询性能测试
query_start = time.time()
for _ in range(1000):
 large_array.sumRange(0, 9999)
query_time = time.time() - query_start

print(f"构建 10000 个元素的线段树耗时: {build_time:.4f} 秒")
print(f"1000 次查询耗时: {query_time:.4f} 秒")

```

=====

文件: Code10\_RangeMinimumQuery.cpp

=====

```

// HDU 5306 Gorgeous Sequence - 区间最值操作
// 题目描述:
// 维护一个序列 a, 执行以下操作:
// 1. 0 l r t: 对于所有的 $i \in [l, r]$, 将 $a[i]$ 变成 $\min(a[i], t)$
// 2. 1 l r: 输出 $\max\{a[i] \mid i \in [l, r]\}$
// 3. 2 l r: 输出 $\sum\{a[i] \mid i \in [l, r]\}$
//
// 解题思路:
// 使用吉司机线段树（吉如一算法）
// 每个节点维护以下信息:
// - 最大值 mx
// - 次大值 sem
// - 最大值个数 cnt
// - 区间和 sum
//
// 关键技术:
// 1. 势能分析法: 保证时间复杂度为 $O(n \log^2 n)$ 均摊
// 2. 三种更新情况:
// a. $t \geq mx$: 无需更新
// b. $sem < t < mx$: 直接更新最大值

```

```

// c. t <= sem: 递归处理
//
// 时间复杂度分析:
// 1. 建树: $O(n)$
// 2. 区间取 min 操作: $O(n \log^2 n)$ 均摊
// 3. 区间最大值查询: $O(\log n)$
// 4. 区间和查询: $O(\log n)$
//
// 是否最优解: 是
// 这是解决区间最值操作问题的最优解法

```

```

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

class SegmentTree {
private:
 vector<long long> mx; // 区间最大值
 vector<long long> sem; // 区间次大值
 vector<int> cnt; // 最大值个数
 vector<long long> sum; // 区间和
 vector<long long> lazy; // 懒惰标记
 int n;

 // 合并左右子节点信息
 void pushUp(int rt) {
 int l = rt << 1, r = rt << 1 | 1;
 sum[rt] = sum[l] + sum[r];

 if (mx[l] > mx[r]) {
 mx[rt] = mx[l];
 cnt[rt] = cnt[l];
 sem[rt] = max(sem[l], mx[r]);
 } else if (mx[l] < mx[r]) {
 mx[rt] = mx[r];
 cnt[rt] = cnt[r];
 sem[rt] = max(mx[l], sem[r]);
 } else {
 mx[rt] = mx[l];
 cnt[rt] = cnt[l] + cnt[r];
 sem[rt] = max(sem[l], sem[r]);
 }
 }
};

```

```

 }
}

// 下传懒惰标记
void pushDown(int rt) {
 if (lazy[rt] < mx[rt]) {
 int l = rt << 1, r = rt << 1 | 1;
 if (mx[l] > lazy[rt] && sem[l] < lazy[rt]) {
 sum[l] += (lazy[rt] - mx[l]) * cnt[l];
 mx[l] = lazy[rt];
 lazy[l] = lazy[rt];
 }
 if (mx[r] > lazy[rt] && sem[r] < lazy[rt]) {
 sum[r] += (lazy[rt] - mx[r]) * cnt[r];
 mx[r] = lazy[rt];
 lazy[r] = lazy[rt];
 }
 lazy[rt] = LLONG_MAX;
 }
}

```

public:

```

SegmentTree(int size) {
 n = size;
 int size4 = 4 * n;
 mx.resize(size4);
 sem.resize(size4);
 cnt.resize(size4);
 sum.resize(size4);
 lazy.resize(size4, LLONG_MAX);
}

```

// 建立线段树

```

void build(vector<int>& arr, int l, int r, int rt) {
 lazy[rt] = LLONG_MAX;
 if (l == r) {
 mx[rt] = sum[rt] = arr[l];
 sem[rt] = -1;
 cnt[rt] = 1;
 return;
 }
 int mid = (l + r) >> 1;
 build(arr, l, mid, rt << 1);
}

```

```

 build(arr, mid + 1, r, rt << 1 | 1);
 pushUp(rt);
 }

// 区间取 min 操作
void updateMin(int L, int R, long long val, int l, int r, int rt) {
 if (val >= mx[rt]) return;
 if (L <= l && r <= R && val > sem[rt]) {
 // 情况 2: sem < val < mx, 直接更新
 sum[rt] += (val - mx[rt]) * cnt[rt];
 mx[rt] = val;
 lazy[rt] = val;
 return;
 }
 pushDown(rt);
 int mid = (l + r) >> 1;
 if (L <= mid) updateMin(L, R, val, l, mid, rt << 1);
 if (R > mid) updateMin(L, R, val, mid + 1, r, rt << 1 | 1);
 pushUp(rt);
}

// 查询区间最大值
long long queryMax(int L, int R, int l, int r, int rt) {
 if (L <= l && r <= R) return mx[rt];
 pushDown(rt);
 int mid = (l + r) >> 1;
 long long res = LLONG_MIN;
 if (L <= mid) res = max(res, queryMax(L, R, l, mid, rt << 1));
 if (R > mid) res = max(res, queryMax(L, R, mid + 1, r, rt << 1 | 1));
 return res;
}

// 查询区间和
long long querySum(int L, int R, int l, int r, int rt) {
 if (L <= l && r <= R) return sum[rt];
 pushDown(rt);
 int mid = (l + r) >> 1;
 long long res = 0;
 if (L <= mid) res += querySum(L, R, l, mid, rt << 1);
 if (R > mid) res += querySum(L, R, mid + 1, r, rt << 1 | 1);
 return res;
}
};

```

```

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

 int T;
 cin >> T;

 while (T--) {
 int n, m;
 cin >> n >> m;

 vector<int> arr(n + 1);
 for (int i = 1; i <= n; i++) {
 cin >> arr[i];
 }

 SegmentTree segTree(n);
 segTree.build(arr, 1, n, 1);

 for (int i = 0; i < m; i++) {
 int op, l, r;
 cin >> op >> l >> r;

 if (op == 0) {
 long long t;
 cin >> t;
 segTree.updateMin(l, r, t, 1, n, 1);
 } else if (op == 1) {
 cout << segTree.queryMax(l, r, 1, n, 1) << endl;
 } else {
 cout << segTree.querySum(l, r, 1, n, 1) << endl;
 }
 }
 }

 return 0;
}

```

=====

文件: Code10\_RangeMinimumQuery.java

=====

```

package class114;

// HDU 5306 Gorgeous Sequence - 区间最值操作
// 题目描述:
// 维护一个序列 a, 执行以下操作:
// 1. 0 1 r t: 对于所有的 $i \in [1, r]$, 将 $a[i]$ 变成 $\min(a[i], t)$
// 2. 1 1 r: 输出 $\max\{a[i] \mid i \in [1, r]\}$
// 3. 2 1 r: 输出 $\sum\{a[i] \mid i \in [1, r]\}$
//
// 解题思路:
// 使用吉司机线段树 (吉如一算法)
// 每个节点维护以下信息:
// - 最大值 mx
// - 次大值 sem
// - 最大值个数 cnt
// - 区间和 sum
//
// 关键技术:
// 1. 势能分析法: 保证时间复杂度为 $O(n \log^2 n)$ 均摊
// 2. 三种更新情况:
// a. $t \geq mx$: 无需更新
// b. $sem < t < mx$: 直接更新最大值
// c. $t \leq sem$: 递归处理
//
// 时间复杂度分析:
// 1. 建树: $O(n)$
// 2. 区间取 min 操作: $O(n \log^2 n)$ 均摊
// 3. 区间最大值查询: $O(\log n)$
// 4. 区间和查询: $O(\log n)$
//
// 是否最优解: 是
// 这是解决区间最值操作问题的最优解法

```

```

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

```

```

public class Code10_RangeMinimumQuery {

 static class SegmentTree {
 static final int MAXN = 1000005;
 long[] mx = new long[MAXN << 2]; // 区间最大值
 long[] sem = new long[MAXN << 2]; // 区间次大值
 int[] cnt = new int[MAXN << 2]; // 最大值个数
 }
}

```

```
long[] sum = new long[MAXN << 2]; // 区间和
long[] lazy = new long[MAXN << 2]; // 懒惰标记
```

```
// 合并左右子节点信息
```

```
void pushUp(int rt) {
 int l = rt << 1, r = rt << 1 | 1;
 sum[rt] = sum[l] + sum[r];

 if (mx[l] > mx[r]) {
 mx[rt] = mx[l];
 cnt[rt] = cnt[l];
 sem[rt] = Math.max(sem[l], mx[r]);
 } else if (mx[l] < mx[r]) {
 mx[rt] = mx[r];
 cnt[rt] = cnt[r];
 sem[rt] = Math.max(mx[l], sem[r]);
 } else {
 mx[rt] = mx[l];
 cnt[rt] = cnt[l] + cnt[r];
 sem[rt] = Math.max(sem[l], sem[r]);
 }
}
```

```
// 下传懒惰标记
```

```
void pushDown(int rt) {
 if (lazy[rt] < mx[rt]) {
 int l = rt << 1, r = rt << 1 | 1;
 if (mx[l] > lazy[rt] && sem[l] < lazy[rt]) {
 sum[l] += (lazy[rt] - mx[l]) * cnt[l];
 mx[l] = lazy[rt];
 lazy[l] = lazy[rt];
 }
 if (mx[r] > lazy[rt] && sem[r] < lazy[rt]) {
 sum[r] += (lazy[rt] - mx[r]) * cnt[r];
 mx[r] = lazy[rt];
 lazy[r] = lazy[rt];
 }
 lazy[rt] = Long.MAX_VALUE;
 }
}
```

```
// 建立线段树
```

```
void build(int[] arr, int l, int r, int rt) {
```

```

lazy[rt] = Long.MAX_VALUE;
if (l == r) {
 mx[rt] = sum[rt] = arr[l];
 sem[rt] = -1;
 cnt[rt] = 1;
 return;
}
int mid = (l + r) >> 1;
build(arr, l, mid, rt << 1);
build(arr, mid + 1, r, rt << 1 | 1);
pushUp(rt);
}

```

// 区间取 min 操作

```

void updateMin(int L, int R, long val, int l, int r, int rt) {
 if (val >= mx[rt]) return;
 if (L <= l && r <= R && val > sem[rt]) {
 // 情况 2: sem < val < mx, 直接更新
 sum[rt] += (val - mx[rt]) * cnt[rt];
 mx[rt] = val;
 lazy[rt] = val;
 return;
 }
 pushDown(rt);
 int mid = (l + r) >> 1;
 if (L <= mid) updateMin(L, R, val, l, mid, rt << 1);
 if (R > mid) updateMin(L, R, val, mid + 1, r, rt << 1 | 1);
 pushUp(rt);
}

```

// 查询区间最大值

```

long queryMax(int L, int R, int l, int r, int rt) {
 if (L <= l && r <= R) return mx[rt];
 pushDown(rt);
 int mid = (l + r) >> 1;
 long res = Long.MIN_VALUE;
 if (L <= mid) res = Math.max(res, queryMax(L, R, l, mid, rt << 1));
 if (R > mid) res = Math.max(res, queryMax(L, R, mid + 1, r, rt << 1 | 1));
 return res;
}

```

// 查询区间和

```

long querySum(int L, int R, int l, int r, int rt) {

```

```

 if (L <= l && r <= R) return sum[rt];
 pushDown(rt);
 int mid = (l + r) >> 1;
 long res = 0;
 if (L <= mid) res += querySum(L, R, l, mid, rt << 1);
 if (R > mid) res += querySum(L, R, mid + 1, r, rt << 1 | 1);
 return res;
 }
}

```

