宫水三叶的刷题日记

树状数组

Author ：宫水三叶
Date ：2021/10/07
QQ Group：703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「树状数组」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「树状数组」即可获取最新下载链接。

💡**下面介绍使用本合集的最佳使用实践：**

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「树状数组」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **307. 区域和检索 - 数组可修改** ，难度为 **中等**。

Tag :「区间和」、「树状数组」

给你一个数组 nums ，请你完成两类查询，其中一类查询要求更新数组下标对应的值，另一类查询要求返回数组中某个范围内元素的总和。

实现 NumArray 类：

- NumArray(int[] nums) 用整数数组 nums 初始化对象
- void update(int index, int val) 将 nums[index] 的值更新为 val
- int sumRange(int left, int right) 返回子数组 nums[left, right] 的总和（即，nums[left] + nums[left + 1], ..., nums[right]）

示例：

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

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

提示：

- 1 <= nums.length <= 3 * 10^4
- -100 <= nums[i] <= 100
- 0 <= index < nums.length
- -100 <= val <= 100
- 0 <= left <= right < nums.length
- 最多调用 3 * 10^4 次 update 和 sumRange 方法

解题思路

这是一道很经典的题目，通常还能拓展出一大类问题。

针对不同的题目，我们有不同的方案可以选择（假设我们有一个数组）：

1. 数组不变，求区间和：「前缀和」、「树状数组」、「线段树」
2. 多次修改某个数，求区间和：「树状数组」、「线段树」
3. 多次整体修改某个区间，求区间和：「线段树」、「树状数组」（看修改区间的数据范围）
4. 多次将某个区间变成同一个数，求区间和：「线段树」、「树状数组」（看修改区间的数据范围）

这样看来，「线段树」能解决的问题是最多的，那我们是不是无论什么情况都写「线段树」呢？

答案并不是，而且恰好相反，只有在我们遇到第 4 类问题，不得不写「线段树」的时候，我们才考虑线段树。

因为「线段树」代码很长，而且常数很大，实际表现不算很好。我们只有在不得不用的时候才考虑「线段树」。

总结一下，我们应该按这样的优先级进行考虑：

1. 简单求区间和，用「前缀和」
2. 多次将某个区间变成同一个数，用「线段树」
3. 其他情况，用「树状数组」

树状数组

本题显然属于第 2 类问题：多次修改某个数，求区间和。

我们使用「树状数组」进行求解。

「树状数组」本身是一个很简单的数据结构，但是要搞懂其为什么可以这样「查询」&「更新」还是比较困难的（特别是为什么可以这样更新），往往需要从「二进制分解」进行出发理解。

因此我这里直接提供「树状数组」的代码，大家可以直接当做模板背过即可。

代码：

```java
class NumArray {
    int[] tree;
    int lowbit(int x) {
        return x & -x;
    }
    int query(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans += tree[i];
        return ans;
    }
    void add(int x, int u) {
        for (int i = x; i <= n; i += lowbit(i)) tree[i] += u;
    }

    int[] nums;
    int n;
    public NumArray(int[] _nums) {
        nums = _nums;
        n = nums.length;
        tree = new int[n + 1];
        for (int i = 0; i < n; i++) add(i + 1, nums[i]);
    }

    public void update(int i, int val) {
        add(i + 1, val - nums[i]);
        nums[i] = val;
    }

    public int sumRange(int l, int r) {
        return query(r + 1) - query(l);
    }
}
```

- 时间复杂度：`add` 操作和 `query` 的复杂度都是 $O(\log n)$，因此构建数组的复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

树状数组模板

代码：

这是 LeetCode 上的 **354. 俄罗斯套娃信封问题** ，难度为 **困难**。

```
// 上来先把三个方法写出来
{
    int[] tree;
    int lowbit(int x) {
        return x & -x;
    }
    // 查询前缀和的方法
    int query(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans += tree[i];
        return ans;
    }
    // 在树状数组 x 位置中增加值 u
    void add(int x, int u) {
        for (int i = x; i <= n; i += lowbit(i)) tree[i] += u;
    }
}

// 初始化「树状数组」，要默认数组是从 1 开始
{
    for (int i = 0; i < n; i++) add(i + 1, nums[i]);
}

// 使用「树状数组」：
{
    void update(int i, int val) {
        // 原有的值是 nums[i]，要使得修改为 val，需要增加 val - nums[i]
        add(i + 1, val - nums[i]);
        nums[i] = val;
    }

    int sumRange(int l, int r) {
        return query(r + 1) - query(l);
    }
}
```

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **354. 俄罗斯套娃信封问题** ，难度为 **困难**。

Tag :「二分」、「序列 DP」

给你一个二维整数数组 envelopes ，其中 envelopes[i] = [wi, hi] ，表示第 i 个信封的宽度和高度。

当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算「最多能有多少个」信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

注意：不允许旋转信封。

示例 1：

```
输入：envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出：3
解释：最多信封的个数为 3, 组合为: [2,3] => [5,4] => [6,7]。
```

