宫水三叶的刷题日记

单调栈

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「单调栈」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「单调栈」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「单调栈」；
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 **42. 接雨水** ，难度为 **困难**。

Tag : 「单调栈」、「数学」

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1：

输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]

输出：6

解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。

示例 2：

输入：height = [4,2,0,3,2,5]

输出：9

提示：

- n == height.length
- 0 <= n <= 3 * 10^4
- 0 <= height[i] <= 10^5

朴素解法

对于每根柱子而言，我们只需要找出「其左边最高的柱子」和「其右边最高的柱子」。

对左右的最高柱子取一个最小值，再和当前柱子的高度做一个比较，即可得出当前位置可以接下的雨水。

同时，边缘的柱子不可能接到雨水（某一侧没有柱子）。

这样的做法属于「暴力做法」，但题目没有给数据范围，我们无法分析到底能否 AC。

唯唯诺诺交一个，过了 ~（好题，建议加入蓝桥杯

代码：

```
class Solution {
    public int trap(int[] height) {
        int n = height.length;
        int ans = 0;
        for (int i = 1; i < n - 1; i++) {
            int cur = height[i];

            // 获取当前位置的左边最大值
            int l = Integer.MIN_VALUE;
            for (int j = i - 1; j >= 0; j--) l = Math.max(l, height[j]);
            if (l <= cur) continue;

            // 获取当前位置的右边边最大值
            int r = Integer.MIN_VALUE;
            for (int j = i + 1; j < n; j++) r = Math.max(r, height[j]);
            if (r <= cur) continue;

            // 计算当前位置可接的雨水
            ans += Math.min(l, r) - cur;
        }
        return ans;
    }
}
```

- 时间复杂度：需要处理所有非边缘的柱子，复杂度为 $O(n)$；对于每根柱子而言，需要往两边扫描分别找到最大值，复杂度为 $O(n)$。整体复杂度为 $O(n^2)$
- 空间复杂度：$O(1)$

预处理最值解法

朴素解法的思路有了，我们想想怎么优化。

事实上，任何的优化无非都是「减少重复」。

想想在朴素思路中有哪些环节比较耗时，耗时环节中又有哪些地方是重复的，可以优化的。

首先对每根柱子进行遍历，求解每根柱子可以接下多少雨水，这个 $O(n)$ 操作肯定省不了。

但在求解某根柱子可以接下多少雨水时，需要对两边进行扫描，求两侧的最大值。每一根柱子都

进行这样的扫描操作，导致每个位置都被扫描了 n 次。这个过程显然是可优化的。

换句话说：**我们希望通过不重复遍历的方式找到任意位置的两侧最大值。**

问题转化为：**给定一个数组，如何求得任意位置的左半边的最大值和右半边的最大值。**

一个很直观的方案是：**直接将某个位置的两侧最大值存起来。**

我们可以先从两端分别出发，预处理每个位置的「左右最值」，这样可以将我们「查找左右最值」的复杂度降到 $O(1)$。

整体算法的复杂度也从 $O(n^2)$ 下降到 $O(n)$。

代码：

```java
class Solution {
    public int trap(int[] height) {
        int n = height.length;
        int ans = 0;
        // 由于预处理最值的时候，我们会直接访问到 height[0] 或者 height[n - 1]，因此要特判一下
        if (n == 0) return ans;

        // 预处理每个位置左边的最值
        int[] lm = new int[n];
        lm[0] = height[0];
        for (int i = 1; i < n; i++) lm[i] = Math.max(height[i], lm[i - 1]);

        // 预处理每个位置右边的最值
        int[] rm = new int[n];
        rm[n - 1] = height[n - 1];
        for (int i = n - 2; i >= 0; i--) rm[i] = Math.max(height[i], rm[i + 1]);

        for (int i = 1; i < n - 1; i++) {
            int cur = height[i];

            int l = lm[i];
            if (l <= cur) continue;

            int r = rm[i];
            if (r <= cur) continue;

            ans += Math.min(l, r) - cur;
        }
        return ans;
    }
}
```

- 时间复杂度：预处理出两个最大值数组，复杂度为 $O(n)$；计算每根柱子可接的雨水量，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：使用了数组存储两侧最大值。复杂度为 $O(n)$

单调栈解法

前面我们讲到，优化思路将问题转化为：**给定一个数组，如何求得任意位置的左半边的最大值和右半边的最大值。**

但仔细一想，其实我们并不需要找两侧最大值，只需要找到两侧最近的比当前位置高的柱子就行

了。

针对这一类找最近值的问题，有一个通用解法：**单调栈**。

单调栈其实就是在栈的基础上，维持一个栈内元素单调。

在这道题，由于需要找某个位置两侧比其高的柱子（只有两侧有比当前位置高的柱子，当前位置才能接下雨水），我们可以维持栈内元素的单调递减。

PS. 找某侧最近一个比其大的值，使用单调栈维持栈内元素递减；找某侧最近一个比其小的值，使用单调栈维持栈内元素递增 ...

