拉勾教育
—互联网人实战大学—

线段树：形成目标数组的子数组最少增加次数

困难/线段树、数组、数学思维

学习目标

拉勾教育
—互联网人实战大学—

掌握线段树的结构

掌握线段树的应用

掌握数学思维的应用

题目描述

给你一个整数数组 target 和一个数组 initial ，initial 数组与 target 数组有同样的维度，且initial一开始全部为 0 。

请你返回从 initial 得到 target 的最少操作次数，每次操作需遵循以下规则：

1. 在 initial中选择任意子数组，并将子数组中每个元素增加 1 。
2. 答案保证在32位有符号整数以内。

题目描述

输入: target = [1,2,3,2,1]
输出: 3
解释: 我们需要至少 3 次操作从 intial 数组得到 target 数组。
[0,0,0,0,0] 将下标为 0 到 4 的元素（包含二者）加 1 。
[1,1,1,1,1] 将下标为 1 到 3 的元素（包含二者）加 1 。
[1,2,2,2,1] 将下表为 2 的元素增加 1 。
[1,2,3,2,1] 得到了目标数组。

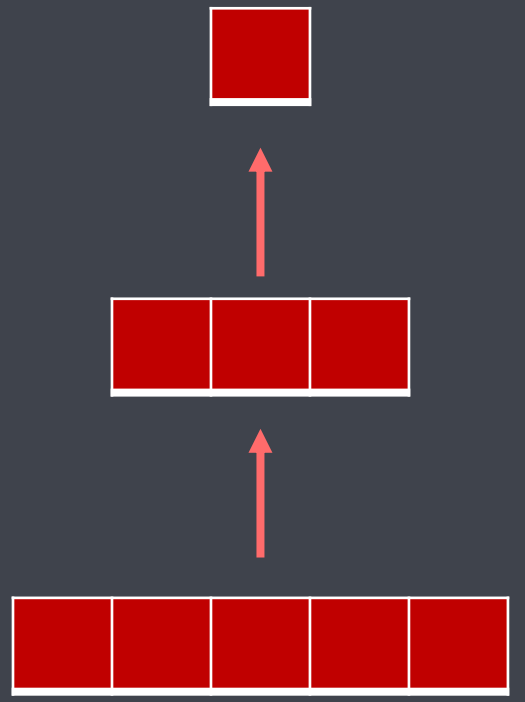
输入: target = [3,1,5,4,2]
输出: 7
解释: (initial)[0,0,0,0,0] -> [1,1,1,1,1] -> [2,1,1,1,1] -> [3,1,1,1,1]
-> [3,1,2,2,2] -> [3,1,3,3,2] ->
[3,1,4,4,2] -> [3,1,5,4,2] (target)。

输入: target = [3,1,1,2]
输出: 4
解释: (initial)[0,0,0,0] -> [1,1,1,1] -> [1,1,1,2] -> [2,1,1,2] -> [3,1,1,2]
(target) 。

一. Comprehend 理解题意

该题中的每一次操作，可以理解为，对于target数组中给定的范围，将整个范围增加最小值。类似于逐层砌墙。

如：target = [1, 2, 3, 2, 1]



第3步：下标2~2区间整体加1

第2步：下标1~3区间整体加1

第1步：下标0~4区间整体加1

二. Choose 数据结构及算法思维选择

在本题中，每次累加操作需要不断叠加，直到区间内的最小值。寻找区间最小值是“线段树”的一个基本功能。

方案一：线段树求最小值，
分治累加（基础解法）

- 数据结构：线段树，数组
- 算法思维：递归

二. Choose 数据结构及算法思维选择

关键知识点：线段树

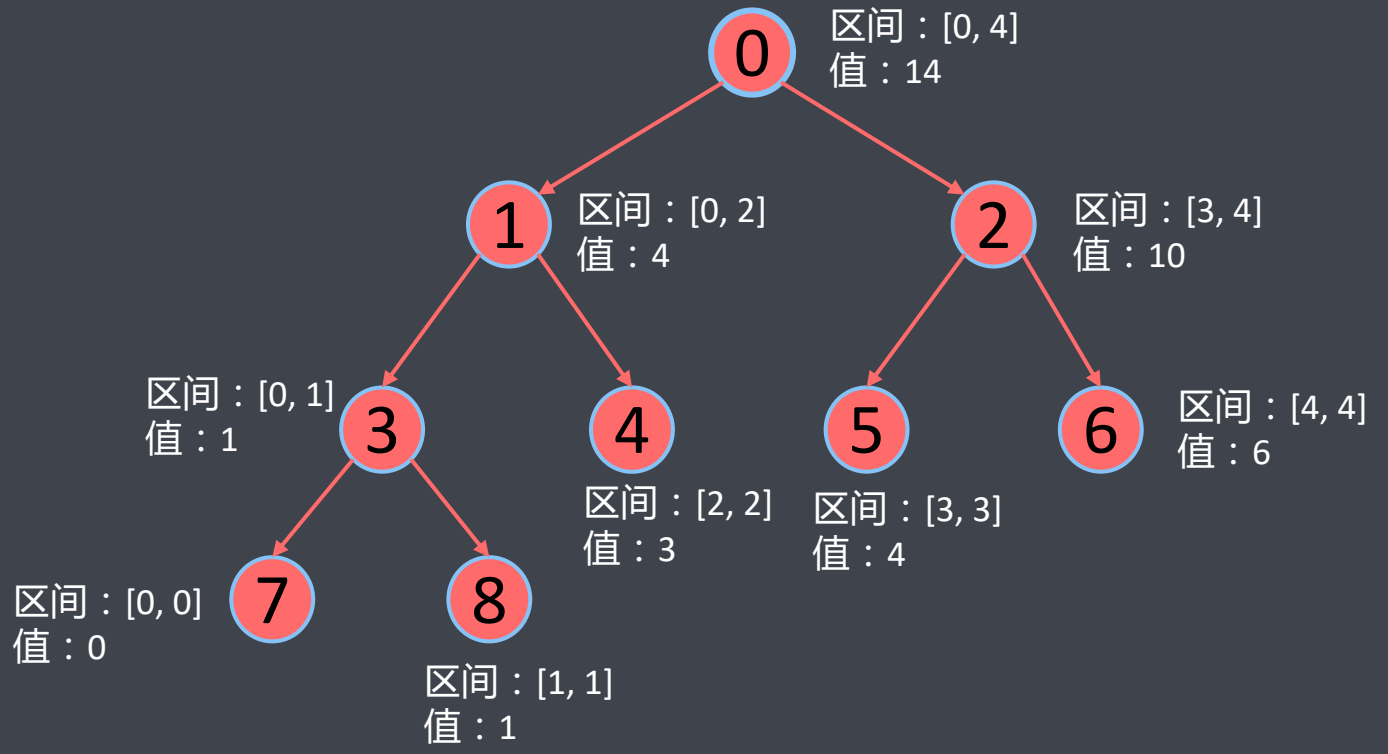
线段树是一种平衡二叉搜索树(完全二叉树)，它将一个线段区间划分成一些单元区间。对于线段树中的每一个非叶子节点，表示区间[a, b]的和，它的左孩子表示的区间为[a,(a+b)/2]，右孩子表示的区间为[(a+b)/2+1,b]，最后的叶子节点数目为N，与数组下标对应。线段树一般包括建立、查询、插入、更新等操作，建立规模为N的时间复杂度是 O(NlogN)，其他操作时间复杂度为O(logN) 。