示例 2：

```
输入：envelopes = [[1,1],[1,1],[1,1]]
输出：1
```

提示：

- 1 <= envelopes.length <= 5000
- envelopes[i].length == 2
- 1 <= wi, hi <= 10^4

动态规划

这是一道经典的 DP 模型题目：最长上升子序列（LIS）。

首先我们先对 `envelopes` 进行排序，确保信封是从小到大进行排序。

问题就转化为我们从这个序列中选择 `k` 个信封形成新的序列，使得新序列中的每个信封都能严格覆盖前面的信封（宽高都严格大于）。

我们可以**定义状态 $f[i]$ 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值。**

对于每个 $f[i]$ 而言，最小值为 1，代表只选择自己一个信封。

那么对于一般的 $f[i]$ 该如何求解呢？因为第 i 件物品是必须选择的。我们可以枚举前面的 $i-1$ 件物品，哪一件可以作为第 i 件物品的上一件物品。

在前 $i-1$ 件物品中只要有符合条件的，我们就使用 $max(f[i], f[j]+1)$ 更新 $f[i]$。

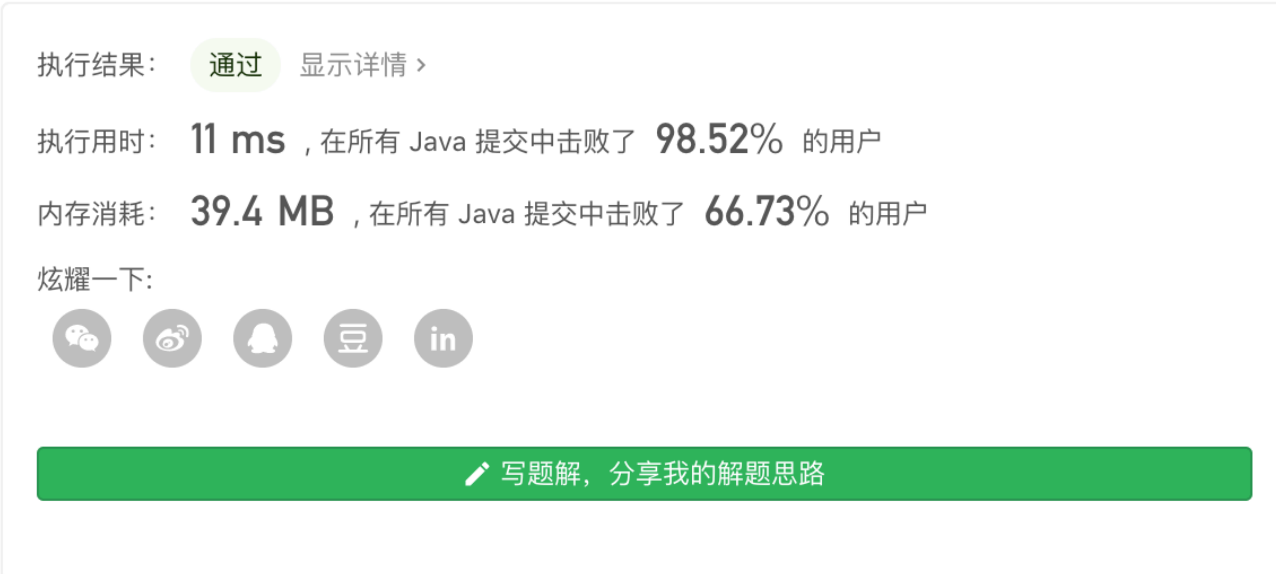
然后在所有方案中取一个 `max` 即是答案。

代码：

```java
class Solution {
    public int maxEnvelopes(int[][] es) {
        int n = es.length;
        if (n == 0) return n;
        // 因为我们在找第 i 件物品的前一件物品时，会对前面的 i - 1 件物品都遍历一遍，因此第二维（高度）
        Arrays.sort(es, (a, b)->a[0]-b[0]);
        int[] f = new int[n]; // f(i) 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值
        int ans = 1;
        for (int i = 0; i < n; i++) {
            // 对于每个 f[i] 都满足最小值为 1
            f[i] = 1;
            // 枚举第 i 件物品的前一件物品，
            for (int j = i - 1; j >= 0; j--) {
                // 只要有满足条件的前一件物品，我们就尝试使用 f[j] + 1 更新 f[i]
                if (check(es, j, i)) {
                    f[i] = Math.max(f[i], f[j] + 1);
                }
            }
            // 在所有的 f[i] 中取 max 作为 ans
            ans = Math.max(ans, f[i]);
        }
        return ans;
    }
    boolean check(int[][] es, int mid, int i) {
        return es[mid][0] < es[i][0] && es[mid][1] < es[i][1];
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

二分 + 动态规划



上述方案其实算是一个朴素方案，复杂度是 $O(n^2)$ 的，也是我最先想到思路，但是题目没有给出数据范围，也不知道能不能过。

唯唯诺诺交了一个居然过了。

下面讲下其他优化解法。

首先还是和之前一样，我们可以通过复杂度分析来想优化方向。

指数算法往下优化就是对数解法或者线性解法。

仔细观察朴素解法，其实可优化的地方主要就是找第 i 件物品的前一件物品的过程。

如果想要加快这个查找过程，我们需要使用某种数据结构进行记录。

并且是边迭代边更新数据结构里面的内容。

首先因为我们对 `w` 进行了排序（从小到大），然后迭代也是从前往后进行，因此我们只需要保证迭代过程中，对于 `w` 相同的数据不更新，就能保证 `g` 中只会出现满足 `w` 条件的信封。

到这一步，还需要用到的东西有两个：一个是 `h`，因为只有 `w` 和 `h` 都同时满足，我们才能加入上升序列中；一个是信封所对应的上升序列长度，这是我们加速查找的核心。

我们使用数组 `g` 来记录，$g[i]$ 表示长度为 i 的最长上升子序列的中的最小「信封高度」，同时

需要使用 `len` 记录当前记录到的最大长度。

还是不理解？没关系，我们可以直接看看代码，我把基本逻辑写在了注释当中（你的重点应该落在对 $g[]$ 数组的理解上）。

代码：