当某个位置的元素弹出栈时，例如位置 `a` ，我们自然可以得到 `a` 位置两侧比 `a` 高的柱子：

- 一个是导致 `a` 位置元素弹出的柱子（ `a` 右侧比 `a` 高的柱子）
- 一个是 `a` 弹栈后的栈顶元素（ `a` 左侧比 `a` 高的柱子）

当有了 `a` 左右两侧比 `a` 高的柱子后，便可计算 `a` 位置可接下的雨水量。

代码：

```
class Solution {
    public int trap(int[] height) {
        int n = height.length;
        int ans = 0;
        Deque<Integer> d = new ArrayDeque<>();
        for (int i = 0; i < n; i++) {
            while (!d.isEmpty() && height[i] > height[d.peekLast()]) {
                int cur = d.pollLast();

                // 如果栈内没有元素，说明当前位置左边没有比其高的柱子，跳过
                if (d.isEmpty()) continue;

                // 左右位置，并有左右位置得出「宽度」和「高度」
                int l = d.peekLast(), r = i;
                int w = r - l + 1 - 2;
                int h = Math.min(height[l], height[r]) - height[cur];
                ans += w * h;
            }
            d.addLast(i);
        }
        return ans;
    }
}
```

- 时间复杂度：每个元素最多进栈和出栈一次。复杂度为 $O(n)$
- 空间复杂度：栈最多存储 n 个元素。复杂度为 $O(n)$

面积差值解法

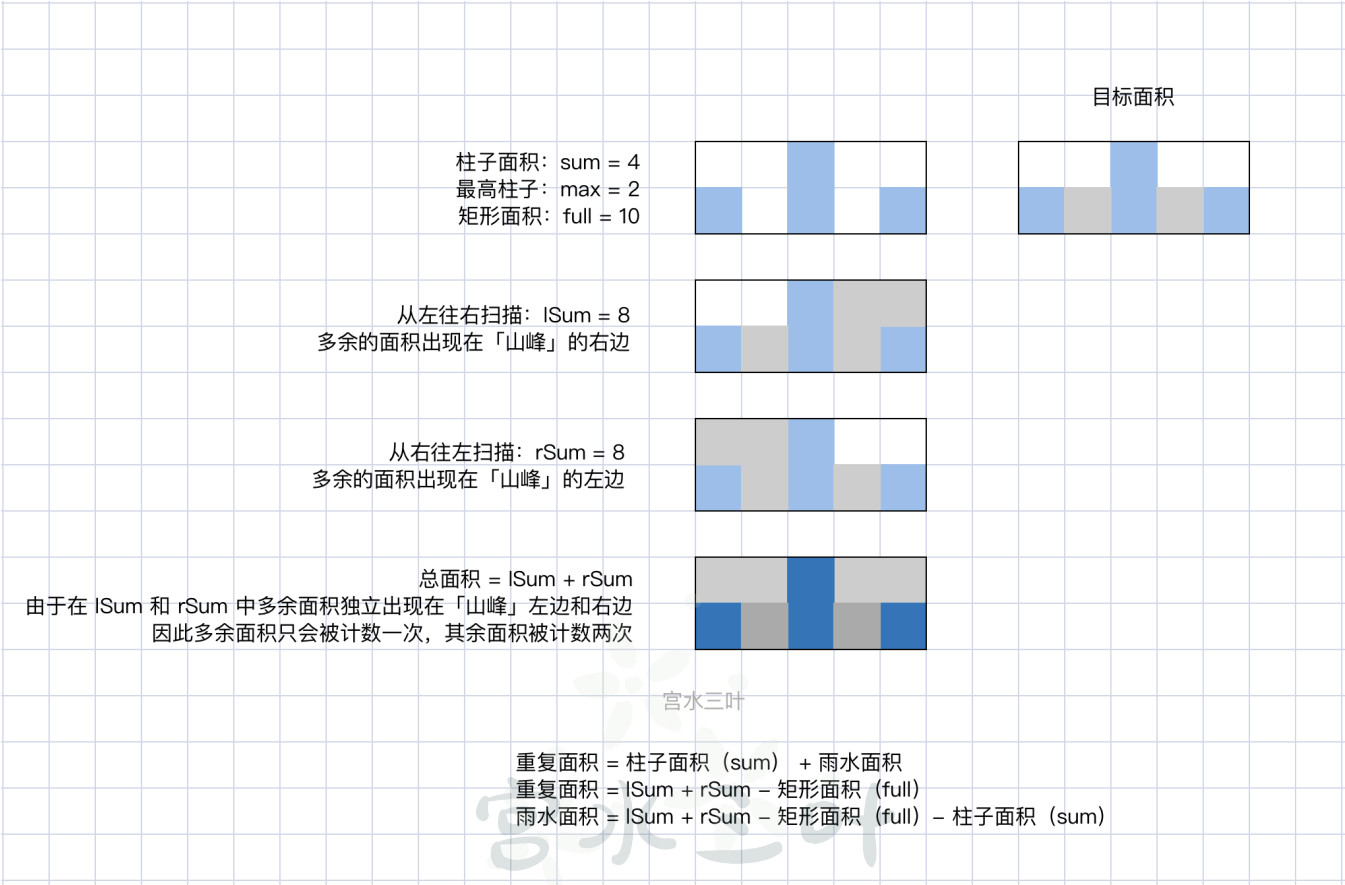
事实上，我们还能利用「面积差值」来进行求解。

我们先统计出「柱子面积」sum 和「以柱子个数为宽、最高柱子高度为高的矩形面积」$full$。

然后分别「从左往右」和「从右往左」计算一次最大高度覆盖面积 $lSum$ 和 $rSum$。

显然会出现重复面积，并且重复面积只会独立地出现在「山峰」的左边和右边。

利用此特性，我们可以通过简单的等式关系求解出「雨水面积」：



代码：

```Java
class Solution {
    public int trap(int[] height) {
        int n = height.length;

        int sum = 0, max = 0;
        for (int i = 0; i < n; i++) {
            int cur = height[i];
            sum += cur;
            max = Math.max(max, cur);