二. Choose 数据结构及算法思维选择

由于线段树是完全二叉树，线段树可以使用数组保存，对二叉树进行层次遍历。

假设某个节点下标为i， 它的左孩子下标为2*i+1， 右孩子下标为2*i+2

如数组 nums = [0, 1, 3, 4, 6]　　存储为线段树数组values = [14, 4, 10, 1, 3, 4, 6, 0, 1]



二. Choose 数据结构及算法思维选择

线段树的构造：

处理数组：nums;

线段树数组：values

线段树数组当前位置下标：pos

处理区间：[left, right]

1、递归出口：当区间内只有一个值，

即left == right 时，将当前位置赋值为唯一的区间值，value[pos] = nums[left]

2、划分线段中心点：mid = (left + right) / 2

3、递归建立左右子树

左子树：线段树数组当前位置下标：2*pos+1

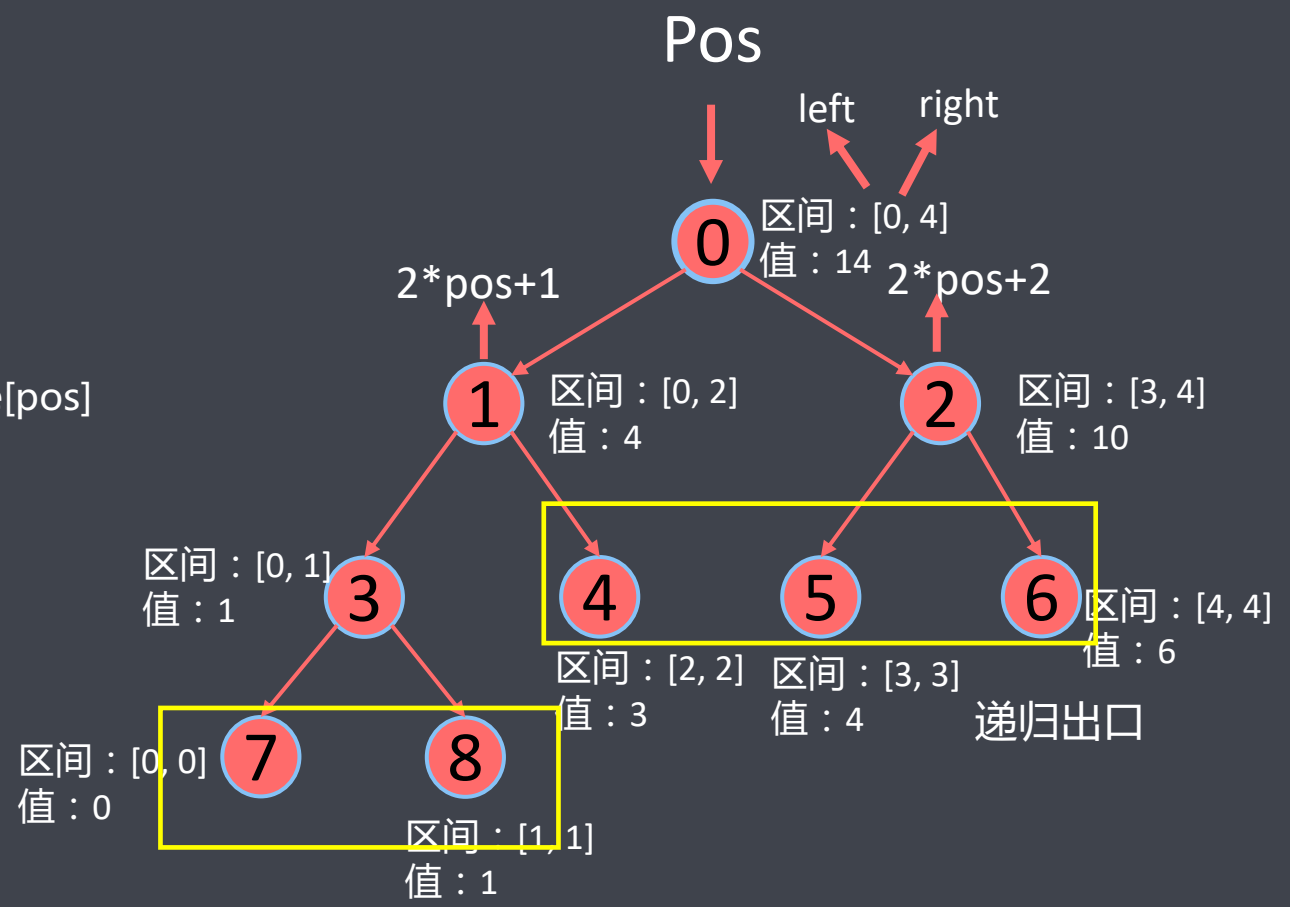
处理区间：[left, mid]