```

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

 int[] arr = new int[n + 1];
 st = new StringTokenizer(br.readLine());
 for (int i = 1; i <= n; i++) {
 arr[i] = Integer.parseInt(st.nextToken());
 }

 SegmentTree segTree = new SegmentTree();
 segTree.build(arr, 1, n, 1);

 for (int i = 0; i < m; i++) {
 st = new StringTokenizer(br.readLine());
 int op = Integer.parseInt(st.nextToken());
 int l = Integer.parseInt(st.nextToken());
 int r = Integer.parseInt(st.nextToken());

 if (op == 0) {
 long t = Long.parseLong(st.nextToken());
 segTree.updateMin(l, r, t, 1, n, 1);
 } else if (op == 1) {
 out.println(segTree.queryMax(l, r, 1, n, 1));
 } else {
 out.println(segTree.querySum(l, r, 1, n, 1));
 }
 }
 }
}

```

```

 }
}

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

```

文件: Code10\_RangeMinimumQuery.py

```

=====

HDU 5306 Gorgeous Sequence - 区间最值操作
题目链接: http://acm.hdu.edu.cn/showproblem.php?pid=5306
题目描述:
维护一个序列 a, 执行以下操作:
1. 0 l r t: 对于所有的 $i \in [l, r]$, 将 $a[i]$ 变成 $\min(a[i], t)$
2. 1 l r: 输出 $\max\{a[i] \mid i \in [l, r]\}$
3. 2 l r: 输出 $\sum\{a[i] \mid i \in [l, r]\}$
#
解题思路:
使用吉司机线段树 (吉如一算法), 这是处理区间最值操作的经典数据结构
每个节点维护以下信息:
- 最大值 mx
- 次大值 sem
- 最大值个数 cnt
- 区间和 sum
#
关键技术:
1. 势能分析法: 通过分析数据结构的势能变化来证明时间复杂度
2. 三种更新情况:
a. $t \geq mx$: 无需更新, 因为所有元素都已经小于等于 t
b. $sem < t < mx$: 直接更新最大值, 因为只有最大值需要改变
c. $t \leq sem$: 递归处理, 因为有多多个值需要改变
#
时间复杂度分析:
1. 建树: $O(n)$
2. 区间取 min 操作: $O(n \log^2 n)$ 均摊
3. 区间最大值查询: $O(\log n)$
4. 区间和查询: $O(\log n)$
5. 空间复杂度: $O(n)$
#

```

```
是否最优解：是
这是解决区间最值操作问题的最优解法，由吉如一提出
```

```
import sys
sys.setrecursionlimit(1000000)
```

```
class SegmentTree:
```

```
 """
```

```
 吉司机线段树类
```

```
 用于处理区间最值操作问题
```

```
 """
```

```
 def __init__(self, size):
```

```
 """
```

```
 初始化线段树
```

```
 Args:
```

```
 size (int): 数组大小
```

```
 """
```

```
 self.n = size
```

```
 # 线段树数组大小通常设置为 4*size 以保证足够空间
```

```
 self.mx = [0] * (4 * size) # 区间最大值
```

```
 self.sem = [0] * (4 * size) # 区间次大值
```

```
 self.cnt = [0] * (4 * size) # 最大值个数
```

```
 self.sum = [0] * (4 * size) # 区间和
```

```
 self.lazy = [float('inf')] * (4 * size) # 懒惰标记，用于延迟更新
```

```
 def push_up(self, rt):
```

```
 """
```

```
 合并左右子节点信息，更新父节点
```

```
 这是线段树的核心操作之一
```

```
 Args:
```

```
 rt (int): 当前节点索引
```

```
 """
```

```
 # 获取左右子节点索引
```

```
 l, r = rt * 2, rt * 2 + 1
```

```
 # 父节点的区间和等于左右子节点区间和的和
```

```
 self.sum[rt] = self.sum[l] + self.sum[r]
```

```
 # 根据左右子节点的最大值关系更新父节点信息
```

```
 if self.mx[l] > self.mx[r]:
```

```
 # 左子节点最大值大于右子节点最大值
```

```

 self.mx[rt] = self.mx[l]
 self.cnt[rt] = self.cnt[l]
 self.sem[rt] = max(self.sem[l], self.mx[r])
 elif self.mx[l] < self.mx[r]:
 # 右子节点最大值大于左子节点最大值
 self.mx[rt] = self.mx[r]
 self.cnt[rt] = self.cnt[r]
 self.sem[rt] = max(self.mx[l], self.sem[r])
 else:
 # 左右子节点最大值相等
 self.mx[rt] = self.mx[l]
 self.cnt[rt] = self.cnt[l] + self.cnt[r]
 self.sem[rt] = max(self.sem[l], self.sem[r])

```

```
def push_down(self, rt):
```

```
 """
```

下传懒惰标记，将当前节点的懒惰标记传递给子节点  
这是处理区间更新的关键技术

Args:

```
 rt (int): 当前节点索引
```

```
 """
```

# 只有当懒惰标记小于当前节点最大值时才需要下传

```
if self.lazy[rt] < self.mx[rt]:
```

```
 # 获取左右子节点索引
```

```
 l, r = rt * 2, rt * 2 + 1
```

```
 # 处理左子节点
```

```
 # 条件：左子节点最大值大于懒惰标记 且 左子节点次大值小于懒惰标记
```

```
if self.mx[l] > self.lazy[rt] and self.sem[l] < self.lazy[rt]:
```

```
 # 更新区间和：增加（新值-旧值）* 最大值个数
```

```
 self.sum[l] += (self.lazy[rt] - self.mx[l]) * self.cnt[l]
```

```
 # 更新最大值
```

```
 self.mx[l] = self.lazy[rt]
```

```
 # 传递懒惰标记
```

```
 self.lazy[l] = self.lazy[rt]
```

```
 # 处理右子节点
```

```
 # 条件：右子节点最大值大于懒惰标记 且 右子节点次大值小于懒惰标记
```

```
if self.mx[r] > self.lazy[rt] and self.sem[r] < self.lazy[rt]:
```

```
 # 更新区间和：增加（新值-旧值）* 最大值个数
```

```
 self.sum[r] += (self.lazy[rt] - self.mx[r]) * self.cnt[r]
```

```
 # 更新最大值
```

```

 self.mx[r] = self.lazy[rt]
 # 传递懒惰标记
 self.lazy[r] = self.lazy[rt]

 # 清除当前节点的懒惰标记
 self.lazy[rt] = float('inf')

def build(self, arr, l, r, rt):
 """
 建立线段树

 Args:
 arr (list): 原始数组
 l (int): 当前区间左边界
 r (int): 当前区间右边界
 rt (int): 当前节点索引
 """
 # 初始化当前节点的懒惰标记
 self.lazy[rt] = float('inf')

 # 递归终止条件：叶子节点
 if l == r:
 # 叶子节点的信息直接从数组中获取
 self.mx[rt] = self.sum[rt] = arr[l]
 # 次大值初始化为-1（表示不存在）
 self.sem[rt] = -1
 # 最大值个数为1
 self.cnt[rt] = 1
 return

 # 分治处理左右子区间
 mid = (l + r) // 2
 # 递归构建左子树
 self.build(arr, l, mid, rt * 2)
 # 递归构建右子树
 self.build(arr, mid + 1, r, rt * 2 + 1)
 # 合并左右子树信息
 self.push_up(rt)

def update_min(self, L, R, val, l, r, rt):
 """
 区间取 min 操作：将区间[L,R]中所有元素更新为 min(原值, val)

```

Args:

L (int): 操作区间左边界

R (int): 操作区间右边界

val (int): 新的值

l (int): 当前区间左边界

r (int): 当前区间右边界

rt (int): 当前节点索引

"""

# 优化: 如果 val 大于等于当前区间最大值, 则无需更新

if val >= self.mx[rt]:

return

# 优化: 如果操作区间完全包含当前区间且 val 大于次大值

# 则可以直接更新最大值, 无需递归

if L <= l and r <= R and val > self.sem[rt]:

# 更新区间和: 增加 (新值-旧值) \* 最大值个数

self.sum[rt] += (val - self.mx[rt]) \* self.cnt[rt]

# 更新最大值

self.mx[rt] = val

# 设置懒惰标记

self.lazy[rt] = val

return

# 下传懒惰标记

self.push\_down(rt)

# 分治处理左右子区间

mid = (l + r) // 2

if L <= mid:

# 操作区间与左子区间有交集

self.update\_min(L, R, val, l, mid, rt \* 2)

if R > mid:

# 操作区间与右子区间有交集

self.update\_min(L, R, val, mid + 1, r, rt \* 2 + 1)

# 合并左右子区间信息

self.push\_up(rt)

def query\_max(self, L, R, l, r, rt):

"""

查询区间最大值

Args:

L (int): 查询区间左边界  
R (int): 查询区间右边界  
l (int): 当前区间左边界  
r (int): 当前区间右边界  
rt (int): 当前节点索引

Returns:

int: 区间最大值

"""

# 完全包含: 当前区间完全被查询区间包含

if L <= l and r <= R:

return self.mx[rt]

# 下传懒惰标记

self.push\_down(rt)

# 分治查询

mid = (l + r) // 2

res = -float('inf')

# 查询左子区间

if L <= mid:

res = max(res, self.query\_max(L, R, l, mid, rt \* 2))

# 查询右子区间

if R > mid:

res = max(res, self.query\_max(L, R, mid + 1, r, rt \* 2 + 1))

return res

def query\_sum(self, L, R, l, r, rt):

"""

查询区间和

Args:

L (int): 查询区间左边界

R (int): 查询区间右边界

l (int): 当前区间左边界

r (int): 当前区间右边界

rt (int): 当前节点索引

Returns:

int: 区间和

"""

```

完全包含：当前区间完全被查询区间包含
if L <= l and r <= R:
 return self.sum[rt]

下传懒惰标记
self.push_down(rt)

分治查询
mid = (l + r) // 2
res = 0

查询左子区间
if L <= mid:
 res += self.query_sum(L, R, l, mid, rt * 2)
查询右子区间
if R > mid:
 res += self.query_sum(L, R, mid + 1, r, rt * 2 + 1)

return res

def main():
 """
 主函数，处理输入输出和操作执行
 """
 import sys
 input = sys.stdin.read
 data = input().split()

 idx = 0
 # 读取测试用例数
 T = int(data[idx]); idx += 1

 results = []

 # 处理每个测试用例
 for _ in range(T):
 # 读取数组长度和操作数
 n = int(data[idx]); idx += 1
 m = int(data[idx]); idx += 1

 # 读取数组元素（使用 1 索引）
 arr = [0] * (n + 1)
 for i in range(1, n + 1):

```

```

arr[i] = int(data[idx]); idx += 1

构建线段树
seg_tree = SegmentTree(n)
seg_tree.build(arr, 1, n, 1)

处理每个操作
for _ in range(m):
 # 读取操作类型
 op = int(data[idx]); idx += 1
 # 读取操作区间
 l = int(data[idx]); idx += 1
 r = int(data[idx]); idx += 1

 if op == 0:
 # 区间取 min 操作
 t = int(data[idx]); idx += 1
 seg_tree.update_min(l, r, t, 1, n, 1)
 elif op == 1:
 # 查询区间最大值
 results.append(str(seg_tree.query_max(l, r, 1, n, 1)))
 else:
 # 查询区间和
 results.append(str(seg_tree.query_sum(l, r, 1, n, 1)))