        }
        int full = max * n;

        int lSum = 0, lMax = 0;
        for (int i = 0; i < n; i++) {
            lMax = Math.max(lMax, height[i]);
            lSum += lMax;
        }

        int rSum = 0, rMax = 0;
        for (int i = n - 1; i >= 0; i--) {
            rMax = Math.max(rMax, height[i]);
            rSum += rMax;
        }

        return lSum + rSum - full - sum;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(1)$

> **🍭 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 **503. 下一个更大元素 II**，难度为 **中等**。

Tag：「单调栈」

给定一个循环数组（最后一个元素的下一个元素是数组的第一个元素），输出每个元素的下一个更大元素。数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的

数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1。

示例 1:

```
输入: [1,2,1]
输出: [2,-1,2]
解释: 第一个 1 的下一个更大的数是 2；
数字 2 找不到下一个更大的数；
第二个 1 的下一个最大的数需要循环搜索，结果也是 2。
```

单调栈解法

对于「找最近一个比当前值大/小」的问题，都可以使用单调栈来解决。

单调栈就是在栈的基础上维护一个栈内元素单调。

在理解单调栈之前，我们先回想一下「朴素解法」是如何解决这个问题的。

对于每个数而言，我们需要遍历其右边的数，直到找到比自身大的数，这是一个 $O(n^2)$ 的做法。

之所以是 $O(n^2)$，是因为每次找下一个最大值，我们是通过「主动」遍历来实现的。

而如果使用的是单调栈的话，可以做到 $O(n)$ 的复杂度，我们**将当前还没得到答案的下标暂存于栈内，从而实现「被动」更新答案。**

也就是说，栈内存放的永远是还没更新答案的下标。

具体的做法是：

每次将当前遍历到的下标存入栈内，**将当前下标存入栈内前，检查一下当前值是否能够作为栈内位置的答案（即成为栈内位置的「下一个更大的元素」），如果可以，则将栈内下标弹出。**

如此一来，我们便实现了「被动」更新答案，同时由于我们的弹栈和出栈逻辑，决定了我们**整个过程中栈内元素单调**。

还有一些编码细节，由于我们要找每一个元素的下一个更大的值，因此我们需要对原数组遍历两次，对遍历下标进行取余转换。

以及因为栈内存放的是还没更新答案的下标，可能会有位置会一直留在栈内（最大值的位置），因此我们要在处理前预设答案为 -1。而从实现那些没有下一个更大元素（不出栈）的位置的答案是 -1。

代码：