```java
class Solution {
    public int maxEnvelopes(int[][] es) {
        int n = es.length;
        if (n == 0) return n;
        // 由于我们使用了 g 记录高度，因此这里只需将 w 从小到达排序即可
        Arrays.sort(es, (a, b)->a[0] - b[0]);
        // f(i) 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值
        int[] f = new int[n];
        // g(i) 记录的是长度为 i 的最长上升子序列的最小「信封高度」
        int[] g = new int[n];
        // 因为要取 min，用一个足够大（不可能）的高度初始化
        Arrays.fill(g, Integer.MAX_VALUE);
        g[0] = 0;
        int ans = 1;
        for (int i = 0, j = 0, len = 1; i < n; i++) {
            // 对于 w 相同的数据，不更新 g 数组
            if (es[i][0] != es[j][0]) {
                // 限制 j 不能越过 i，确保 g 数组中只会出现第 i 个信封前的「历史信封」
                while (j < i) {
                    int prev = f[j], cur = es[j][1];
                    if (prev == len) {
                        // 与当前长度一致了，说明上升序列多增加一位
                        g[len++] = cur;
                    } else {
                        // 始终保留最小的「信封高度」，这样可以确保有更多的信封可以与其行程上升序列
                        // 举例：同样是上升长度为 5 的序列，保留最小高度为 5 记录（而不是保留任意的，
                        g[prev] = Math.min(g[prev], cur);
                    }
                    j++;
                }
            }

            // 二分过程
            // g[i] 代表的是上升子序列长度为 i 的「最小信封高度」
            int l = 0, r = len;
            while (l < r) {
                int mid = l + r >> 1;
                // 令 check 条件为 es[i][1] <= g[mid]（代表 w 和 h 都严格小于当前信封）
                // 这样我们找到的就是满足条件，最靠近数组中心点的数据（也就是满足 check 条件的最大下标
                // 对应回 g[] 数组的含义，其实就是找到 w 和 h 都满足条件的最大上升长度
                if (es[i][1] <= g[mid]) {
                    r = mid;
                } else {
                    l = mid + 1;
                }
            }
```

```
            // 更新 f[i] 与答案
            f[i] = r;
            ans = Math.max(ans, f[i]);
        }
        return ans;
    }
}
```

- 时间复杂度：对于每件物品都是通过「二分」找到其前一件物品。复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

证明

我们可以这样做的前提是 `g` 数组具有二段性，可以通过证明其具有「单调性」来实现。

当然这里指的是 `g` 被使用的部分，也就是 $[0, len - 1]$ 的部分。

我们再回顾一下 `g[]` 数组的定义：$g[i]$ **表示长度为 i 的最长上升子序列的中的最小「信封高度」**

例如 $g[] = [0, 3, 4, 5]$ 代表的含义是：

- 上升序列长度为 0 的最小历史信封高度为 0
- 上升序列长度为 1 的最小历史信封高度为 3
- 上升序列长度为 2 的最小历史信封高度为 4
- 上升序列长度为 3 的最小历史信封高度为 5

可以通过反证法来证明其单调性:

假设 $g[]$ 不具有单调性，即至少有 $g[i] > g[j]$ ($i < j$，令 $a = g[i], b = g[j]$)

显然与我们的处理逻辑冲突。因为如果考虑一个「最小高度」为 `b` 的信封能够凑出长度为 `j` 的上升序列，自然也能凑出比 `j` 短的上升序列，对吧？

举个🌰，我们有信封：1,1],[2,2],[3,3],[4,4],[5,5，我们能凑出很多种长度为 2 的上升序列方案，其中最小的方案是高度最小的方案是 1,1],[2,2。因此这时候 g[2] = 2，代表能凑出长度为 2 的上升序列所 **必须使用的信封** 的最小高度为 2。

这时候反过来考虑，如果使用 [2,2] 能够凑出长度为 2 的上升序列，必然也能凑出长度为 1 的上升序列（删除前面的其他信封即可）。

推而广之，如果我们有 $g[j] = b$，也就是凑成长度为 `j` 必须使用的最小信封高度为 b。那么我必然能够保留高度为 `b` 的信封，删掉上升序列中的一些信封，凑成任意长度比 `j` 小的上升序列。

综上，$g[i] > g[j]$（$i < j$）与处理逻辑冲突，$g[]$ 数组为严格单调上升数组。

既然 $g[]$ 具有单调性，我们可以通过「二分」找到恰满足 check 条件的最大下标（最大下标达标表示最长上升序列长度）。

树状数组 + 动态规划

在「二分 + 动态规划」的解法中，我们通过「二分」来优化找第 i 个文件的前一个文件过程。

这个过程同样能通过「树状数组」来实现。

首先仍然是对 `w` 进行排序，然后使用「树状数组」来维护 h 维度的前缀最大值。

对于 `h` 的高度，我们只关心多个信封之间的大小关系，而不关心具体相差多少，我们需要对 `h` 进行离散化。

通常使用「树状数组」都需要进行离散化，尤其是这里我们本身就要使用 $O(n)$ 的空间来存储 dp 值。

代码：