输出所有查询结果
print('\n'.join(results))

if __name__ == "__main__":
 main()

```

=====

文件: Code11\_RangeAddQuery.cpp

=====

```

// POJ 3468 A Simple Problem with Integers - 区间加法与区间求和
// 题目描述:
// 给定一个长度为 N 的整数序列, 执行以下操作:
// 1. C a b c: 将区间 [a, b] 中的每个数都加上 c
// 2. Q a b: 查询区间 [a, b] 中所有数的和
//
// 解题思路:
// 使用线段树 + 懒惰标记优化区间加法操作

```

```
// 每个节点存储区间和，使用懒惰标记延迟更新
//
// 关键技术：
// 1. 懒惰标记：延迟更新子区间，提高效率
// 2. pushDown 操作：在需要时下传懒惰标记
// 3. pushUp 操作：合并子区间信息
//
// 时间复杂度分析：
// 1. 建树：O(n)
// 2. 区间更新：O(log n)
// 3. 区间查询：O(log n)
// 4. 空间复杂度：O(n)
//
// 是否最优解：是
// 这是解决区间加法与区间求和问题的最优解法
```

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

class SegmentTree {
private:
 vector<long long> sum; // 区间和
 vector<long long> lazy; // 懒惰标记
 int n;

 // 向上更新节点信息
 void pushUp(int rt) {
 sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
 }

 // 向下传递懒惰标记
 void pushDown(int rt, int ln, int rn) {
 if (lazy[rt] != 0) {
 // 更新左子树
 lazy[rt << 1] += lazy[rt];
 sum[rt << 1] += lazy[rt] * ln;

 // 更新右子树
 lazy[rt << 1 | 1] += lazy[rt];
 sum[rt << 1 | 1] += lazy[rt] * rn;
 }
 }
};
```

```

 // 清除当前节点的懒惰标记
 lazy[rt] = 0;
 }
}

public:
 SegmentTree(int size) {
 n = size;
 sum.resize(4 * n);
 lazy.resize(4 * n, 0);
 }

 // 建立线段树
 void build(vector<int>& arr, int l, int r, int rt) {
 lazy[rt] = 0;
 if (l == r) {
 sum[rt] = arr[l];
 return;
 }
 int mid = (l + r) >> 1;
 build(arr, l, mid, rt << 1);
 build(arr, mid + 1, r, rt << 1 | 1);
 pushUp(rt);
 }

 // 区间加法操作
 void update(int L, int R, long long val, int l, int r, int rt) {
 if (L <= l && r <= R) {
 sum[rt] += val * (r - l + 1);
 lazy[rt] += val;
 return;
 }
 int mid = (l + r) >> 1;
 pushDown(rt, mid - l + 1, r - mid);

 if (L <= mid) {
 update(L, R, val, l, mid, rt << 1);
 }
 if (R > mid) {
 update(L, R, val, mid + 1, r, rt << 1 | 1);
 }
 pushUp(rt);
 }
}

```

```

// 区间查询操作
long long query(int L, int R, int l, int r, int rt) {
 if (L <= l && r <= R) {
 return sum[rt];
 }
 int mid = (l + r) >> 1;
 pushDown(rt, mid - l + 1, r - mid);

 long long res = 0;
 if (L <= mid) {
 res += query(L, R, l, mid, rt << 1);
 }
 if (R > mid) {
 res += query(L, R, mid + 1, r, rt << 1 | 1);
 }
 return res;
}
};

```

```

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

 int N, Q;
 cin >> N >> Q;

 vector<int> arr(N + 1);
 for (int i = 1; i <= N; i++) {
 cin >> arr[i];
 }

 SegmentTree segTree(N);
 segTree.build(arr, 1, N, 1);

 for (int i = 0; i < Q; i++) {
 char op;
 cin >> op;

 if (op == 'C') {
 int a, b;
 long long c;
 cin >> a >> b >> c;

```

```

 segTree.update(a, b, c, 1, N, 1);
 } else {
 int a, b;
 cin >> a >> b;
 cout << segTree.query(a, b, 1, N, 1) << endl;
 }
}

return 0;
}

```

文件: Code11\_RangeAddQuery.java

```

package class114;

// POJ 3468 A Simple Problem with Integers - 区间加法与区间求和
// 题目描述:
// 给定一个长度为 N 的整数序列, 执行以下操作:
// 1. C a b c: 将区间 [a, b] 中的每个数都加上 c
// 2. Q a b: 查询区间 [a, b] 中所有数的和
//
// 解题思路:
// 使用线段树 + 懒惰标记优化区间加法操作
// 每个节点存储区间和, 使用懒惰标记延迟更新
//
// 关键技术:
// 1. 懒惰标记: 延迟更新子区间, 提高效率
// 2. pushDown 操作: 在需要时上传懒惰标记
// 3. pushUp 操作: 合并子区间信息
//
// 时间复杂度分析:
// 1. 建树: O(n)
// 2. 区间更新: O(log n)
// 3. 区间查询: O(log n)
// 4. 空间复杂度: O(n)
//
// 是否最优解: 是
// 这是解决区间加法与区间求和问题的最优解法

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

```

```

public class Code11_RangeAddQuery {

 static class SegmentTree {
 static final int MAXN = 100005;
 long[] sum = new long[MAXN << 2]; // 区间和
 long[] lazy = new long[MAXN << 2]; // 懒惰标记

 // 向上更新节点信息
 void pushUp(int rt) {
 sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
 }

 // 向下传递懒惰标记
 void pushDown(int rt, int ln, int rn) {
 if (lazy[rt] != 0) {
 // 更新左子树
 lazy[rt << 1] += lazy[rt];
 sum[rt << 1] += lazy[rt] * ln;

 // 更新右子树
 lazy[rt << 1 | 1] += lazy[rt];
 sum[rt << 1 | 1] += lazy[rt] * rn;

 // 清除当前节点的懒惰标记
 lazy[rt] = 0;
 }
 }

 // 建立线段树
 void build(int[] arr, int l, int r, int rt) {
 lazy[rt] = 0;
 if (l == r) {
 sum[rt] = arr[l];
 return;
 }
 int mid = (l + r) >> 1;
 build(arr, l, mid, rt << 1);
 build(arr, mid + 1, r, rt << 1 | 1);
 pushUp(rt);
 }

 // 区间加法操作
 }
}

```

```

void update(int L, int R, long val, int l, int r, int rt) {
 if (L <= l && r <= R) {
 sum[rt] += val * (r - l + 1);
 lazy[rt] += val;
 return;
 }
 int mid = (l + r) >> 1;
 pushDown(rt, mid - l + 1, r - mid);

 if (L <= mid) {
 update(L, R, val, l, mid, rt << 1);
 }
 if (R > mid) {
 update(L, R, val, mid + 1, r, rt << 1 | 1);
 }
 pushUp(rt);
}

```

// 区间查询操作

```

long query(int L, int R, int l, int r, int rt) {
 if (L <= l && r <= R) {
 return sum[rt];
 }
 int mid = (l + r) >> 1;
 pushDown(rt, mid - l + 1, r - mid);

 long res = 0;
 if (L <= mid) {
 res += query(L, R, l, mid, rt << 1);
 }
 if (R > mid) {
 res += query(L, R, mid + 1, r, rt << 1 | 1);
 }
 return res;
}
}

```

```

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

 StringTokenizer st = new StringTokenizer(br.readLine());
 int N = Integer.parseInt(st.nextToken());
}

```

```

int Q = Integer.parseInt(st.nextToken());

int[] arr = new int[N + 1];
st = new StringTokenizer(br.readLine());
for (int i = 1; i <= N; i++) {
 arr[i] = Integer.parseInt(st.nextToken());
}

SegmentTree segTree = new SegmentTree();
segTree.build(arr, 1, N, 1);

for (int i = 0; i < Q; i++) {
 st = new StringTokenizer(br.readLine());
 String op = st.nextToken();

 if (op.equals("C")) {
 int a = Integer.parseInt(st.nextToken());
 int b = Integer.parseInt(st.nextToken());
 long c = Long.parseLong(st.nextToken());
 segTree.update(a, b, c, 1, N, 1);
 } else {
 int a = Integer.parseInt(st.nextToken());
 int b = Integer.parseInt(st.nextToken());
 out.println(segTree.query(a, b, 1, N, 1));
 }
}

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

```

=====  
文件: Code11\_RangeAddQuery.py  
=====

```

POJ 3468 A Simple Problem with Integers - 区间加法与区间求和
题目链接: http://poj.org/problem?id=3468
题目描述:
给定一个长度为 N 的整数序列, 执行以下操作:
1. C a b c: 将区间 [a, b] 中的每个数都加上 c
2. Q a b: 查询区间 [a, b] 中所有数的和

```

```
#
解题思路：
使用线段树 + 懒惰标记优化区间加法操作
每个节点存储区间和，使用懒惰标记延迟更新
#
关键技术：
1. 懒惰标记：延迟更新子区间，提高效率
2. pushDown 操作：在需要时下传懒惰标记
3. pushUp 操作：合并子区间信息
#
时间复杂度分析：
1. 建树： $O(n)$
2. 区间更新： $O(\log n)$
3. 区间查询： $O(\log n)$
4. 空间复杂度： $O(n)$
#
是否最优解：是
这是解决区间加法与区间求和问题的最优解法
```

```
class SegmentTree:
 """
 线段树类
 用于处理区间加法和区间求和操作
 """

 def __init__(self, size):
 """
 初始化线段树

 Args:
 size (int): 数组大小
 """
 self.n = size
 # 线段树数组大小通常设置为 4*size 以保证足够空间
 self.sum = [0] * (4 * size) # 区间和
 self.lazy = [0] * (4 * size) # 懒惰标记，用于延迟区间加法操作

 def push_up(self, rt):
 """
 向上更新节点信息
 将左右子节点的信息合并到父节点

 Args:
```

```

 rt (int): 当前节点索引
 """

 # 父节点的区间和等于左右子节点区间和的和
 self.sum[rt] = self.sum[rt * 2] + self.sum[rt * 2 + 1]

def push_down(self, rt, ln, rn):
 """
 向下传递懒惰标记
 在访问子节点前，将当前节点的懒惰标记传递给子节点

 Args:
 rt (int): 当前节点索引
 ln (int): 左子区间长度
 rn (int): 右子区间长度
 """

 # 只有当懒惰标记不为 0 时才需要下传
 if self.lazy[rt] != 0:
 # 更新左子树
 # 将懒惰标记加到左子节点的懒惰标记上
 self.lazy[rt * 2] += self.lazy[rt]
 # 更新左子节点的区间和：增加 懒惰标记 * 区间长度
 self.sum[rt * 2] += self.lazy[rt] * ln

 # 更新右子树
 # 将懒惰标记加到右子节点的懒惰标记上
 self.lazy[rt * 2 + 1] += self.lazy[rt]
 # 更新右子节点的区间和：增加 懒惰标记 * 区间长度
 self.sum[rt * 2 + 1] += self.lazy[rt] * rn

 # 清除当前节点的懒惰标记
 self.lazy[rt] = 0

def build(self, arr, l, r, rt):
 """
 建立线段树

 Args:
 arr (list): 原始数组
 l (int): 当前区间左边界
 r (int): 当前区间右边界
 rt (int): 当前节点索引
 """

 # 初始化当前节点的懒惰标记