右子树：线段树数组当前位置下标：2*pos+2

处理区间：[mid + 1，right]

4、线段树数组当前位置赋值：等于左右孩子值之和：

Values[pos] = values[2*pos+1]+values[2*pos+2]



二. Choose 数据结构及算法思维选择

线段树的构造：


```
public class SegmentTree {
    int[] value, nums;
    int n;
    public SegmentTree(int n, int[] nums) {
        value = new int[4 * n];
        this.n = n;
        this.nums = nums;
        build(0, 0, n - 1);
    }

    public void build(int pos, int left, int right) {
        if (left == right) {value[pos] = nums[left];return;}
        int mid = (left + right) / 2;
        build(2 * pos + 1, left, mid);
        build(2 * pos + 2, mid + 1, right);
        value[pos] = value[2 * pos + 1] + value[2 * pos + 2];
    }
}
```

二. Choose 数据结构及算法思维选择

线段树的求和：

处理数组：nums;

线段树数组：values

线段树数组当前位置下标：pos

处理区间：[left, right]

求和区间：[qleft, qright]

1、递归出口：

- 当求和区间超出处理区间范围，返回0。
- 当求和区间和处理区间刚好重叠，返回当前线段树所存的值。

2、划分线段中心点：mid = (left + right) / 2

3、递归查找左右子树涉及部分的和：

左子树：线段树数组当前位置下标：2*pos+1

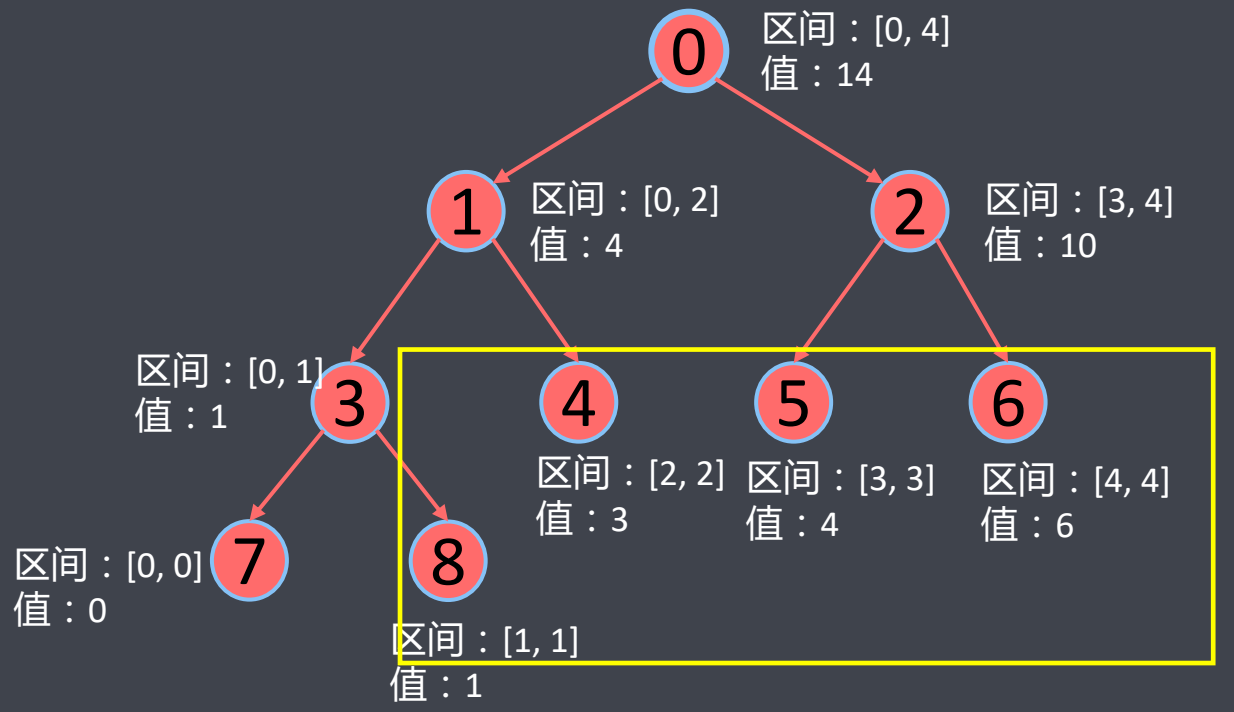
处理区间：[left, mid]

右子树：线段树数组当前位置下标：2*pos+2

处理区间：[mid + 1，right]

求和区间不变

如对于数组nums = = [0, 1, 3, 4, 6]
求区间 [1, 4]的和，为1+3+4+6 = 14



二. Choose 数据结构及算法思维选择

线段树的求和：

```
public class SegmentTree {
    int[] value, nums;
    int n;
    public SegmentTree(int n, int[] nums) {
        value = new int[4 * n];
        this.n = n;
        this.nums = nums;
        build(0, 0, n - 1);
    }
    // value[pos] 存储着nums[left,right]的和  比如query(0,0,4,1,4)
    public int query(int pos, int left, int right, int qleft, int qright) {
        if (qleft > right || qright < left) return 0;//  查询区间在当前区间之外
        if (qleft <= left && qright >= right)  return value[pos];// 查询区间完全包含当前区间
        int mid = (left + right) / 2;
        int leftSum = query_sum(2 * pos + 1, left, mid, qleft, qright);// 查询左子树
        int rightSum = query_sum(2 * pos + 2, mid + 1, right, qleft, qright);// 查询右子树
        return leftSum + rightSum;
    }
}
```

二. Choose 数据结构及算法思维选择

线段树的更新：

处理数组：nums

线段树数组：values

线段树数组当前位置下标：pos

处理区间：[left, right]