```java
class Solution {
    int[] tree;
    int lowbit(int x) {
        return x & -x;
    }

    public int maxEnvelopes(int[][] es) {
        int n = es.length;
        if (n == 0) return n;

        // 由于我们使用了 g 记录高度，因此这里只需将 w 从小到达排序即可
        Arrays.sort(es, (a, b)->a[0] - b[0]);

        // 先将所有的 h 进行离散化
        Set<Integer> set = new HashSet<>();
        for (int i = 0; i < n; i++) set.add(es[i][1]);
        int cnt = set.size();
        int[] hs = new int[cnt];
        int idx = 0;
        for (int i : set) hs[idx++] = i;
        Arrays.sort(hs);
        for (int i = 0; i < n; i++) es[i][1] = Arrays.binarySearch(hs, es[i][1]) + 1;

        // 创建树状数组
        tree = new int[cnt + 1];

        // f(i) 为考虑前 i 个物品，并以第 i 个物品为结尾的最大值
        int[] f = new int[n];
        int ans = 1;
        for (int i = 0, j = 0; i < n; i++) {
            // 对于 w 相同的数据，不更新 tree 数组
            if (es[i][0] != es[j][0]) {
                // 限制 j 不能越过 i，确保 tree 数组中只会出现第 i 个信封前的「历史信封」
                while (j < i) {
                    for (int u = es[j][1]; u <= cnt; u += lowbit(u)) {
                        tree[u] = Math.max(tree[u], f[j]);
                    }
                    j++;
                }
            }
            f[i] = 1;
            for (int u = es[i][1] - 1; u > 0; u -= lowbit(u)) {
                f[i] = Math.max(f[i], tree[u] + 1);
            }
            ans = Math.max(ans, f[i]);
        }
```

```
        return ans;
    }
}
```

- 时间复杂度：处理每个物品时更新「树状数组」复杂度为$O(\log n)$。整体复杂度为$O(n \log n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **673. 最长递增子序列的个数** ，难度为 **中等**。

Tag : 「动态规划」、「序列 DP」、「树状数组」、「最长上升子序列」

给定一个未排序的整数数组，找到最长递增子序列的个数。

示例 1:

```
输入: [1,3,5,4,7]

输出: 2

解释: 有两个最长递增子序列，分别是 [1, 3, 4, 7] 和[1, 3, 5, 7]。
```

示例 2:

```
输入: [2,2,2,2,2]

输出: 5

解释: 最长递增子序列的长度是1，并且存在5个子序列的长度为1，因此输出5。
注意: 给定的数组长度不超过 2000 并且结果一定是32位有符号整数。
```

序列 DP

与朴素的 LIS 问题（问长度）相比，本题问的是最长上升子序列的个数。

我们只需要在朴素 LIS 问题的基础上通过「记录额外信息」来进行求解即可。

在朴素的 LIS 问题中，我们定义 $f[i]$ **为考虑以 $nums[i]$ 为结尾的最长上升子序列的长度。** 最终答案为所有 $f[0...(n-1)]$ 中的最大值。

不失一般性地考虑 $f[i]$ 该如何转移：

- 由于每个数都能独自一个成为子序列，因此起始必然有 $f[i] = 1$；
- 枚举区间 $[0, i)$ 的所有数 $nums[j]$，如果满足 $nums[j] < nums[i]$，说明 $nums[i]$ 可以接在 $nums[j]$ 后面形成上升子序列，此时使用 $f[j]$ 更新 $f[i]$，即有 $f[i] = f[j] + 1$。

回到本题，由于我们需要求解的是最长上升子序列的个数，因此需要额外定义 $g[i]$ **为考虑以 $nums[i]$ 结尾的最长上升子序列的个数。**

结合 $f[i]$ 的转移过程，不失一般性地考虑 $g[i]$ 该如何转移：

- 同理，由于每个数都能独自一个成为子序列，因此起始必然有 $g[i] = 1$；
- 枚举区间 $[0, i)$ 的所有数 $nums[j]$，如果满足 $nums[j] < nums[i]$，说明 $nums[i]$ 可以接在 $nums[j]$ 后面形成上升子序列，这时候对 $f[i]$ 和 $f[j] + 1$ 的大小关系进行分情况讨论：
  - 满足 $f[i] < f[j] + 1$：说明 $f[i]$ 会被 $f[j] + 1$ 直接更新，此时同步直接更新 $g[i] = g[j]$ 即可；
  - 满足 $f[i] = f[j] + 1$：说明找到了一个新的符合条件的前驱，此时将值继续累加到方案数当中，即有 $g[i] += g[j]$。

在转移过程，我们可以同时记录全局最长上升子序列的最大长度 max，最终答案为所有满足 $f[i] = max$ 的 $g[i]$ 的累加值。

代码：