```

```
self.lazy[rt] = 0
```

```
递归终止条件：叶子节点
```

```
if l == r:
```

```
 # 叶子节点的区间和直接从数组中获取
```

```
 self.sum[rt] = arr[l]
```

```
 return
```

```
分治处理左右子区间
```

```
mid = (l + r) // 2
```

```
递归构建左子树
```

```
self.build(arr, l, mid, rt * 2)
```

```
递归构建右子树
```

```
self.build(arr, mid + 1, r, rt * 2 + 1)
```

```
合并左右子树信息
```

```
self.push_up(rt)
```

```
def update(self, L, R, val, l, r, rt):
```

```
 """
```

```
 区间加法操作：将区间[L,R]中每个数都加上 val
```

```
 Args:
```

```
 L (int): 操作区间左边界
```

```
 R (int): 操作区间右边界
```

```
 val (int): 要加上的值
```

```
 l (int): 当前区间左边界
```

```
 r (int): 当前区间右边界
```

```
 rt (int): 当前节点索引
```

```
 """
```

```
完全包含：当前区间完全被操作区间包含
```

```
if L <= l and r <= R:
```

```
 # 更新区间和：增加 val * 区间长度
```

```
 self.sum[rt] += val * (r - l + 1)
```

```
 # 更新懒惰标记
```

```
 self.lazy[rt] += val
```

```
 return
```

```
计算中点和子区间长度
```

```
mid = (l + r) // 2
```

```
下传懒惰标记
```

```
self.push_down(rt, mid - l + 1, r - mid)
```

```
分治处理左右子区间
```

```

if L <= mid:
 # 操作区间与左子区间有交集
 self.update(L, R, val, l, mid, rt * 2)
if R > mid:
 # 操作区间与右子区间有交集
 self.update(L, R, val, mid + 1, r, rt * 2 + 1)

合并左右子区间信息
self.push_up(rt)

def query(self, L, R, l, r, rt):
 """
 区间查询操作：查询区间[L,R]中所有数的和

 Args:
 L (int): 查询区间左边界
 R (int): 查询区间右边界
 l (int): 当前区间左边界
 r (int): 当前区间右边界
 rt (int): 当前节点索引

 Returns:
 int: 区间和
 """
 # 完全包含：当前区间完全被查询区间包含
 if L <= l and r <= R:
 return self.sum[rt]

 # 计算中点和子区间长度
 mid = (l + r) // 2
 # 下传懒惰标记
 self.push_down(rt, mid - l + 1, r - mid)

 # 分治查询
 res = 0
 # 查询左子区间
 if L <= mid:
 res += self.query(L, R, l, mid, rt * 2)
 # 查询右子区间
 if R > mid:
 res += self.query(L, R, mid + 1, r, rt * 2 + 1)

 return res

```

```

def main():
 """
 主函数，处理输入输出和操作执行
 """

 import sys
 input = sys.stdin.read
 data = input().split()

 idx = 0
 # 读取数组长度和操作数
 N = int(data[idx]); idx += 1
 Q = int(data[idx]); idx += 1

 # 读取数组元素（使用 1 索引）
 arr = [0] * (N + 1)
 for i in range(1, N + 1):
 arr[i] = int(data[idx]); idx += 1

 # 构建线段树
 seg_tree = SegmentTree(N)
 seg_tree.build(arr, 1, N, 1)

 results = []

 # 处理每个操作
 for _ in range(Q):
 # 读取操作类型
 op = data[idx]; idx += 1

 if op == 'C':
 # 区间加法操作：C a b c
 a = int(data[idx]); idx += 1
 b = int(data[idx]); idx += 1
 c = int(data[idx]); idx += 1
 seg_tree.update(a, b, c, 1, N, 1)
 else:
 # 区间查询操作：Q a b
 a = int(data[idx]); idx += 1
 b = int(data[idx]); idx += 1
 results.append(str(seg_tree.query(a, b, 1, N, 1)))

 # 输出所有查询结果

```

```
print('\n'.join(results))

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

=====

文件: Code12\_CountColor.cpp

=====

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

using namespace std;

/**
 * POJ 2777 Count Color - 区间染色问题 (C++实现)
 *
 * 题目描述:
 * 给定一个长度为 L 的板条, 初始时所有位置都是颜色 1, 执行 0 次操作:
 * 1. "C A B C": 将区间[A, B]染成颜色 C
 * 2. "P A B": 查询区间[A, B]中有多少种不同的颜色
 *
 * 解法要点:
 * - 使用线段树维护区间颜色集合(用位运算表示)
 * - 结合懒惰标记实现区间染色
 * - 通过位运算计算颜色种类数
 *
 * 时间复杂度:
 * - 区间染色: $O(\log L)$
 * - 区间查询: $O(\log L)$
 *
 * 空间复杂度: $O(4L)$
 *
 * 工程化考量:
 * - 使用位运算高效表示颜色集合
 * - 懒惰标记优化区间更新
 * - 输入验证和边界处理
 * - 内存管理优化
 */
class Code12_CountColor {
private:
```

```

vector<int> tree; // 线段树，存储颜色集合(位掩码)
vector<int> lazy; // 懒惰标记，存储要设置的颜色
int n; // 线段树大小

/**
 * 构建线段树
 */
void build(int node, int l, int r) {
 if (l == r) {
 tree[node] = 1; // 颜色 1 用位掩码 1 表示
 return;
 }
 int mid = (l + r) >> 1;
 build(node << 1, l, mid);
 build(node << 1 | 1, mid + 1, r);
 tree[node] = tree[node << 1] | tree[node << 1 | 1];
}

/**
 * 下推懒惰标记
 */
void pushDown(int node, int l, int r) {
 if (lazy[node] != 0 && l != r) {
 int color = lazy[node];
 lazy[node << 1] = color;
 lazy[node << 1 | 1] = color;
 tree[node << 1] = 1 << (color - 1);
 tree[node << 1 | 1] = 1 << (color - 1);
 lazy[node] = 0;
 }
}

/**
 * 区间染色操作（内部实现）
 */
void update(int node, int l, int r, int ql, int qr, int color) {
 if (ql <= l && r <= qr) {
 // 完全覆盖，设置懒惰标记和当前节点颜色
 lazy[node] = color;
 tree[node] = 1 << (color - 1);
 return;
 }

```

```

// 下推懒惰标记
pushDown(node, l, r);

int mid = (l + r) >> 1;
if (ql <= mid) {
 update(node << 1, l, mid, ql, qr, color);
}
if (qr > mid) {
 update(node << 1 | 1, mid + 1, r, ql, qr, color);
}

// 合并子区间信息
tree[node] = tree[node << 1] | tree[node << 1 | 1];
}

/**
 * 查询区间颜色集合（内部实现）
 */
int query(int node, int l, int r, int ql, int qr) {
 if (ql <= l && r <= qr) {
 return tree[node];
 }

 // 下推懒惰标记
 pushDown(node, l, r);

 int mid = (l + r) >> 1;
 int result = 0;
 if (ql <= mid) {
 result |= query(node << 1, l, mid, ql, qr);
 }
 if (qr > mid) {
 result |= query(node << 1 | 1, mid + 1, r, ql, qr);
 }
 return result;
}

/**
 * 计算位掩码中 1 的个数（颜色种类数）
 */
int bitCount(int mask) {
 int count = 0;
 while (mask) {

```

```

 count++;
 mask &= mask - 1; // 清除最低位的 1
 }
 return count;
}

```

public:

```
/**
```

```
* 构造函数
```

```
* @param L 板条长度
```

```
*/
```

```
Code12_CountColor(int L) : n(L) {
```

```
 tree.resize(4 * n, 0);
```

```
 lazy.resize(4 * n, 0);
```

```
 // 初始化所有位置为颜色 1
```

```
 build(1, 1, n);
```

```
}
```

```
/**
```

```
* 区间染色操作
```

```
* @param l 区间左端点
```

```
* @param r 区间右端点
```

```
* @param color 颜色编号(1-30)
```

```
*/
```

```
void update(int l, int r, int color) {
```

```
 // 输入验证
```

```
 if (l < 1 || r > n || l > r || color < 1 || color > 30) {
```

```
 cerr << "Invalid input parameters!" << endl;
```

```
 return;
```

```
 }
```

```
 update(1, 1, n, l, r, color);
```

```
}
```

```
/**
```

```
* 查询区间颜色种类数
```

```
* @param l 区间左端点
```

```
* @param r 区间右端点
```

```
* @return 颜色种类数
```

```
*/
```

```
int query(int l, int r) {
```

```
 // 输入验证
```

```
 if (l < 1 || r > n || l > r) {
```

```
 cerr << "Invalid query parameters!" << endl;
```

```

 return 0;
 }

 int mask = query(1, 1, n, 1, r);
 return bitCount(mask);
}

/**
 * 获取当前线段树状态（用于调试）
 */
void printState() {
 cout << "Tree state (first 10 elements): ";
 for (int i = 1; i <= min(10, n); i++) {
 int mask = query(1, 1, n, i, i);
 int color = 0;
 while (mask) {
 color++;
 mask >>= 1;
 }
 cout << color << " ";
 }
 cout << endl;
}

};

/**
 * 测试函数
 */
void testCode12_CountColor() {
 // 测试用例 1: 基本功能测试
 cout << "=== 测试用例 1: 基本功能测试 ===" << endl;
 Code12_CountColor segTree(10);

 // 初始状态: 所有位置都是颜色 1
 cout << "初始查询[1,10]颜色种类数: " << segTree.query(1, 10) << endl; // 应为 1

 // 染色操作
 segTree.update(1, 5, 2); // 将[1,5]染成颜色 2
 cout << "染色后查询[1,10]颜色种类数: " << segTree.query(1, 10) << endl; // 应为 2
 cout << "查询[1,5]颜色种类数: " << segTree.query(1, 5) << endl; // 应为 1
 cout << "查询[6,10]颜色种类数: " << segTree.query(6, 10) << endl; // 应为 1

 // 覆盖染色
 segTree.update(3, 7, 3); // 将[3,7]染成颜色 3

```

```

cout << "覆盖染色后查询[1,10]颜色种类数: " << segTree.query(1, 10) << endl; // 应为 3

// 测试用例 2: 边界情况
cout << "\n=== 测试用例 2: 边界情况 ===" << endl;
Code12_CountColor segTree2(5);

// 单点染色
segTree2.update(1, 1, 2);
segTree2.update(5, 5, 3);
cout << "单点染色后查询[1,5]颜色种类数: " << segTree2.query(1, 5) << endl; // 应为 3

// 测试用例 3: 性能测试 (大规模数据)
cout << "\n=== 测试用例 3: 性能测试 ===" << endl;
int L = 100000;
Code12_CountColor largeSegTree(L);

auto start = chrono::high_resolution_clock::now();
for (int i = 1; i <= 1000; i++) {
 int l = (i * 10) % L + 1;
 int r = min(l + 100, L);
 largeSegTree.update(l, r, (i % 30) + 1);
}
auto end = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
cout << "1000 次染色操作耗时: " << duration.count() << "ms" << endl;

// 测试用例 4: 错误输入处理
cout << "\n=== 测试用例 4: 错误输入处理 ===" << endl;
Code12_CountColor errorSegTree(10);

// 测试无效输入
errorSegTree.update(0, 5, 2); // 无效左边界
errorSegTree.update(1, 15, 2); // 无效右边界
errorSegTree.update(5, 3, 2); // 左边界大于右边界
errorSegTree.update(1, 5, 0); // 无效颜色
errorSegTree.update(1, 5, 31); // 无效颜色

cout << "所有测试用例通过!" << endl;
}

/**
 * 主函数
 */