要更新的下标：index

要更新的值：new_value

1、递归出口：当更新的的为叶子节点时，对上面的结构没有影响，直接更新。

Value[pos] = new_value

2、划分线段中心点：mid = (left + right) / 2

3、在左右子树递归查找更新点

如果index <= mid, 在左子树更新。线段树数组当前位置下标：2*pos+1，处理区间：[left, mid]

否则在右子树更新值。线段树数组当前位置下标：2*pos+2，处理区间：[mid + 1，right]

如对于数组nums = [0, 1, 3, 4, 6]
线段树数组values = [14, 4, 10, 1, 3, 4, 6, 0, 1]

如果将nums[1]修改为10：
数组nums = [0, 10, 3, 4, 6]
线段树数组values = [23, 13, 10, 10, 3, 4, 6, 0, 10]

二. Choose 数据结构及算法思维选择

线段树的更新：

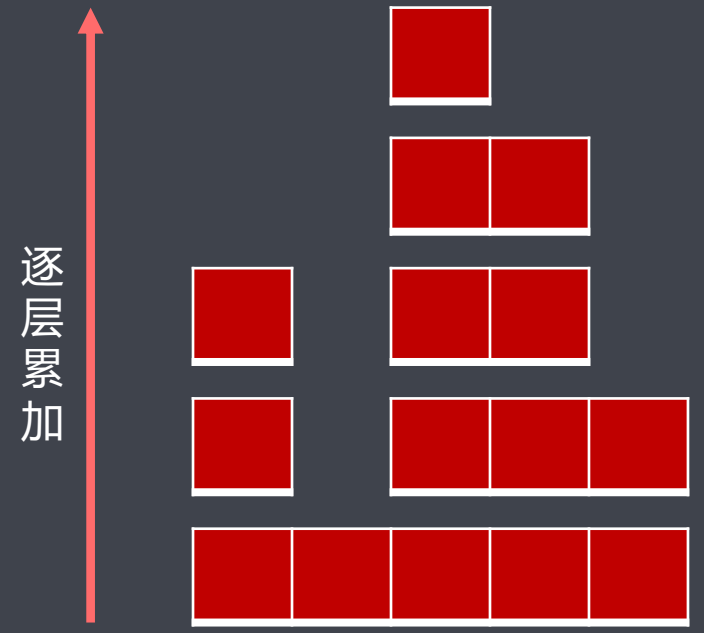
```
public class SegmentTree {
    int[] value, nums;
    int n;
    public SegmentTree(int n, int[] nums) {
        value = new int[4 * n];
        this.n = n;
        this.nums = nums;
        build(0, 0, n - 1);
    }
     // 在nums[left,right]范围内更新index的值为newValue
    public void update(int pos, int left, int right, int index, int newValue) {
        if (left == right && left == index) {value[pos] = newValue;return;}
        int mid = (left + right) / 2;
        if (index <= mid) {update(2 * pos + 1, left, mid, index, newValue);
        } else { update(2 * pos + 2, mid + 1, right, index, newValue);
        }
        value[pos] = value[pos * 2 + 1] + value[pos * 2 + 2]
    }
}
```

三. Code 基本解法及编码实现

解法一：线段树+分治

1. 分治算法就是每次找到给定区间的最小值，然后减去这个最小值，再分成左右两部分去找
2. 线段树优化查找区间最小值的过程。不需要每次减掉这个最小值更新线段树，但是需要带着这个最小值继续分治。

如例子：
Target = [3 1 5 4 2]



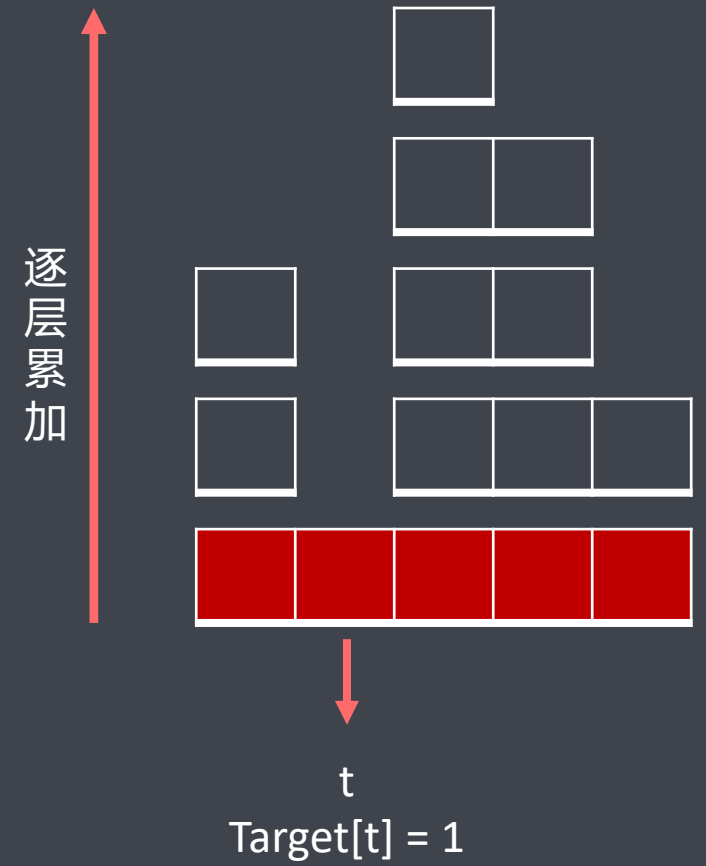
三. Code 基本解法及编码实现

Step1

对于给定区间[0, n-1]，查找最小值，并用最小值填充。记录最小值的位置t。填充次数val 累加，直到最小值target[t]