```java
class Solution {
    public int findNumberOfLIS(int[] nums) {
        int n = nums.length;
        int[] f = new int[n], g = new int[n];
        int max = 1;
        for (int i = 0; i < n; i++) {
            f[i] = g[i] = 1;
            for (int j = 0; j < i; j++) {
                if (nums[j] < nums[i]) {
                    if (f[i] < f[j] + 1) {
                        f[i] = f[j] + 1;
                        g[i] = g[j];
                    } else if (f[i] == f[j] + 1) {
                        g[i] += g[j];
                    }
                }
            }
            max = Math.max(max, f[i]);
        }
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (f[i] == max) ans += g[i];
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

LIS 问题的贪心解 + 树状数组

我们知道，对于朴素的 LIS 问题存在贪心解法，能够在 $O(n\log n)$ 复杂度内求解 LIS 问题。

在贪心解中，我们会多开一个贪心数组 q，用来记录长度为 len 的最长上升子序列的「最小结尾元素」为何值：$q[len] = x$ **代表长度为 len 的最长上升子序列的最小结尾元素为 x**。

可以证明 q 存在单调性，因此每次确定 $nums[i]$ 可以接在哪个 $nums[j]$ 后面会形成最长上升子序列时，可以通过「二分」来找到满足 $nums[j] < nums[i]$ 的最大下标来实现。

对于本题，由于我们需要求最长上升子序列的个数，单纯使用一维的贪心数组记录最小结尾元素

并不足以。

考虑对其进行扩展，期望能取到「最大长度」的同时，能够知道这个「最大长度」对应多少个子序列数量，同时期望该操作复杂度为 $O(\log n)$。

我们可以使用「树状数组」维护二元组 (len, cnt) 信息：

1. 因为数据范围较大（$-10^6 <= nums[i] <= 10^6$），但数的个数为 2000，因此第一步先对 $nums$ 进行离散化操作；
2. 在遍历 $nums$ 时，每次从树状数组中查询值严格小于 $nums[i]$ 离散值（利用 $nums[i]$ 离散化后的值仍为正整数，我们可以直接查询小于等于 $nums[i]$ 离散值 -1 的值）的最大长度，及最大长度对应的数量；
3. 对于流程 2 中查得的 (len, cnt)，由于 $nums[i]$ 可以接在其后，因此首先长度加一，同时数量将 cnt 累加到该离散值中。

代码：

```java
class Solution {
    int n;
    int[][] tr = new int[2010][2];
    int lowbit(int x) {
        return x & -x;
    }
    int[] query(int x) {
        int len = 0, cnt = 0;
        for (int i = x; i > 0; i -= lowbit(i)) {
            if (len == tr[i][0]) {
                cnt += tr[i][1];
            } else if (len < tr[i][0]) {
                len = tr[i][0];
                cnt = tr[i][1];
            }
        }
        return new int[]{len, cnt};
    }
    void add(int x, int[] info) {
        for (int i = x; i <= n; i += lowbit(i)) {
            int len = tr[i][0], cnt = tr[i][1];
            if (len == info[0]) {
                cnt += info[1];
            } else if (len < info[0]) {
                len = info[0];
                cnt = info[1];
            }
            tr[i][0] = len; tr[i][1] = cnt;
        }
    }
    public int findNumberOfLIS(int[] nums) {
        n = nums.length;
        // 离散化
        int[] tmp = nums.clone();
        Arrays.sort(tmp);
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0, idx = 1; i < n; i++) {
            if (!map.containsKey(tmp[i])) map.put(tmp[i], idx++);
        }
        // 树状数组维护 (len, cnt) 信息
        for (int i = 0; i < n; i++) {
            int x = map.get(nums[i]);
            int[] info = query(x - 1);
            int len = info[0], cnt = info[1];
            add(x, new int[]{len + 1, Math.max(cnt, 1)});
        }
```

```
        int[] ans = query(n);
        return ans[1];
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1310. 子数组异或查询** ，难度为 **中等**。

Tag : 「数学」、「树状数组」、「前缀和」

有一个正整数数组 arr，现给你一个对应的查询数组 queries，其中 queries[i] = [Li, Ri]。

对于每个查询 i，请你计算从 Li 到 Ri 的 XOR 值（即 arr[Li] xor arr[Li+1] xor ... xor arr[Ri]）作为本次查询的结果。

并返回一个包含给定查询 queries 所有结果的数组。

示例 1：