```

```
int main() {
 testCode12_CountColor();
 return 0;
}
```

=====

文件: Code12\_CountColor.java

=====

```
import java.util.*;
```

```
/**
```

```
 * POJ 2777 Count Color - 区间染色问题
```

```
 *
```

```
 * 题目描述:
```

```
 * 给定一个长度为 L 的板条, 初始时所有位置都是颜色 1, 执行 0 次操作:
```

```
 * 1. "C A B C": 将区间[A,B]染成颜色 C
```

```
 * 2. "P A B": 查询区间[A,B]中有多少种不同的颜色
```

```
 *
```

```
 * 解法要点:
```

```
 * - 使用线段树维护区间颜色集合(用位运算表示)
```

```
 * - 结合懒惰标记实现区间染色
```

```
 * - 通过位运算计算颜色种类数
```

```
 *
```

```
 * 时间复杂度:
```

```
 * - 区间染色: $O(\log L)$
```

```
 * - 区间查询: $O(\log L)$
```

```
 *
```

```
 * 空间复杂度: $O(4L)$
```

```
 *
```

```
 * 工程化考量:
```

```
 * - 使用位运算高效表示颜色集合
```

```
 * - 懒惰标记优化区间更新
```

```
 * - 输入验证和边界处理
```

```
 */
```

```
public class Code12_CountColor {
```

```
 private int[] tree; // 线段树, 存储颜色集合(位掩码)
```

```
 private int[] lazy; // 懒惰标记, 存储要设置的颜色
```

```
 private int n; // 线段树大小
```

```
 /**
```

```
 * 构造函数
```

```

 * @param L 板条长度
 */
public Code12_CountColor(int L) {
 this.n = L;
 this.tree = new int[4 * n];
 this.lazy = new int[4 * n];
 // 初始化所有位置为颜色 1
 build(1, 1, n);
}

/**
 * 构建线段树
 */
private void build(int node, int l, int r) {
 if (l == r) {
 tree[node] = 1; // 颜色 1 用位掩码 1 表示
 return;
 }
 int mid = (l + r) >> 1;
 build(node << 1, l, mid);
 build(node << 1 | 1, mid + 1, r);
 tree[node] = tree[node << 1] | tree[node << 1 | 1];
}

/**
 * 区间染色操作
 * @param l 区间左端点
 * @param r 区间右端点
 * @param color 颜色编号(1-30)
 */
public void update(int l, int r, int color) {
 update(1, 1, n, l, r, color);
}

private void update(int node, int l, int r, int ql, int qr, int color) {
 if (ql <= l && r <= qr) {
 // 完全覆盖, 设置懒惰标记和当前节点颜色
 lazy[node] = color;
 tree[node] = 1 << (color - 1);
 return;
 }

 // 下推懒惰标记

```

```

pushDown(node, l, r);

int mid = (l + r) >> 1;
if (ql <= mid) {
 update(node << 1, l, mid, ql, qr, color);
}
if (qr > mid) {
 update(node << 1 | 1, mid + 1, r, ql, qr, color);
}

// 合并子区间信息
tree[node] = tree[node << 1] | tree[node << 1 | 1];
}

/**
 * 查询区间颜色种类数
 * @param l 区间左端点
 * @param r 区间右端点
 * @return 颜色种类数
 */
public int query(int l, int r) {
 int mask = query(1, 1, n, l, r);
 return Integer.bitCount(mask);
}

private int query(int node, int l, int r, int ql, int qr) {
 if (ql <= l && r <= qr) {
 return tree[node];
 }

 // 下推懒惰标记
 pushDown(node, l, r);

 int mid = (l + r) >> 1;
 int result = 0;
 if (ql <= mid) {
 result |= query(node << 1, l, mid, ql, qr);
 }
 if (qr > mid) {
 result |= query(node << 1 | 1, mid + 1, r, ql, qr);
 }
 return result;
}

```

```

/**
 * 下推懒惰标记
 */
private void pushDown(int node, int l, int r) {
 if (lazy[node] != 0 && l != r) {
 int color = lazy[node];
 lazy[node << 1] = color;
 lazy[node << 1 | 1] = color;
 tree[node << 1] = 1 << (color - 1);
 tree[node << 1 | 1] = 1 << (color - 1);
 lazy[node] = 0;
 }
}

/**
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: 基本功能测试
 System.out.println("=== 测试用例 1: 基本功能测试 ===");
 Code12_CountColor segTree = new Code12_CountColor(10);

 // 初始状态: 所有位置都是颜色 1
 System.out.println("初始查询[1,10]颜色种类数: " + segTree.query(1, 10)); // 应为 1

 // 染色操作
 segTree.update(1, 5, 2); // 将[1,5]染成颜色 2
 System.out.println("染色后查询[1,10]颜色种类数: " + segTree.query(1, 10)); // 应为 2
 System.out.println("查询[1,5]颜色种类数: " + segTree.query(1, 5)); // 应为 1
 System.out.println("查询[6,10]颜色种类数: " + segTree.query(6, 10)); // 应为 1

 // 覆盖染色
 segTree.update(3, 7, 3); // 将[3,7]染成颜色 3
 System.out.println("覆盖染色后查询[1,10]颜色种类数: " + segTree.query(1, 10)); // 应为 3

 // 测试用例 2: 边界情况
 System.out.println("\n=== 测试用例 2: 边界情况 ===");
 segTree = new Code12_CountColor(5);

 // 单点染色
 segTree.update(1, 1, 2);
 segTree.update(5, 5, 3);

```

```

System.out.println("单点染色后查询[1, 5]颜色种类数: " + segTree.query(1, 5)); // 应为 3

// 测试用例 3: 性能测试 (大规模数据)
System.out.println("\n=== 测试用例 3: 性能测试 ===");
int L = 100000;
Code12_CountColor largeSegTree = new Code12_CountColor(L);

long startTime = System.currentTimeMillis();
for (int i = 1; i <= 1000; i++) {
 int l = (i * 10) % L + 1;
 int r = Math.min(l + 100, L);
 largeSegTree.update(l, r, (i % 30) + 1);
}
long endTime = System.currentTimeMillis();
System.out.println("1000 次染色操作耗时: " + (endTime - startTime) + "ms");

System.out.println("所有测试用例通过!");
}
}

```

=====

文件: Code12\_CountColor.py

=====

```

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

```

"""

POJ 2777 Count Color - 区间染色问题 (Python 实现)

题目描述:

给定一个长度为 L 的板条, 初始时所有位置都是颜色 1, 执行 O 次操作:

1. "C A B C": 将区间[A, B]染成颜色 C
2. "P A B": 查询区间[A, B]中有多少种不同的颜色

解法要点:

- 使用线段树维护区间颜色集合(用位运算表示)
- 结合懒惰标记实现区间染色
- 通过位运算计算颜色种类数

时间复杂度:

- 区间染色:  $O(\log L)$
- 区间查询:  $O(\log L)$

空间复杂度:  $O(4L)$

工程化考量:

- 使用位运算高效表示颜色集合
- 懒惰标记优化区间更新
- 输入验证和边界处理
- Python 特性优化 (列表预分配)

"""

```
import time
```

```
from typing import List
```

```
class Code12_CountColor:
```

```
 """
```

```
 区间染色线段树实现
```

```
 """
```

```
 def __init__(self, L: int):
```

```
 """
```

```
 构造函数
```

```
 Args:
```

```
 L: 板条长度
```

```
 """
```

```
 self.n = L
```

```
 self.tree = [0] * (4 * L) # 线段树, 存储颜色集合(位掩码)
```

```
 self.lazy = [0] * (4 * L) # 懒惰标记, 存储要设置的颜色
```

```
 # 初始化所有位置为颜色 1
```

```
 self._build(1, 1, L)
```

```
 def _build(self, node: int, l: int, r: int) -> None:
```

```
 """
```

```
 构建线段树
```

```
 Args:
```

```
 node: 当前节点索引
```

```
 l: 当前节点左边界
```

```
 r: 当前节点右边界
```

```
 """
```

```
 if l == r:
```

```
 self.tree[node] = 1 # 颜色 1 用位掩码 1 表示
```

```
return
```

```
mid = (l + r) // 2
```

```
self._build(node * 2, l, mid)
```

```
self._build(node * 2 + 1, mid + 1, r)
```

```
self.tree[node] = self.tree[node * 2] | self.tree[node * 2 + 1]
```

```
def _push_down(self, node: int, l: int, r: int) -> None:
```

```
"""
```

下推懒惰标记

Args:

node: 当前节点索引

l: 当前节点左边界

r: 当前节点右边界

```
"""
```

```
if self.lazy[node] != 0 and l != r:
```

```
 color = self.lazy[node]
```

```
 self.lazy[node * 2] = color
```

```
 self.lazy[node * 2 + 1] = color
```

```
 self.tree[node * 2] = 1 << (color - 1)
```

```
 self.tree[node * 2 + 1] = 1 << (color - 1)
```

```
 self.lazy[node] = 0
```

```
def update(self, l: int, r: int, color: int) -> None:
```

```
"""
```

区间染色操作

Args:

l: 区间左端点

r: 区间右端点

color: 颜色编号(1-30)

Raises:

ValueError: 输入参数不合法

```
"""
```

# 输入验证

```
if l < 1 or r > self.n or l > r or color < 1 or color > 30:
```

```
 raise ValueError(f"Invalid parameters: l={l}, r={r}, color={color}")
```

```
self._update(1, 1, self.n, l, r, color)
```

```
def _update(self, node: int, l: int, r: int, ql: int, qr: int, color: int) -> None:
```

```
"""
```

区间染色操作（内部实现）

Args:

node: 当前节点索引

l: 当前节点左边界

r: 当前节点右边界

ql: 查询区间左边界

qr: 查询区间右边界

color: 颜色编号

```
"""
```

```
if ql <= l and r <= qr:
```

```
 # 完全覆盖，设置懒惰标记和当前节点颜色
```

```
 self.lazy[node] = color
```

```
 self.tree[node] = 1 << (color - 1)
```

```
 return
```

```
下推懒惰标记
```

```
self._push_down(node, l, r)
```

```
mid = (l + r) // 2
```

```
if ql <= mid:
```

```
 self._update(node * 2, l, mid, ql, qr, color)
```

```
if qr > mid:
```

```
 self._update(node * 2 + 1, mid + 1, r, ql, qr, color)
```

```
合并子区间信息
```

```
self.tree[node] = self.tree[node * 2] | self.tree[node * 2 + 1]
```

```
def query(self, l: int, r: int) -> int:
```

```
"""
```

查询区间颜色种类数

Args:

l: 区间左端点

r: 区间右端点

Returns:

int: 颜色种类数

Raises:

ValueError: 输入参数不合法

```
"""
```

```

输入验证
if l < 1 or r > self.n or l > r:
 raise ValueError(f"Invalid query parameters: l={l}, r={r}")

mask = self._query(1, 1, self.n, l, r)
return bin(mask).count('1') # 计算位掩码中 1 的个数

def _query(self, node: int, l: int, r: int, ql: int, qr: int) -> int:
 """
 查询区间颜色集合（内部实现）

 Args:
 node: 当前节点索引
 l: 当前节点左边界
 r: 当前节点右边界
 ql: 查询区间左边界
 qr: 查询区间右边界

 Returns:
 int: 颜色集合的位掩码
 """
 if ql <= l and r <= qr:
 return self.tree[node]

 # 下推懒惰标记
 self._push_down(node, l, r)

 mid = (l + r) // 2
 result = 0
 if ql <= mid:
 result |= self._query(node * 2, l, mid, ql, qr)
 if qr > mid:
 result |= self._query(node * 2 + 1, mid + 1, r, ql, qr)
 return result

def print_state(self) -> None:
 """
 打印当前线段树状态（用于调试）
 """
 print("Tree state (first 10 elements): ", end="")
 for i in range(1, min(11, self.n + 1)):
 mask = self._query(1, 1, self.n, i, i)
 color = 0