即：val = val + (target[t] - last) = 0+（1-0）=1

记录上一次填充后的基数last = 1



三. Code 基本解法及编码实现

Step2

我们将区间分为两部分，[0, t - 1] 和 [t + 1, n - 1]，然后分别找到两部分的最小值的位置。

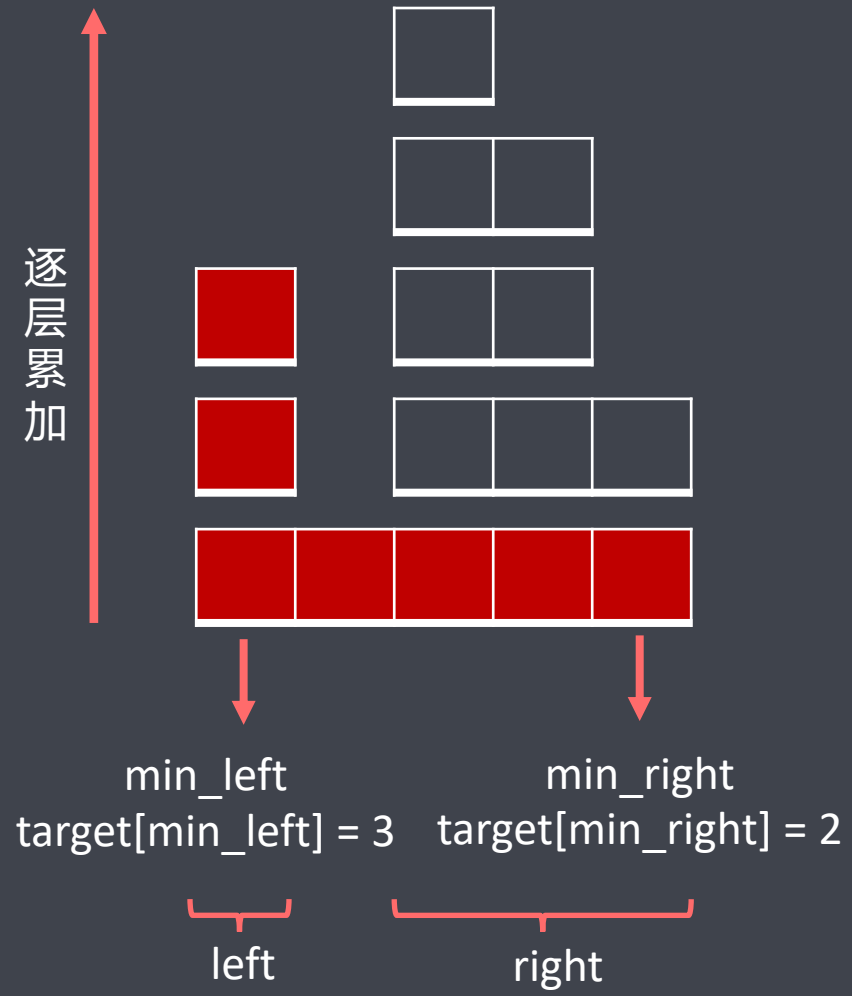
target[min_left] = 3

target[min_right] = 2

- 对于左边，填充两次，直到最小值3.

val = val + (target[t] – last) = 1+(3-1) = 3

last_left = 3



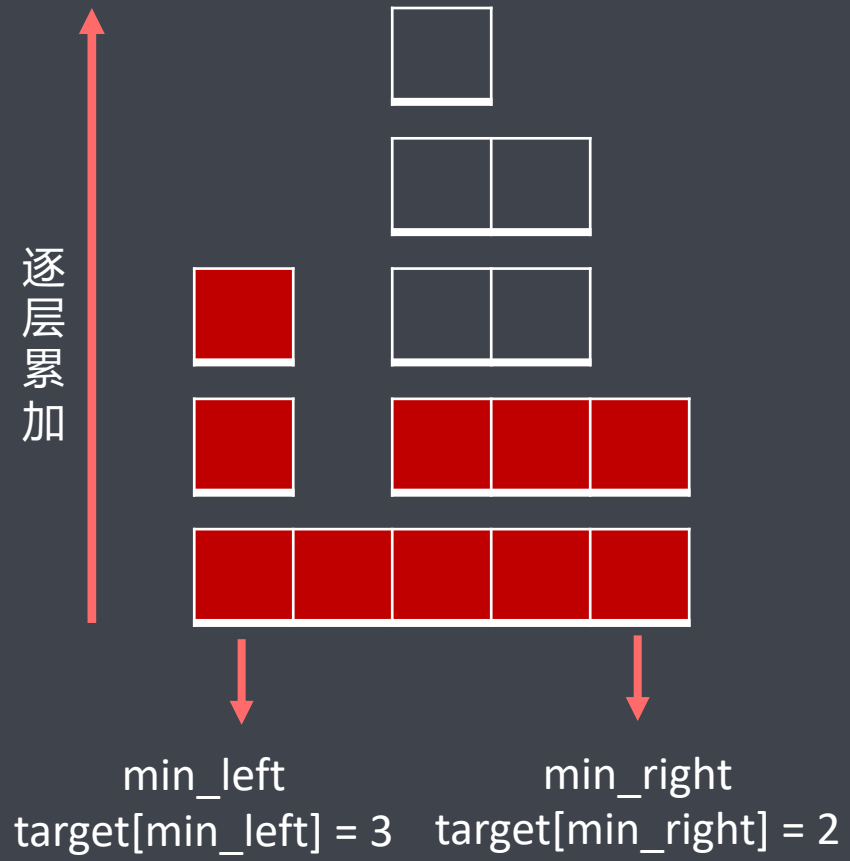
三. Code 基本解法及编码实现

Step3

- 对于右边，填充1次，直到最小值2.