```
输入：arr = [1,3,4,8], queries = [[0,1],[1,2],[0,3],[3,3]]

输出：[2,7,14,8]

解释：
数组中元素的二进制表示形式是：
1 = 0001
3 = 0011
4 = 0100
8 = 1000
查询的 XOR 值为：
[0,1] = 1 xor 3 = 2
[1,2] = 3 xor 4 = 7
[0,3] = 1 xor 3 xor 4 xor 8 = 14
[3,3] = 8
```

示例 2：

```
输入：arr = [4,8,2,10], queries = [[2,3],[1,3],[0,0],[0,3]]

输出：[8,0,4,4]
```

提示：

- 1 <= arr.length <= 3 * 10^4
- 1 <= arr[i] <= 10^9
- 1 <= queries.length <= 3 * 10^4
- queries[i].length == 2
- 0 <= queries[i][0] <= queries[i][1] < arr.length

基本分析

令数组 `arr` 和数组 `queries` 的长度分别为 `n` 和 `m`。

`n` 和 `m` 的数据范围均为 10^4，因此 $O(m * n)$ 的暴力做法我们不用考虑了。

数据范围要求我们做到「对数复杂度」或「线性复杂度」。

本题主要利用异或运算中的「相同数值进行运算结果为 0」的特性。

对于特定数组 $[a1, a2, a3, ..., an]$，要求得任意区间 $[l, r]$ 的异或结果，可以通过 $[1, r]$ 和 $[1, l-1]$ 的异或结果得出：

$$xor(l, r) = xor(1, r) \oplus xor(1, l-1)$$

本质上还是利用集合（区间结果）的容斥原理。只不过前缀和需要利用「减法（逆运算）」做容斥，而前缀异或是利用「相同数值进行异或结果为 0（偶数次的异或结果为 0）」的特性实现容斥。

对于「区间求值」问题，之前在 【题解】307. 区域和检索 - 数组可修改 也做过总结。

针对不同的题目，有不同的方案可以选择（假设有一个数组）：

1. 数组不变，求区间和：「前缀和」、「树状数组」、「线段树」

2. 多次修改某个数，求区间和：「树状数组」、「线段树」
3. 多次整体修改某个区间，求区间和：「线段树」、「树状数组」（看修改区间的数据范围）
4. 多次将某个区间变成同一个数，求区间和：「线段树」、「树状数组」（看修改区间的数据范围）

虽然「线段树」能解决的问题最多，但「线段树」代码很长，且常数很大，实际表现不算好。我们只有在不得不用的情况下才考虑「线段树」。

本题我们使用「树状数组」和「前缀和」来求解。

树状数组

使用「树状数组」分段记录我们某些区间的「异或结果」，再根据 `queries` 中的询问将分段「异或结果」汇总（执行异或运算），得出最终答案。

代码：

```java
class Solution {
    int n;
    int[] c = new int[100009];
    int lowbit(int x) {
        return x & -x;
    }
    void add(int x, int u) {
        for (int i = x; i <= n; i += lowbit(i)) c[i] ^= u;
    }
    int query(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans ^= c[i];
        return ans;
    }
    public int[] xorQueries(int[] arr, int[][] qs) {
        n = arr.length;
        int m = qs.length;
        for (int i = 1; i <= n; i++) add(i, arr[i - 1]);
        int[] ans = new int[m];
        for (int i = 0; i < m; i++) {
            int l = qs[i][0] + 1, r = qs[i][1] + 1;
            ans[i] = query(r) ^ query(l - 1);
        }
        return ans;
    }
}
```

- 时间复杂度：令 `arr` 数组长度为 `n`，`qs` 数组的长度为 `m`。创建树状数组复杂度为 $O(n\log n)$；查询的复杂度为 $O(m\log n)$。整体复杂度为 $O((n+m)\log n)$
- 空间复杂度：$O(n)$

前缀异或

「树状数组」的查询复杂度为 $O(\log n)$，而本题其实不涉及「修改操作」，我们可以使用「前缀异或」来代替「树状数组」。

虽说「树状数组」也有 $O(n)$ 的创建方式，但这里使用「前缀异或」主要是为了降低查询的复杂度。

代码：