```

```

 temp = mask
 while temp:
 color += 1
 temp >>= 1
 print(f"{color}", end=" ")
 print()

```

```
def test_code12_count_color():
```

```
 """
```

```
 测试函数
```

```
 """
```

```
 # 测试用例 1: 基本功能测试
```

```
 print("=== 测试用例 1: 基本功能测试 ===")
```

```
 seg_tree = Code12_CountColor(10)
```

```
 # 初始状态: 所有位置都是颜色 1
```

```
 print(f"初始查询[1,10]颜色种类数: {seg_tree.query(1, 10)}") # 应为 1
```

```
 # 染色操作
```

```
 seg_tree.update(1, 5, 2) # 将[1,5]染成颜色 2
```

```
 print(f"染色后查询[1,10]颜色种类数: {seg_tree.query(1, 10)}") # 应为 2
```

```
 print(f"查询[1,5]颜色种类数: {seg_tree.query(1, 5)}") # 应为 1
```

```
 print(f"查询[6,10]颜色种类数: {seg_tree.query(6, 10)}") # 应为 1
```

```
 # 覆盖染色
```

```
 seg_tree.update(3, 7, 3) # 将[3,7]染成颜色 3
```

```
 print(f"覆盖染色后查询[1,10]颜色种类数: {seg_tree.query(1, 10)}") # 应为 3
```

```
 # 测试用例 2: 边界情况
```

```
 print("\n=== 测试用例 2: 边界情况 ===")
```

```
 seg_tree2 = Code12_CountColor(5)
```

```
 # 单点染色
```

```
 seg_tree2.update(1, 1, 2)
```

```
 seg_tree2.update(5, 5, 3)
```

```
 print(f"单点染色后查询[1,5]颜色种类数: {seg_tree2.query(1, 5)}") # 应为 3
```

```
 # 测试用例 3: 性能测试 (大规模数据)
```

```
 print("\n=== 测试用例 3: 性能测试 ===")
```

```
 L = 100000
```

```
 large_seg_tree = Code12_CountColor(L)
```

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

```

for i in range(1, 1001):
 l = (i * 10) % L + 1
 r = min(l + 100, L)
 large_seg_tree.update(l, r, (i % 30) + 1)
end_time = time.time()
print(f"1000 次染色操作耗时: {(end_time - start_time) * 1000:.2f}ms")

测试用例 4: 错误输入处理
print("\n=== 测试用例 4: 错误输入处理 ===")
error_seg_tree = Code12_CountColor(10)

测试无效输入
try:
 error_seg_tree.update(0, 5, 2) # 无效左边界
except ValueError as e:
 print(f"捕获到预期错误: {e}")

try:
 error_seg_tree.update(1, 15, 2) # 无效右边界
except ValueError as e:
 print(f"捕获到预期错误: {e}")

try:
 error_seg_tree.update(5, 3, 2) # 左边界大于右边界
except ValueError as e:
 print(f"捕获到预期错误: {e}")

try:
 error_seg_tree.update(1, 5, 0) # 无效颜色
except ValueError as e:
 print(f"捕获到预期错误: {e}")

try:
 error_seg_tree.update(1, 5, 31) # 无效颜色
except ValueError as e:
 print(f"捕获到预期错误: {e}")

print("所有测试用例通过!")

性能优化技巧说明
class Code12_CountColorOptimized(Code12_CountColor):
 """
 优化版本的区间染色线段树

```

优化点:

1. 使用局部变量减少属性访问
2. 避免重复计算
3. 使用位运算优化

"""

```
def _update_optimized(self, node: int, l: int, r: int, ql: int, qr: int, color: int) -> None:
 """
```

优化版本的区间更新

"""

```
 tree = self.tree
```

```
 lazy = self.lazy
```

```
 if ql <= l and r <= qr:
```

```
 lazy[node] = color
```

```
 tree[node] = 1 << (color - 1)
```

```
 return
```

```
 # 使用局部变量优化
```

```
 mid = (l + r) // 2
```

```
 left_node = node * 2
```

```
 right_node = node * 2 + 1
```

```
 # 下推懒惰标记
```

```
 if lazy[node] != 0 and l != r:
```

```
 temp_color = lazy[node]
```

```
 lazy[left_node] = temp_color
```

```
 lazy[right_node] = temp_color
```

```
 tree[left_node] = 1 << (temp_color - 1)
```

```
 tree[right_node] = 1 << (temp_color - 1)
```

```
 lazy[node] = 0
```

```
 if ql <= mid:
```

```
 self._update_optimized(left_node, l, mid, ql, qr, color)
```

```
 if qr > mid:
```

```
 self._update_optimized(right_node, mid + 1, r, ql, qr, color)
```

```
 tree[node] = tree[left_node] | tree[right_node]
```

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

```
 test_code12_count_color()
```

```
=====
文件: Code13_MaximumSubarraySum.java
=====
```

```
import java.util.*;
```

```
/**
 * SPOJ GSS1 - Can you answer these queries I - 区间最大子段和问题
 *
 * 题目描述:
 * 给定一个长度为 n 的整数序列, 执行 m 次查询操作, 每次查询[l, r]区间内的最大子段和。
 *
 * 解法要点:
 * - 使用线段树维护区间信息
 * - 每个节点存储: 区间最大子段和、左最大子段和、右最大子段和、区间总和
 * - 通过合并子区间信息得到父区间信息
 *
 * 时间复杂度:
 * - 构建线段树: $O(n)$
 * - 每次查询: $O(\log n)$
 *
 * 空间复杂度: $O(4n)$
 *
 * 工程化考量:
 * - 信息合并的边界处理
 * - 负数处理
 * - 空区间处理
 * - 性能优化
 */
```

```
public class Code13_MaximumSubarraySum {

 // 线段树节点信息
 static class Node {
 int sum; // 区间总和
 int maxSum; // 区间最大子段和
 int leftMax; // 左最大子段和 (从区间左端点开始)
 int rightMax; // 右最大子段和 (到区间右端点结束)

 Node() {}

 Node(int val) {
 this.sum = val;
 this.maxSum = val;
 }
 }
}
```

```

 this.leftMax = val;
 this.rightMax = val;
 }

```

// 合并两个节点信息

```

static Node merge(Node left, Node right) {
 if (left == null) return right;
 if (right == null) return left;

 Node res = new Node();
 res.sum = left.sum + right.sum;
 res.leftMax = Math.max(left.leftMax, left.sum + right.leftMax);
 res.rightMax = Math.max(right.rightMax, right.sum + left.rightMax);
 res.maxSum = Math.max(Math.max(left.maxSum, right.maxSum),
 left.rightMax + right.leftMax);

 return res;
}

```

```

private Node[] tree;
private int[] arr;
private int n;

```

```

/**
 * 构造函数
 * @param arr 输入数组
 */
public Code13_MaximumSubarraySum(int[] arr) {
 this.arr = arr.clone();
 this.n = arr.length;
 this.tree = new Node[4 * n];
 build(1, 0, n - 1);
}

```

```

/**
 * 构建线段树
 */
private void build(int node, int l, int r) {
 if (l == r) {
 tree[node] = new Node(arr[l]);
 return;
 }

```

```

 int mid = (l + r) >> 1;
 build(node << 1, l, mid);
 build(node << 1 | 1, mid + 1, r);
 tree[node] = Node.merge(tree[node << 1], tree[node << 1 | 1]);
 }

```

```

/**

```

```

 * 查询区间最大子段和

```

```

 * @param l 区间左端点

```

```

 * @param r 区间右端点

```

```

 * @return 最大子段和

```

```

 */

```

```

public int query(int l, int r) {
 Node res = query(1, 0, n - 1, l, r);
 return res.maxSum;
}

```

```

private Node query(int node, int l, int r, int ql, int qr) {
 if (ql <= l && r <= qr) {
 return tree[node];
 }

```

```

 int mid = (l + r) >> 1;

```

```

 Node leftRes = null, rightRes = null;

```

```

 if (ql <= mid) {

```

```

 leftRes = query(node << 1, l, mid, ql, qr);
 }

```

```

 if (qr > mid) {

```

```

 rightRes = query(node << 1 | 1, mid + 1, r, ql, qr);
 }

```

```

 if (leftRes == null) return rightRes;

```

```

 if (rightRes == null) return leftRes;

```

```

 return Node.merge(leftRes, rightRes);
}

```

```

/**

```

```

 * 单点更新（可选功能）

```

```

 * @param index 索引

```

```

 * @param value 新值

```

```

 */

```

```

public void update(int index, int value) {

```

```

 update(l, 0, n - 1, index, value);
 }

private void update(int node, int l, int r, int index, int value) {
 if (l == r) {
 arr[l] = value;
 tree[node] = new Node(value);
 return;
 }

 int mid = (l + r) >> 1;
 if (index <= mid) {
 update(node << 1, l, mid, index, value);
 } else {
 update(node << 1 | 1, mid + 1, r, index, value);
 }

 tree[node] = Node.merge(tree[node << 1], tree[node << 1 | 1]);
}

/**
 * 测试方法
 */
public static void main(String[] args) {
 // 测试用例 1: 基本功能测试
 System.out.println("=== 测试用例 1: 基本功能测试 ===");
 int[] arr1 = {1, 2, 3, 4, 5};
 Code13_MaximumSubarraySum segTree1 = new Code13_MaximumSubarraySum(arr1);

 System.out.println("数组: " + Arrays.toString(arr1));
 System.out.println("查询[0,4]最大子段和: " + segTree1.query(0, 4)); // 应为 15
 System.out.println("查询[0,2]最大子段和: " + segTree1.query(0, 2)); // 应为 6
 System.out.println("查询[2,4]最大子段和: " + segTree1.query(2, 4)); // 应为 12

 // 测试用例 2: 包含负数的情况
 System.out.println("\n=== 测试用例 2: 包含负数的情况 ===");
 int[] arr2 = {-1, 2, 3, -4, 5, -6};
 Code13_MaximumSubarraySum segTree2 = new Code13_MaximumSubarraySum(arr2);

 System.out.println("数组: " + Arrays.toString(arr2));
 System.out.println("查询[0,5]最大子段和: " + segTree2.query(0, 5)); // 应为 6 (2+3-4+5)
 System.out.println("查询[1,4]最大子段和: " + segTree2.query(1, 4)); // 应为 6 (2+3-4+5)
 System.out.println("查询[0,3]最大子段和: " + segTree2.query(0, 3)); // 应为 5 (2+3)
}

```

// 测试用例 3: 全负数情况

```
System.out.println("\n=== 测试用例 3: 全负数情况 ===");
```

```
int[] arr3 = {-5, -3, -2, -7, -1};
```

```
Code13_MaximumSubarraySum segTree3 = new Code13_MaximumSubarraySum(arr3);
```

```
System.out.println("数组: " + Arrays.toString(arr3));
```

```
System.out.println("查询[0,4]最大子段和: " + segTree3.query(0, 4)); // 应为-1
```

```
System.out.println("查询[1,3]最大子段和: " + segTree3.query(1, 3)); // 应为-2
```

// 测试用例 4: 单点更新测试

```
System.out.println("\n=== 测试用例 4: 单点更新测试 ===");
```

```
int[] arr4 = {1, -2, 3, -4, 5};
```

```
Code13_MaximumSubarraySum segTree4 = new Code13_MaximumSubarraySum(arr4);
```

```
System.out.println("更新前数组: " + Arrays.toString(arr4));
```

```
System.out.println("更新前查询[0,4]最大子段和: " + segTree4.query(0, 4)); // 应为 5
```

```
segTree4.update(1, 10); // 将-2 改为 10
```

```
System.out.println("更新后查询[0,4]最大子段和: " + segTree4.query(0, 4)); // 应为 15
```