val = val + (target[t] – last) = 3+(2-1) = 4

last_right = 2



三. Code 基本解法及编码实现

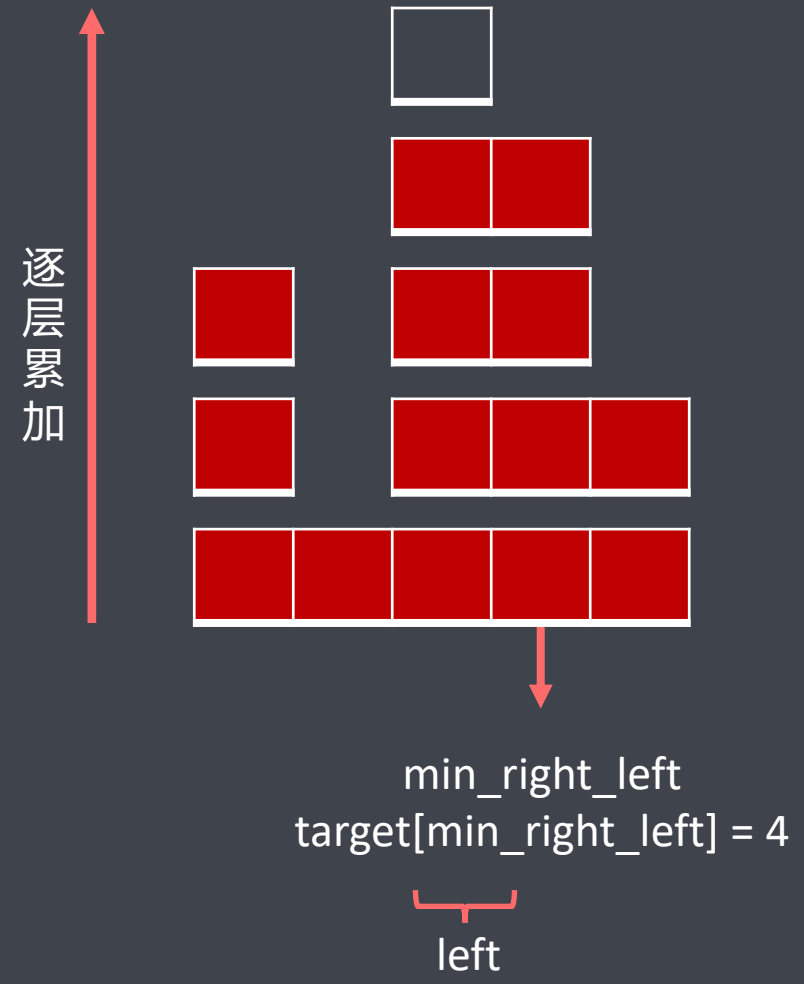
Step4

对于左右两个区间，分别以最小值为中心，递归的分成左右两个子区间，再次找到左右两个子区间的最小值。

- 对于左子区间，已经不能继续划分。
- 对于右区间，以min_right为中心，划分左右两个子区间。
- 对于左子区间，找到最小值的位置min_right_left，填充2次，至最小值target[min_right_left] =4

val = val + (target[min_right_left] – last_right) = 4 + (4 - 2) = 6

last = 4



三. Code 基本解法及编码实现

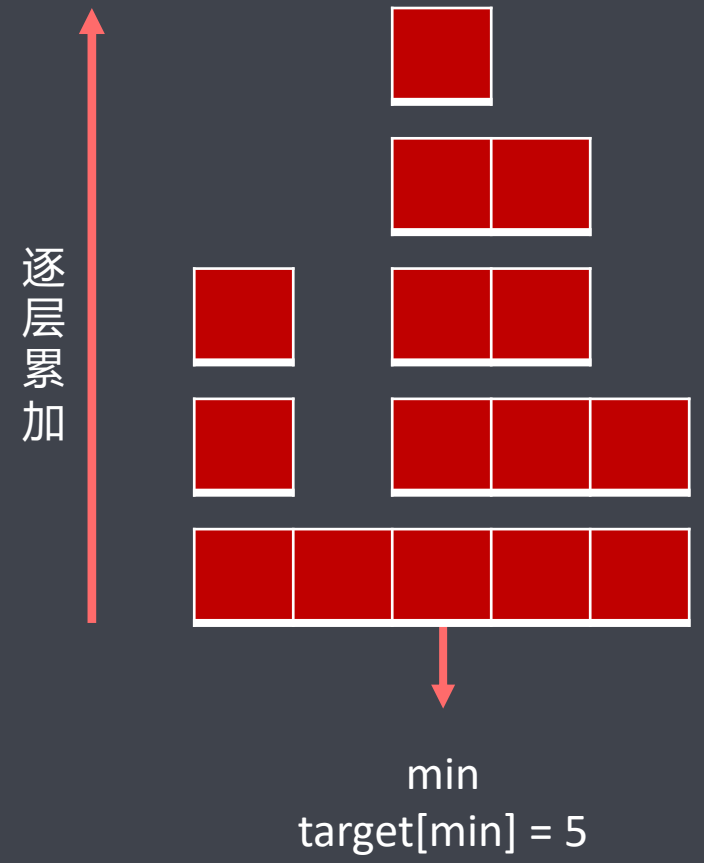
Step5

继续递归以最小值所在的位置划分左右子区间，填充至最小值的高度，记录上次填充后的基数。

val = val + (target[min_left] – last) = 6+（5 - 4）= 7

last = 5

至此递归完成，操作次数为7



三. Code 基本解法及编码实现

解法一：线段树复杂度分析

时间复杂度：O(nlogn)