```java
class Solution {
    public int[] xorQueries(int[] arr, int[][] qs) {
        int n = arr.length, m = qs.length;
        int[] sum = new int[n + 1];
        for (int i = 1; i <= n; i++) sum[i] = sum[i - 1] ^ arr[i - 1];
        int[] ans = new int[m];
        for (int i = 0; i < m; i++) {
            int l = qs[i][0] + 1, r = qs[i][1] + 1;
            ans[i] = sum[r] ^ sum[l - 1];
        }
        return ans;
    }
}
```

- 时间复杂度：令 `arr` 数组长度为 `n`，`qs` 数组的长度为 `m`。预处理前缀和数组复杂度为 $O(n)$；查询的复杂度为 $O(m)$。整体复杂度为 $O(n+m)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **1893. 检查是否区域内所有整数都被覆盖** ，难度为 **简单**。

Tag : 「模拟」、「树状数组」、「线段树」

给你一个二维整数数组 ranges 和两个整数 left 和 right 。每个 ranges[i] = [starti, endi] 表示一个从 starti 到 endi 的 闭区间 。

如果闭区间 [left, right] 内每个整数都被 ranges 中 至少一个 区间覆盖，那么请你返回 true ，否则返回 false 。

已知区间 ranges[i] = [starti, endi] ，如果整数 x 满足 starti <= x <= endi ，那么我们称整数x 被覆盖了。

示例 1：

```
输入：ranges = [[1,2],[3,4],[5,6]], left = 2, right = 5

输出：true

解释：2 到 5 的每个整数都被覆盖了：
- 2 被第一个区间覆盖。
- 3 和 4 被第二个区间覆盖。
- 5 被第三个区间覆盖。
```

示例 2：

```
输入：ranges = [[1,10],[10,20]], left = 21, right = 21

输出：false

解释：21 没有被任何一个区间覆盖。
```

提示：

- 1 <= ranges.length <= 50
- 1 <= starti <= endi <= 50
- 1 <= left <= right <= 50

模拟

一个简单的想法是根据题意进行模拟，检查 $[left, right]$ 中的每个整数，如果检查过程中发现某个整数没被 $ranges$ 中的闭区间所覆盖，那么直接返回 `False`，所有数值通过检查则返回 `True`。

代码：

```
class Solution {
    public boolean isCovered(int[][] rs, int l, int r) {
        for (int i = l; i <= r; i++) {
            boolean ok = false;
            for (int[] cur : rs) {
                int a = cur[0], b = cur[1];
                if (a <= i && i <= b) {
                    ok = true;
                    break;
                }
            }
            if (!ok) return false;
        }
        return true;
    }
}
```

- 时间复杂度：令 $[left, right]$ 之间整数数量为 n，$ranges$ 长度为 m。整体复杂度为 $O(n * m)$
- 空间复杂度：$O(1)$

树状数组

针对此题，可以有一个很有意思的拓展，将本题难度提升到【中等】甚至是【困难】。

将查询 $[left, right]$ 修改为「四元查询数组」$querys$，每个 $querys[i]$ 包含四个指标 (a, b, l, r)：代表询问 $[l, r]$ 中的每个数是否在 $range$ 中 $[a, b]$ 的闭区间所覆盖过。

如果进行这样的拓展的话，那么我们需要使用「持久化树状数组」或者「主席树」来配合「容斥原理」来做。

基本思想都是使用 $range[0, b]$ 的计数情况减去 $range[0, a-1]$ 的计数情况来得出 $[a, b]$ 的计数情况。

回到本题，由于数据范围很小，只有 50，我们可以使用「树状数组」进行求解：

- `void add(int x, int u)`：对于数值 x 出现次数进行 $+u$ 操作；
- `int query(int x)`：查询某个满足 $<= x$ 的数值的个数。

那么显然，如果我们需要查询一个数值 x 是否出现过，可以通过查询 $cnt = query(x) -$

$query(x-1)$ 来得知。

代码：

```
class Solution {
    int n = 55;
    int[] tr = new int[n];
    int lowbit(int x) {
        return x & -x;
    }
    void add(int x, int u) {
        for (int i = x; i <= n; i += lowbit(i)) tr[i] += u;
    }
    int query(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans += tr[i];
        return ans;
    }
    public boolean isCovered(int[][] rs, int l, int r) {
        for (int[] cur : rs) {
            int a = cur[0], b = cur[1];
            for (int i = a; i <= b; i++) {
                add(i, 1);
            }
        }
        for (int i = l; i <= r; i++) {
            int cnt = query(i) - query(i - 1);
            if (cnt == 0) return false;
        }
        return true;
    }
}
```

- 时间复杂度：令 $[left, right]$ 之间整数数量为 n，$\sum_{i=0}^{range.legth-1} ranges[i].length$ 为 sum，常数 C 固定为 55。建树复杂度为 $O(sum\log C)$，查询查询复杂度为 $O(n\log C)$。整体复杂度为 $O(sum\log C + n\log C)$
- 空间复杂度：$O(C)$

树状数组（去重优化）

在朴素的「树状数组」解法中，我们无法直接查询 $[l, r]$ 区间中被覆盖过的个数的根本原因是「某个值可能会被重复添加到树状数组中」。

因此，一种更加优秀的做法：**在往树状数组中添数的时候进行去重，然后通过 $cnt = query(r) - query(l - 1)$ 直接得出 $[l, r]$ 范围内有多少个数被添加过。**

这样的 `Set` 去重操作可以使得我们查询的复杂度从 $O(n \log C)$ 下降到 $O(\log C)$。

> 由于数值范围很小，自然也能够使用数组来代替 `Set` 进行标记（见 P2）

代码：