// 测试用例 5: 边界情况

```
System.out.println("\n=== 测试用例 5: 边界情况 ===");
```

```
int[] arr5 = {10};
```

```
Code13_MaximumSubarraySum segTree5 = new Code13_MaximumSubarraySum(arr5);
```

```
System.out.println("单元数组: " + Arrays.toString(arr5));
```

```
System.out.println("查询[0,0]最大子段和: " + segTree5.query(0, 0)); // 应为 10
```

// 测试用例 6: 性能测试

```
System.out.println("\n=== 测试用例 6: 性能测试 ===");
```

```
int size = 100000;
```

```
int[] largeArr = new int[size];
```

```
Random rand = new Random();
```

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

```
 largeArr[i] = rand.nextInt(2001) - 1000; // -1000 到 1000 的随机数
```

```
}
```

```
long startTime = System.currentTimeMillis();
```

```
Code13_MaximumSubarraySum largeSegTree = new Code13_MaximumSubarraySum(largeArr);
```

```
long buildTime = System.currentTimeMillis();
```

// 执行 1000 次查询

```

 for (int i = 0; i < 1000; i++) {
 int l = rand.nextInt(size);
 int r = l + rand.nextInt(Math.min(1000, size - l));
 largeSegTree.query(l, r);
 }
 long queryTime = System.currentTimeMillis();

 System.out.println("构建" + size + "个元素的线段树耗时：" + (buildTime - startTime) +
"ms");
 System.out.println("1000 次查询耗时：" + (queryTime - buildTime) + "ms");

 System.out.println("所有测试用例通过！");
 }
}

```

=====

文件: Code14\_FallingSquares.cpp

=====

```

/**
 * LeetCode 699. Falling Squares
 *
 * 题目链接: https://leetcode.cn/problems/falling-squares/
 *
 * 题目描述:
 * 在二维平面上的 x 轴上, 放置着一些方块。给你一个二维整数数组 positions,
 * 其中 positions[i] = [lefti, sideLengthi] 表示: 第 i 个方块边长为 sideLengthi,
 * 其左侧边与 x 轴上坐标点 lefti 对齐。每个方块都从一个比目前所有的落地方块更高的高度掉落而下。
 * 方块沿 y 轴负方向下落, 直到着陆到另一个正方形的顶边或者是 x 轴上。一旦着陆, 它就会固定在原地,
 * 无法移动。
 *
 * 在每个方块掉落後, 你必须记录目前所有已经落稳的方块堆叠的最高高度。返回一个整数数组 ans,
 * 其中 ans[i] 表示在第 i 块方块掉落後堆叠的最高高度。
 *
 *
 * 解题思路:
 * 使用动态开点线段树维护区间最大值。对于每个掉落的方块:
 * 1. 查询其底部区间的最大高度
 * 2. 计算方块顶部的高度 (底部最大高度 + 方块边长)
 * 3. 更新区间为方块顶部高度
 * 4. 记录当前全局最大高度
 *
 *
 * 关键技术:
 * 1. 动态开点线段树: 处理大数据范围
 * 2. 区间更新: 将区间更新为固定值

```

- \* 3. 区间查询：查询区间最大值
- \*
- \* 时间复杂度分析：
- \* 1. 建树： $O(1)$  - 按需创建
- \* 2. 区间更新： $O(\log n)$
- \* 3. 区间查询： $O(\log n)$
- \* 4. 总体复杂度： $O(n \log n)$
- \* 5. 空间复杂度： $O(n \log n)$
- \*
- \* 是否最优解：是
- \* 动态开点线段树是处理此类区间操作问题的最优解法
- \*
- \* 工程化考量：
- \* 1. 动态内存分配：按需创建节点
- \* 2. 懒惰标记优化：延迟更新子区间
- \* 3. 边界处理：处理节点创建和查询边界情况
- \*/

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

class Code14_FallingSquares {
private:
 /**
 * 线段树节点结构
 */
 struct Node {
 int maxVal; // 当前区间的最大高度
 int lazy; // 懒惰标记，表示待下传的固定值
 Node* left; // 左子节点
 Node* right; // 右子节点

 Node() : maxVal(0), lazy(0), left(nullptr), right(nullptr) {}
 };

 Node* root; // 线段树根节点
 const int MAX_RANGE = 1000000000; // 最大坐标范围

 /**
 * 区间更新：将 [L, R] 区间内的所有值更新为 val
 */
```

```

* @param node 当前节点
* @param start 当前区间左端点
* @param end 当前区间右端点
* @param L 更新区间左端点
* @param R 更新区间右端点
* @param val 更新的值
*/
void update(Node* &node, int start, int end, int L, int R, int val) {
 if (!node) node = new Node();

 // 如果当前区间完全包含在目标区间内
 if (L <= start && end <= R) {
 node->maxVal = val;
 node->lazy = val;
 return;
 }

 // 下传懒惰标记
 pushDown(node, start, end);

 int mid = start + (end - start) / 2;
 // 更新左子区间
 if (L <= mid) {
 update(node->left, start, mid, L, R, val);
 }
 // 更新右子区间
 if (R > mid) {
 update(node->right, mid + 1, end, L, R, val);
 }

 // 上传信息
 node->maxVal = max(
 (node->left ? node->left->maxVal : 0),
 (node->right ? node->right->maxVal : 0)
);
}

/**
* 查询区间 [L, R] 内的最大值
*
* @param node 当前节点
* @param start 当前区间左端点
* @param end 当前区间右端点

```

```

* @param L 查询区间左端点
* @param R 查询区间右端点
* @return 区间最大值
*/
int query(Node* node, int start, int end, int L, int R) {
 if (!node) return 0;

 // 如果当前区间完全包含在目标区间内
 if (L <= start && end <= R) {
 return node->maxVal;
 }

 // 下传懒惰标记
 pushDown(node, start, end);

 int mid = start + (end - start) / 2;
 int maxVal = 0;

 // 查询左子区间
 if (L <= mid) {
 maxVal = max(maxVal, query(node->left, start, mid, L, R));
 }

 // 查询右子区间
 if (R > mid) {
 maxVal = max(maxVal, query(node->right, mid + 1, end, L, R));
 }

 return maxVal;
}

/**
* 下传懒惰标记
*
* @param node 当前节点
* @param start 当前区间左端点
* @param end 当前区间右端点
*/
void pushDown(Node* node, int start, int end) {
 if (node->lazy != 0) {
 // 为左子节点创建并传递标记
 if (!node->left) node->left = new Node();
 node->left->maxVal = node->lazy;
 node->left->lazy = node->lazy;
 }
}

```

```

 // 为右子节点创建并传递标记
 if (!node->right) node->right = new Node();
 node->right->maxVal = node->lazy;
 node->right->lazy = node->lazy;

 // 清除当前节点的懒惰标记
 node->lazy = 0;
 }
}

/**
 * 释放节点内存
 *
 * @param node 要释放的节点
 */
void freeTree(Node* node) {
 if (node) {
 freeTree(node->left);
 freeTree(node->right);
 delete node;
 }
}

public:
 /**
 * 构造函数
 */
 Code14_FallingSquares() {
 root = new Node();
 }

 /**
 * 析构函数
 */
 ~Code14_FallingSquares() {
 freeTree(root);
 }

 /**
 * 处理掉落的方块，返回每次掉落后的最大高度列表
 *
 * @param positions 方块的位置列表，每个元素为 [left, side_length]

```

```

 * @return 每次掉落后的最大高度列表
 */
vector<int> fallingSquares(vector<vector<int>>& positions) {
 vector<int> result;
 int maxHeight = 0;

 for (const auto& pos : positions) {
 int left = pos[0];
 int sideLength = pos[1];
 int right = left + sideLength - 1;

 // 查询当前区间的最大高度
 int currentMax = query(root, 0, MAX_RANGE, left, right);

 // 计算新方块的顶部高度
 int newHeight = currentMax + sideLength;

 // 更新区间高度
 update(root, 0, MAX_RANGE, left, right, newHeight);

 // 更新全局最大高度
 maxHeight = max(maxHeight, newHeight);
 result.push_back(maxHeight);
 }

 return result;
}

};

/**
 * 测试函数
 */
int main() {
 Code14_FallingSquares solution;

 // 测试用例 1
 vector<vector<int>> positions1 = {{1, 2}, {2, 3}, {6, 1}};
 vector<int> result1 = solution.fallingSquares(positions1);
 cout << "测试用例 1 结果: [";
 for (int i = 0; i < result1.size(); i++) {
 cout << result1[i];
 if (i < result1.size() - 1) cout << ", ";
 }
}

```

```

cout << "]" << endl; // 输出: [2, 5, 5]

// 测试用例 2
vector<vector<int>> positions2 = {{100, 100}, {200, 100}};
vector<int> result2 = solution.fallingSquares(positions2);
cout << "测试用例 2 结果: [";
for (int i = 0; i < result2.size(); i++) {
 cout << result2[i];
 if (i < result2.size() - 1) cout << ", ";
}
cout << "]" << endl; // 输出: [100, 100]

return 0;
}

```

=====

文件: Code14\_FallingSquares.java

=====

```

package class114;

/**
 * LeetCode 699. Falling Squares
 *
 * 题目链接: https://leetcode.cn/problems/falling-squares/
 *
 * 题目描述:
 * 在二维平面上的 x 轴上, 放置着一些方块。给你一个二维整数数组 positions,
 * 其中 positions[i] = [lefti, sideLengthi] 表示: 第 i 个方块边长为 sideLengthi,
 * 其左侧边与 x 轴上坐标点 lefti 对齐。每个方块都从一个比目前所有的落地方块更高的高度掉落而下。
 * 方块沿 y 轴负方向下落, 直到着陆到另一个正方形的顶边或者是 x 轴上。一旦着陆, 它就会固定在原地,
 * 无法移动。
 * 在每个方块掉落後, 你必须记录目前所有已经落稳的方块堆叠的最高高度。返回一个整数数组 ans,
 * 其中 ans[i] 表示在第 i 块方块掉落後堆叠的最高高度。
 *
 * 解题思路:
 * 使用动态开点线段树维护区间最大值。对于每个掉落的方块:
 * 1. 查询其底部区间的最大高度
 * 2. 计算方块顶部的高度 (底部最大高度 + 方块边长)
 * 3. 更新区间为方块顶部高度
 * 4. 记录当前全局最大高度
 *
 * 关键技术:

```

- \* 1. 动态开点线段树：处理大数据范围
- \* 2. 区间更新：将区间更新为固定值
- \* 3. 区间查询：查询区间最大值
- \*
- \* 时间复杂度分析：
- \* 1. 建树： $O(1)$  - 按需创建
- \* 2. 区间更新： $O(\log n)$
- \* 3. 区间查询： $O(\log n)$
- \* 4. 总体复杂度： $O(n \log n)$
- \* 5. 空间复杂度： $O(n \log n)$
- \*
- \* 是否最优解：是
- \* 动态开点线段树是处理此类区间操作问题的最优解法
- \*
- \* 工程化考量：
- \* 1. 动态内存分配：按需创建节点
- \* 2. 懒惰标记优化：延迟更新子区间
- \* 3. 边界处理：处理节点创建和查询边界情况
- \*/

```
import java.util.*;
```

```
public class Code14_FallingSquares {

 /**
 * 线段树节点类
 */
 static class Node {
 int max; // 当前区间的最大高度
 int lazy; // 懒惰标记，表示待下传的固定值
 Node left; // 左子节点
 Node right; // 右子节点
 }

 private Node root; // 线段树根节点
 private final int MAX_RANGE = 1000000000; // 最大坐标范围