```
class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] ans = new int[n];
        Arrays.fill(ans, -1);
        Deque<Integer> d = new ArrayDeque<>();
        for (int i = 0; i < n * 2; i++) {
            while (!d.isEmpty() && nums[i % n] > nums[d.peekLast()]) {
                int u = d.pollLast();
                ans[u] = nums[i % n];
            }
            d.addLast(i % n);
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

卡常小技巧

本题不需要用到这个技巧，但是还是介绍一下，可作为拓展。

我们可以使用静态数组来模拟栈，这样我们的代码将会更快一点：

```
class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] ans = new int[n];
        Arrays.fill(ans, -1);
        // 使用数组模拟栈，hh 代表栈底，tt 代表栈顶
        int[] d = new int[n * 2];
        int hh = 0, tt = -1;
        for (int i = 0; i < n * 2; i++) {
            while (hh <= tt && nums[i % n] > nums[d[tt]]) {
                int u = d[tt--];
                ans[u] = nums[i % n];
            }
            d[++tt] = i % n;
        }
        return ans;
    }
}
```

总结

要从逻辑上去理解为什么能用「单调栈」解决问题：

1. 我们希望将 $O(n^2)$ 算法优化为 $O(n)$ 算法，因此需要将「主动」获取答案转换为「被动」更新
2. 我们需要使用数据结构保持那些「尚未更新」的位置下标，由于题目要求的是找「下一个更大的元素」，因此使用栈来保存
3. 「被动」更新答案的逻辑导致了我们栈内元素单调

> **🍭 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 **456. 132 模式** ，难度为 **中等**。

Tag：「单调栈」

给你一个整数数组 nums ，数组中共有 n 个整数。132 模式的子序列 由三个整数 nums[i]、nums[j] 和 nums[k] 组成，并同时满足：i < j < k 和 nums[i] < nums[k] < nums[j] 。

如果 nums 中存在 132 模式的子序列 ，返回 true ；否则，返回 false 。

进阶：很容易想到时间复杂度为 $O(n^2)$ 的解决方案，你可以设计一个时间复杂度为 $O(nlogn)$ 或$O(n)$ 的解决方案吗？

示例 1：

```
输入：nums = [1,2,3,4]
输出：false
解释：序列中不存在 132 模式的子序列。
```

示例 2：

```
输入：nums = [3,1,4,2]
输出：true
解释：序列中有 1 个 132 模式的子序列： [1, 4, 2] 。
```

示例 3：

```
输入：nums = [-1,3,2,0]
输出：true
解释：序列中有 3 个 132 模式的的子序列：[-1, 3, 2]、[-1, 3, 0] 和 [-1, 2, 0] 。
```

提示：

- n == nums.length
- 1 <= n <= 10^4
- -10^9 <= nums[i] <= 10^9

基本思路

朴素的做法是分别对三个数进行枚举，这样的做法是 $O(n^3)$ 的，数据范围是 10^4，稳稳超时。

事实上，这样的数据范围甚至不足以我们枚举其中两个数，然后优化找第三个数的 $O(n^2)$ 做法。

这时候根据数据范围会联想到树状数组，使用树状数组的复杂度是 $O(n \log n)$ 的，可以过。但是代码量会较多一点，还需要理解离散化等前置知识。题解也不太好写。