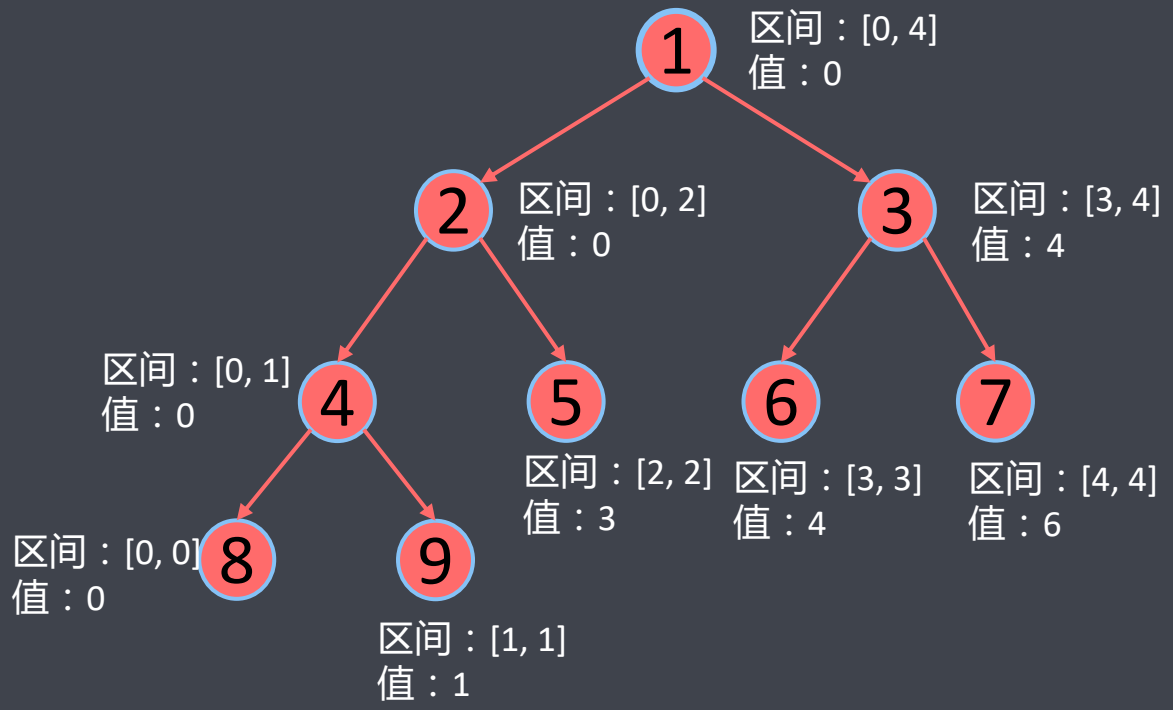
- 构建线段树O(nlogn)
- 递归求取区间最小值O(nlogn)
- O(nlogn)+O(nlogn)=O(2nlogn)
- 忽略常数后：O(nlogn)

空间复杂度：O(nlogn)

- 线段树空间O(nlogn)
- 存储最小值位置，存储上次基数，O(1)
- O(nlogn)+ O(1)
- 忽略常数后： O(nlogn)

三. Code 基本解法及编码实现

如对于数组nums = = [0, 1, 3, 4, 6] ， value=[0,0,0,4,0,3,4,6,0,1]
求区间 [1, 4]的最小值



三. Code 基本解法及编码实现

解法一：线段树参考代码

Java实现线段树


```
class SegTree { // 定义线段树，采用数组存储
        int[] value, nums;
        int n;// 表示nums数组的长度
        SegTree(int n, int[] nums) {
            value = new int[4 * n];this.n = n;this.nums = nums;
            build(1, 0, n - 1);//注意bulid pos的初值是1 后序的坐标也要跟着变
        }
        // pos表示 nums[left,right]最小值坐标在value中存储的位置
        public void build(int pos, int left, int right) {
            if (right == left) {value[pos] = left;return;}// 注意存储的是坐标
            build(2 * pos, left, (left + right) / 2);// 建立左子树
            build(2 * pos + 1, (right + left) / 2 + 1, right);// 建立右子树
            int leftIndex=value[2 * pos],rightIndex=value[2 * pos + 1];
            value[pos] = nums[leftIndex] > nums[rightIndex] ? rightIndex : leftIndex;
        }
}
```

三. Code 基本解法及编码实现

解法一：线段树参考代码

Java实现线段树


```
class SegTree { // 定义线段树，采用数组存储
        int[] value, nums;
        int n;// 表示nums数组的长度
        SegTree(int n, int[] nums) {
            value = new int[4 * n];this.n = n;this.nums = nums;
            build(1, 0, n - 1);//注意bulid pos的初值是1 后序的坐标也要跟着变
        }
        // pos 为nums[left,right]最小值的坐标在value中存储的位置
        int query(int pos, int qleft, int qright, int left, int right) {// 递归的找到给定区间的最小值
            if (left > right) return -1;
            if (qright == right && qleft == left) return value[pos];// 递归出口
            int mid = (qright + qleft) / 2;
            int leftIndex = query(2 * pos, qleft, mid, left, Math.min(mid, right));//左侧区间最小值的坐标
            int rightIndex = query(2 * pos + 1, mid + 1, qright, Math.max(left, mid + 1), right);
            if (leftIndex == -1) return rightIndex;
            if (rightIndex == -1) return leftIndex;
            return nums[leftIndex] > nums[rightIndex] ? rightIndex : leftIndex;
        }
}
```

三. Code 基本解法及编码实现

解法一：线段树参考代码

Java实现线段树


```
class SegTree { // 定义线段树，采用数组存储
        int[] value, nums;
        int n;// 表示nums数组的长度
        SegTree(int n, int[] nums) {
            value = new int[4 * n];this.n = n;this.nums = nums;
            build(1, 0, n - 1);//注意bulid pos的初值是1 后序的坐标也要跟着变
        }
        int rec(int od, int l, int r) {// 分别对左右子区间递归累加直至区间最小值 本题定制函数
            if (l > r) return 0;
            int m = query(1, 0, n - 1, l, r);
            return nums[m] - od + rec(nums[m], l, m - 1) + rec(nums[m], m + 1, r);
        }
}
```

三. Code 基本解法及编码实现

拉勾教育
—互联网人实战大学—

解法一：线段树参考代码

```
class Solution {
    public int minNumberOperations(int[] t) {
        // 新建一棵线段树
       SegTree st=new SegTree(t.length,t);
       // 调用线段树求出目标值
       return st.rec(0,0,t.length-1);
    }
}
```