```java
class Solution {
    int n = 55;
    int[] tr = new int[n];
    int lowbit(int x) {
        return x & -x;
    }
    void add(int x, int u) {
        for (int i = x; i <= n; i += lowbit(i)) tr[i] += u;
    }
    int query(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans += tr[i];
        return ans;
    }
    public boolean isCovered(int[][] rs, int l, int r) {
        Set<Integer> set = new HashSet<>();
        for (int[] cur : rs) {
            int a = cur[0], b = cur[1];
            for (int i = a; i <= b; i++) {
                if (!set.contains(i)) {
                    add(i, 1);
                    set.add(i);
                }
            }
        }
        int tot = r - l + 1, cnt = query(r) - query(l - 1);
        return tot == cnt;
    }
}
```

```java
class Solution {
    int n = 55;
    int[] tr = new int[n];
    boolean[] vis = new boolean[n];
    int lowbit(int x) {
        return x & -x;
    }
    void add(int x, int u) {
        for (int i = x; i <= n; i += lowbit(i)) tr[i] += u;
    }
    int query(int x) {
        int ans = 0;
        for (int i = x; i > 0; i -= lowbit(i)) ans += tr[i];
        return ans;
    }
    public boolean isCovered(int[][] rs, int l, int r) {
        for (int[] cur : rs) {
            int a = cur[0], b = cur[1];
            for (int i = a; i <= b; i++) {
                if (!vis[i]) {
                    add(i, 1);
                    vis[i] = true;
                }
            }
        }
        int tot = r - l + 1, cnt = query(r) - query(l - 1);
        return tot == cnt;
    }
}
```

- 时间复杂度：令 $[left, right]$ 之间整数数量为 n，$\sum_{i=0}^{range.legth-1} ranges[i].length$ 为 sum，常数 C 固定为 55。建树复杂度为 $O(sum \log C)$，查询查询复杂度为 $O(\log C)$。整体复杂度为 $O(sum \log C + \log C)$
- 空间复杂度：$O(C + \sum_{i=0}^{range.legth-1} ranges[i].length)$

线段树（不含“懒标记”）

更加进阶的做法是使用「线段树」来做，与「树状数组（优化）」解法一样，线段树配合持久化也可以用于求解「在线」问题。

与主要解决「单点修改 & 区间查询」的树状数组不同，线段树能够解决绝大多数「区间修改（区间修改/单点修改）& 区间查询」问题。

对于本题，由于数据范围只有 55，因此我们可以使用与「树状数组（优化）」解法相同的思路，实现一个不包含“懒标记”的线段树来做（仅支持单点修改 & 区间查询）。

代码：

```java
class Solution {
    // 代表 [l, r] 区间有 cnt 个数被覆盖
    class Node {
        int l, r, cnt;
        Node (int _l, int _r, int _cnt) {
            l = _l; r = _r; cnt = _cnt;
        }
    }
    int N = 55;
    Node[] tr = new Node[N * 4];
    void pushup(int u) {
        tr[u].cnt = tr[u << 1].cnt + tr[u << 1 | 1].cnt;
    }
    void build(int u, int l, int r) {
        if (l == r) {
            tr[u] = new Node(l, r, 0);
        } else {
            tr[u] = new Node(l, r, 0);
            int mid = l + r >> 1;
            build(u << 1, l, mid);
            build(u << 1 | 1, mid + 1, r);
            pushup(u);
        }
    }
    // 从 tr 数组的下标 u 开始，在数值 x 的位置进行标记
    void update(int u, int x) {
        if (tr[u].l == x && tr[u].r == x) {
            tr[u].cnt = 1;
        } else {
            int mid = tr[u].l + tr[u].r >> 1;
            if (x <= mid) update(u << 1, x);
            else update(u << 1 | 1, x);
            pushup(u);
        }
    }
    // 从 tr 数组的下标 u 开始，查询 [l,r] 范围内有多少个数值被标记
    int query(int u, int l, int r) {
        if (l <= tr[u].l && tr[u].r <= r) return tr[u].cnt;
        int mid = tr[u].l + tr[u].r >> 1;
        int ans = 0;
        if (l <= mid) ans += query(u << 1, l, r);
        if (r > mid) ans += query(u << 1 | 1, l, r);
        return ans;
    }
    public boolean isCovered(int[][] rs, int l, int r) {
        build(1, 1, N);
```

```java
        for (int[] cur : rs) {
            int a = cur[0], b = cur[1];
            for (int i = a; i <= b; i++) {
                update(1, i);
            }
        }
        int tot = r - l + 1 , cnt = query(1, l, r);
        return tot == cnt;
    }
}
```

- 时间复杂度：令 $[left, right]$ 之间整数数量为 n，$\sum_{i=0}^{range.legth-1} ranges[i].length$ 为 sum，常数 C 固定为 55。建树复杂度为 $O(sum \log C)$，查询查询复杂度为 $O(\log C)$。整体复杂度为 $O(sum \log C + \log C)$
- 空间复杂度：$O(C * 4)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

💡**更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。**

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「树状数组」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：


“给作者手机充个电”
YOLO 的赞赏码