 /**
 * 构造函数
 */
 public Code14_FallingSquares() {
 root = new Node();
 }
}
```

```

/**
 * 处理掉落的方块，返回每次掉落后的最大高度列表
 *
 * @param positions 方块的位置列表，每个元素为 [left, side_length]
 * @return 每次掉落后的最大高度列表
 */
public List<Integer> fallingSquares(int[][] positions) {
 List<Integer> result = new ArrayList<>();
 int maxHeight = 0;

 for (int[] pos : positions) {
 int left = pos[0];
 int sideLength = pos[1];
 int right = left + sideLength - 1;

 // 查询当前区间的最大高度
 int currentMax = query(root, 0, MAX_RANGE, left, right);

 // 计算新方块的顶部高度
 int newHeight = currentMax + sideLength;

 // 更新区间高度
 update(root, 0, MAX_RANGE, left, right, newHeight);

 // 更新全局最大高度
 maxHeight = Math.max(maxHeight, newHeight);
 result.add(maxHeight);
 }

 return result;
}

/**
 * 区间更新：将 [L, R] 区间内的所有值更新为 val
 *
 * @param node 当前节点
 * @param start 当前区间左端点
 * @param end 当前区间右端点
 * @param L 更新区间左端点
 * @param R 更新区间右端点
 * @param val 更新的值
 */

```

```

private void update(Node node, int start, int end, int L, int R, int val) {
 // 如果当前区间完全包含在目标区间内
 if (L <= start && end <= R) {
 node.max = val;
 node.lazy = val;
 return;
 }

 // 下传懒惰标记
 pushDown(node, start, end);

 int mid = start + (end - start) / 2;
 // 更新左子区间
 if (L <= mid) {
 if (node.left == null) node.left = new Node();
 update(node.left, start, mid, L, R, val);
 }
 // 更新右子区间
 if (R > mid) {
 if (node.right == null) node.right = new Node();
 update(node.right, mid + 1, end, L, R, val);
 }

 // 上传信息
 node.max = Math.max(
 (node.left != null ? node.left.max : 0),
 (node.right != null ? node.right.max : 0)
);
}

```

```

/**

```

```

 * 查询区间 [L, R] 内的最大值

```

```

 *

```

```

 * @param node 当前节点

```

```

 * @param start 当前区间左端点

```

```

 * @param end 当前区间右端点

```

```

 * @param L 查询区间左端点

```

```

 * @param R 查询区间右端点

```

```

 * @return 区间最大值

```

```

 */

```

```

private int query(Node node, int start, int end, int L, int R) {
 // 如果当前节点为空, 返回 0
 if (node == null) return 0;

```

```

// 如果当前区间完全包含在目标区间内
if (L <= start && end <= R) {
 return node.max;
}

// 下传懒惰标记
pushDown(node, start, end);

int mid = start + (end - start) / 2;
int maxVal = 0;

// 查询左子区间
if (L <= mid) {
 maxVal = Math.max(maxVal, query(node.left, start, mid, L, R));
}
// 查询右子区间
if (R > mid) {
 maxVal = Math.max(maxVal, query(node.right, mid + 1, end, L, R));
}

return maxVal;
}

/**
 * 下传懒惰标记
 *
 * @param node 当前节点
 * @param start 当前区间左端点
 * @param end 当前区间右端点
 */
private void pushDown(Node node, int start, int end) {
 if (node.lazy != 0) {
 // 为左子节点创建并传递标记
 if (node.left == null) node.left = new Node();
 node.left.max = node.lazy;
 node.left.lazy = node.lazy;

 // 为右子节点创建并传递标记
 if (node.right == null) node.right = new Node();
 node.right.max = node.lazy;
 node.right.lazy = node.lazy;
 }
}

```

```

 // 清除当前节点的懒惰标记
 node.lazy = 0;
 }
}

/**
 * 主方法，用于测试
 */
public static void main(String[] args) {
 Code14_FallingSquares solution = new Code14_FallingSquares();

 // 测试用例 1
 int[][] positions1 = {{1, 2}, {2, 3}, {6, 1}};
 System.out.println("测试用例 1 结果: " + solution.fallingSquares(positions1)); // 输出:
 [2, 5, 5]

 // 测试用例 2
 int[][] positions2 = {{100, 100}, {200, 100}};
 System.out.println("测试用例 2 结果: " + solution.fallingSquares(positions2)); // 输出:
 [100, 100]
}
}

```

文件: Code14\_FallingSquares.py

```

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

```

"""

LeetCode 699. Falling Squares

题目链接: <https://leetcode.cn/problems/falling-squares/>

题目描述:

在二维平面上的  $x$  轴上，放置着一些方块。给你一个二维整数数组 `positions`，其中 `positions[i] = [lefti, sideLengthi]` 表示：第  $i$  个方块边长为 `sideLengthi`，其左侧边与  $x$  轴上坐标点 `lefti` 对齐。每个方块都从一个比目前所有的落地方块更高的高度掉落而下。方块沿  $y$  轴负方向下落，直到着陆到另一个正方形的顶边或者是  $x$  轴上。一旦着陆，它就会固定在原地，无法移动。

在每个方块掉落後，你必须记录目前所有已经落稳的方块堆叠的最高高度。返回一个整数数组 `ans`，其中 `ans[i]` 表示在第  $i$  块方块掉落後堆叠的最高高度。

解题思路：

使用动态开点线段树维护区间最大值。对于每个掉落的方块：

1. 查询其底部区间的最大高度
2. 计算方块顶部的高度（底部最大高度 + 方块边长）
3. 更新区间为方块顶部高度
4. 记录当前全局最大高度

关键技术：

1. 动态开点线段树：处理大数据范围
2. 区间更新：将区间更新为固定值
3. 区间查询：查询区间最大值

时间复杂度分析：

1. 建树： $O(1)$  - 按需创建
2. 区间更新： $O(\log n)$
3. 区间查询： $O(\log n)$
4. 总体复杂度： $O(n \log n)$
5. 空间复杂度： $O(n \log n)$

是否最优解：是

动态开点线段树是处理此类区间操作问题的最优解法

工程化考量：

1. 动态内存分配：按需创建节点
2. 懒惰标记优化：延迟更新子区间
3. 边界处理：处理节点创建和查询边界情况

"""

```
class Code14_FallingSquares:
```

```
 """
```

```
 掉落的方块解决方案类
```

```
 """
```

```
 def __init__(self):
```

```
 """
```

```
 初始化线段树
```

```
 """
```

```
 # 使用字典动态存储节点，键为节点索引，值为节点信息
```

```
 self.tree = {0: {'max_height': 0, 'lazy': 0, 'left': 0, 'right': 0}}
```

```
 self.node_count = 1 # 当前已使用的节点数
```

```
 self.MAX_VAL = 10**9 # 最大坐标范围
```

```
def push_down(self, node, start, end):
```

```
 """
```

下推懒惰标记

Args:

node (int): 当前节点索引

start (int): 当前区间左端点

end (int): 当前区间右端点

```
 """
```

```
 if self.tree[node]['lazy'] != 0 and start < end:
```

```
 # 确保左右子节点存在
```

```
 if self.tree[node]['left'] == 0:
```

```
 self.tree[node]['left'] = self.node_count
```

```
 self.tree[self.node_count] = {'max_height': 0, 'lazy': 0, 'left': 0, 'right': 0}
```

```
 self.node_count += 1
```

```
 if self.tree[node]['right'] == 0:
```

```
 self.tree[node]['right'] = self.node_count
```

```
 self.tree[self.node_count] = {'max_height': 0, 'lazy': 0, 'left': 0, 'right': 0}
```

```
 self.node_count += 1
```

```
 left_node = self.tree[node]['left']
```

```
 right_node = self.tree[node]['right']
```

```
 # 下传标记
```

```
 self.tree[left_node]['max_height'] = self.tree[node]['lazy']
```

```
 self.tree[left_node]['lazy'] = self.tree[node]['lazy']
```

```
 self.tree[right_node]['max_height'] = self.tree[node]['lazy']
```

```
 self.tree[right_node]['lazy'] = self.tree[node]['lazy']
```

```
 # 清除当前节点标记
```

```
 self.tree[node]['lazy'] = 0
```

```
def update(self, node, start, end, l, r, val):
```

```
 """
```

区间更新：将 [l, r] 区间内的所有值更新为 val

Args:

node (int): 当前节点索引

start (int): 当前区间左端点

end (int): 当前区间右端点

l (int): 更新区间左端点

r (int): 更新区间右端点

val (int): 更新的值

```

"""
if start > r or end < l:
 return

if l <= start and end <= r:
 self.tree[node]['max_height'] = val
 self.tree[node]['lazy'] = val
 return

self.push_down(node, start, end)
mid = start + (end - start) // 2

left_node = self.tree[node]['left']
right_node = self.tree[node]['right']

self.update(left_node, start, mid, l, r, val)
self.update(right_node, mid + 1, end, l, r, val)

更新当前节点的最大值
self.tree[node]['max_height'] = max(
 self.tree[left_node]['max_height'] if left_node != 0 else 0,
 self.tree[right_node]['max_height'] if right_node != 0 else 0
)

def query(self, node, start, end, l, r):
 """
 查询区间 [l, r] 内的最大值

 Args:
 node (int): 当前节点索引
 start (int): 当前区间左端点
 end (int): 当前区间右端点
 l (int): 查询区间左端点
 r (int): 查询区间右端点

 Returns:
 int: 区间最大值
 """
 if start > r or end < l:
 return 0 # 不在查询范围内

 if l <= start and end <= r:
 return self.tree[node]['max_height']

```

```

self.push_down(node, start, end)
mid = start + (end - start) // 2

left_node = self.tree[node]['left']
right_node = self.tree[node]['right']

max_val = 0
if left_node != 0:
 max_val = max(max_val, self.query(left_node, start, mid, l, r))
if right_node != 0:
 max_val = max(max_val, self.query(right_node, mid + 1, end, l, r))

return max_val

```

```
def fallingSquares(self, positions):
```

```
 """
```

处理掉落的方块，返回每次掉落后的最大高度列表

Args:

positions (List[List[int]]): 方块的位置列表，每个元素为 [left, side\_length]

Returns:

List[int]: 每次掉落后的最大高度列表

```
 """
```

```
 result = []
```

```
 max_height = 0
```

```
 for pos in positions:
```

```
 left = pos[0]
```

```
 side_length = pos[1]
```

```
 right = left + side_length - 1
```

```
 # 查询当前区间的最大高度
```

```
 current_max = self.query(0, 0, self.MAX_VAL, left, right)
```

```
 # 计算新方块的顶部高度
```

```
 new_height = current_max + side_length
```

```
 # 更新区间高度
```

```
 self.update(0, 0, self.MAX_VAL, left, right, new_height)
```

```
 # 更新全局最大高度
```

```
 max_height = max(max_height, new_height)
 result.append(max_height)
```

```
 return result
```

```
def main():
```

```
 """
```

```
 主函数，用于测试
```

```
 """
```

```
 solution = Code14_FallingSquares()
```

```
 # 测试用例 1
```

```
 positions1 = [[1, 2], [2, 3], [6, 1]]
```

```
 result1 = solution.fallingSquares(positions1)
```

```
 print("测试用例 1 结果:", result1) # 输出: [2, 5, 5]
```

```
 # 测试用例 2
```

```
 positions2 = [[100, 100], [200, 100]]
```

```
 result2 = solution.fallingSquares(positions2)
```

```
 print("测试用例 2 结果:", result2) # 输出: [100, 100]
```

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

```
 main()
```

```
=====
```