因此，我们可以从 132 的大小特性去分析，如果在确定一个数之后，如何快速找到另外两个数（我们使用 `ijk` 来代指 132 结构）：

1. 枚举 `i`：由于 `i` 是 132 结构中最小的数，那么相当于我们要从 i 后面，找到一个对数 `(j,k)`，使得 `(j,k)` 都满足比 `i` 大，同时 `j` 和 `k` 之间存在 `j > k` 的关系。由于我们的遍历是单向的，因此我们可以将问题转化为找 `k`，首先 `k` 需要比 `i` 大，同时在 `[i, k]` 之间存在比 `k` 大的数即可。
2. 枚举 `j`：由于 `j` 是 132 结构里最大的数，因此我们需要在 `j` 的右边中比 `j` 小的「最大」的数，在 `j` 的左边找比 `j` 小的「最小」的数。这很容易联想到单调栈，但是朴素的单调栈是帮助我们找到左边或者右边「最近」的数，无法直接满足我们「最大」和「最小」的要求，需要引入额外逻辑。
3. 枚举 `k`：由于 `k` 是 132 结构中的中间值，这里的分析逻辑和「枚举 i」类似，因为遍历是单向的，我们需要找到 `k` 左边的 `i`，同时确保 `[i,k]` 之间存在比 `i` 和 `k` 大的数字。

以上三种分析方法都是可行的，但「枚举 i」的做法是最简单的。

因为如果存在 `(j,k)` 满足要求的话，我们只需要找到一个最大的满足条件的 `k`，通过与 `i` 的比较即可。

也许你还不理解是什么意思。没关系，我们一边证明一边说。

过程 & 证明

先说处理过程吧，我们从后往前做，维护一个「单调递减」的栈，同时使用 `k` 记录所有出栈元素的最大值（ `k` 代表满足 132 结构中的 2）。

那么当我们遍历到 `i`，只要满足发现满足 `nums[i] < k`，说明我们找到了符合条件的 `i j k`。

举个🌰，对于样例数据 `[3, 1, 4, 2]`，我们知道满足 132 结构的子序列是 `[1, 4, 2]`，其处理逻辑是（遍历从后往前）：

1. 枚举到 2：栈内元素为 [2]，`k` = INF
2. 枚举到 4：不满足「单调递减」，2 出栈更新 `k`，4 入栈。栈内元素为 [4]，`k` = 2
3. 枚举到 1：满足 `nums[i] < k`，说明对于 `i` 而言，后面有一个比其大的元素（满足 `i < k` 的条件），同时这个 `k` 的来源又是因为维护「单调递减」而弹出导致被更新的（满足 `i` 和 `k` 之间，有比 `k` 要大的元素）。因此我们找到了满足 132 结构的组合。

这样做的本质是：我们通过维护「单调递减」来确保已经找到了有效的 `(j,k)`。换句话说如果 `k` 有值的话，那么必然是因为有 `j > k`，导致的有值。也就是 132 结构中，我们找到了 32，剩下的 `i`（也就是 132 结构中的 1）则是通过遍历过程中与 `k` 的比较来找到。这样做的复杂度是 $O(n)$ 的，比树状数组还要快。

从过程上分析，是没有问题的。

搞清楚了处理过程，证明也变得十分简单。

我们不失一般性的考虑任意数组 `nums`，假如真实存在 `ijk` 符合 132 的结构（这里的 `ijk` 特指所有满足 132 结构要求的组合中 `k` 最大的那个组合）。

由于我们的比较逻辑只针对 `i` 和 `k`，而 `i` 是从后往前的处理的，必然会被遍历到；漏掉 `ijk` 的情况只能是：在遍历到 `i` 的时候，我们没有将 `k` 更新到变量中：

1. 这时候变量的值要比真实情况下的 `k` 要小，说明 `k` 还在栈中，而遍历位置已经到达了 `i`，说明 `j` 和 `k` 同时在栈中，与「单调递减」的性质冲突。
2. 这时候变量的值要比真实情况下的 `k` 要大，说明在 `k` 出栈之后，有比 `k` 更大的数值出栈了（同时必然有比变量更大的值在栈中），这时候要么与我们假设 `ijk` 是 `k` 最大的组合冲突；要么与我们遍历到的位置为 `i` 冲突。

综上，由于「单调递减」的性质，我们至少能找到「遍历过程中」所有符合条件的 `ijk` 中 `k` 最大的那个组合。

单调栈

代码：

```
class Solution {
    public boolean find132pattern(int[] nums) {
        int n = nums.length;
        Deque<Integer> d = new ArrayDeque<>();
        int k = Integer.MIN_VALUE;
        for (int i = n - 1; i >= 0; i--) {
            if (nums[i] < k) return true;
            while (!d.isEmpty() && d.peekLast() < nums[i]) {
                // 事实上，k 的变化也具有单调性，直接使用 k = pollLast() 也是可以的
                k = Math.max(k, d.pollLast());
            }
            d.addLast(nums[i]);
        }
        return false;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

💡**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「单调栈」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：