执行结果： 通过 显示详情 >

执行用时： 215 ms , 在所有 Java 提交中击败了 5.18% 的用户

内存消耗： 55.4 MB , 在所有 Java 提交中击败了 5.65% 的用户

四. Consider 思考更优解

- ✓ **寻找更好的算法思维**
  - **既然是整数，能否用数学思维？**
  - **借鉴其它算法**

- 求出数组target 中相邻两元素的差值，保留大于 0 的部分，累加即为答案。

如何证明这样做是正确的呢？

五. Code 最优解思路及编码实现

最优解：差分数组

序号	0	1	2	3	4	5	6	7
原始数组 a[x]	0	2	5	4	9	7	10	0
差分数组 d[x]		2	3	-1	5	-2	3	-10

- d[i]=a[i]– a[i-1], a[i]=a[i-1]+d[i]
- 将原始数组中元素同时加上或者减掉某个数，差分数组不会变化
- 对一个区间增减某个值，差分数组左端点的值会同步变化，右端点的后一个值会相反地变化
- 差分数组的作用就是求多次进行区间修改后的数组
- 对数组a某个区间内所有值的加减操作 === 差分数组中两个值的操作

五. Code 最优解思路及编码实现

最优解：差分数组

一种较为直观的证明方法，从左向右考虑数组 target 中的每个数。对于 target[0]，它最少需要操作的次数就为 target[0]；而对于两个相邻的数 target[i] 和 target[i+1]：

- 如果 target[i]≥target[i+1]，那么在给 target[i] 增加 1 时，可以顺便给 target[i+1] 增加 1，这样 target[i+1] 不会占用额外的操作次数；
- 如果 target[i]<target[i+1]，那么即使在target[i] 增加 1 的同时顺便给 target[i+1] 增加 1，那么还需要 target[i+1]−target[i] 次操作才能得到正确的结果。

五. Code 最优解思路及编码实现

最优解：差分数组

这样我们可以得到最少的操作次数为：

$$target[0] + \sum_{i=0}^{n-2} \max\left\{target[i+1] - target[i], 0\right\}$$

但对于本题来说，有一种非常严谨且可以得出操作方案的证明方法，即“差分数组”。

五. Code 最优解思路及编码实现

最优解：差分数组

时间复杂度：O(n)

- 对数组的每一位遍历处理O(n)

空间复杂度：O(1)

- 只需要一个整数变量O(1)

五. Code 最优解思路及编码实现

最优解：数学思维法参考代码

```
class Solution {
    public int minNumberOperations(int[] target) {
        int res = target[0];
        for (int i = 1; i < target.length; i++) {
            res += Math.max(target[i] - target[i - 1], 0);
        }
        return res;
    }
}
```

执行结果：通过 显示详情 >

执行用时：3 ms，在所有 Java 提交中击败了 100.00% 的用户

内存消耗：48.2 MB，在所有 Java 提交中击败了 14.35% 的用户

六. Change 变形延伸

拉勾教育
—互联网人实战大学—

题目变形

- （练习）区域和检索 – 数组可修改 (Leetcode 307)

延伸扩展

- 线段树在可变数组的单点更新、区间查询中有很强应用
- 在面对一些数字类处理的题目时，数学思维可以使问题简单化

本题来源：

- leetcode 1526 https://leetcode.com/problems/minimum-number-of-increments-on-subarrays-to-form-a-target-array/

总结

拉勾教育
—互联网人实战大学—

- 掌握线段树的单点更新和区间查询功能
- 掌握线段树的应用
- 掌握数学思维的应用

课后练习

拉勾教育
—互联网人实战大学—

1. 区域和检索 - 数组可修改(Leetcode 307 /中等)
2. 最后k个数的乘积（ Leetcode 1352 /中等 ）
3. Range模块（ Leetcode 715 /困难 ）
4. 翻转对（ Leetcode 493 /困难 ）

课后练习

1. 区域和检索 - 数组可修改(Leetcode 307 /中等)

提示：这道题体现线段树组的更新和求区域和的基本功能。数组仅可以在 update 函数下进行修改。你可以假设 update 函数与 sumRange 函数的调用次数是均匀分布的。

示例:

```
Given nums = [1, 3, 5]

sumRange(0, 2) -> 9
update(1, 2)
sumRange(0, 2) -> 8
```

课后练习

2. 最后k个数的乘积（Leetcode 1352 /中等）

请你实现一个「数字乘积类」ProductOfNumbers，要求支持下述两种方法：

（1）. add(int num)

将数字 num 添加到当前数字列表的最后面。

（2）. getProduct(int k)

返回当前数字列表中，最后 k 个数字的乘积。

你可以假设当前列表中始终 至少 包含 k 个数字。

题目数据保证：任何时候，任一连续数字序列的乘积都在 32-bit 整数范围内，不会溢出。

课后练习

3. Range模块（Leetcode 715 /困难）

提示：Range 模块是跟踪数字范围的模块。你的任务是以一种有效的方式设计和实现以下接口。

- addRange(int left, int right) 添加半开区间 [left, right)，跟踪该区间中的每个实数。添加与当前跟踪的数字部分重叠的区间时，应当添加在区间 [left, right) 中尚未跟踪的任何数字到该区间中。
- queryRange(int left, int right) 只有在当前正在跟踪区间 [left, right) 中的每一个实数时，才返回 true。
- removeRange(int left, int right) 停止跟踪区间 [left, right) 中当前正在跟踪的每个实数。

```
addRange(10, 20): null
removeRange(14, 16): null
queryRange(10, 14): true （区间 [10, 14) 中的每个数都正在被跟踪）
queryRange(13, 15): false （未跟踪区间 [13, 15) 中像 14, 14.03, 14.17 这样的数字）
queryRange(16, 17): true （尽管执行了删除操作，区间 [16, 17) 中的数字 16 仍然会被跟踪）
```

课后练习

拉勾教育
—互联网人实战大学—

4. 翻转对（Leetcode 493 /困难）

提示：给定数组的长度不会超过50000。输入数组中的所有数字都在32位整数的表示范围内。

```
输入: [1,3,2,3,1]
输出: 2
```

```
输入: [2,4,3,5,1]
输出: 3
```

拉勾教育

—互联网人实战大学—

下载「拉勾教育App」
获取更多内容