解法三: 动态规划

我们注意到，解法二中。对于每一列，我们求它左边最高的墙和右边最高的墙，都是重新遍历一遍所有高度，这里我们可以优化一下。

首先用两个数组，max\_left [i] 代表第 i 列左边最高的墙的高度，max\_right[i] 代表第 i 列右边最高的墙的高度。（一定要注意下，第 i 列左（右）边最高的墙，是不包括自身的，和 leetcode 上边的讲的有些不同）

对于 max\_left我们其实可以这样求。

max\_left [i] = Max(max\_left [i-1],height[i-1])。它前边的墙的左边的最高高度和它前边的墙的高度选一个较大的，就是当前列左边最高的墙了。

对于 max\_right我们可以这样求。

max\_right[i] = Max(max\_right[i+1],height[i+1]) 。它后边的墙的右边的最高高度和它后边的墙的高度选一个较大的，就是当前列右边最高的墙了。

这样，我们再利用解法二的算法，就不用在 for 循环里每次重新遍历一次求 max\_left 和 max\_right 了。

Java

public int trap(int[] height) {

int sum = 0;

int[] max\_left = new int[height.length];

int[] max\_right = new int[height.length];

for (int i = 1; i < height.length - 1; i++) {

max\_left[i] = Math.max(max\_left[i - 1], height[i - 1]);

}

for (int i = height.length - 2; i >= 0; i--) {

max\_right[i] = Math.max(max\_right[i + 1], height[i + 1]);

}

for (int i = 1; i < height.length - 1; i++) {

int min = Math.min(max\_left[i], max\_right[i]);

if (min > height[i]) {

sum = sum + (min - height[i]);

}

}

return sum;

}

时间复杂度：O(n)O(n)。

空间复杂度：O(n)O(n)，用来保存每一列左边最高的墙和右边最高的墙。

解法四：双指针

动态规划中，我们常常可以对空间复杂度进行进一步的优化。

例如这道题中，可以看到，max\_left [ i ] 和 max\_right [ i ] 数组中的元素我们其实只用一次，然后就再也不会用到了。所以我们可以不用数组，只用一个元素就行了。我们先改造下 max\_left。

Java

public int trap(int[] height) {

int sum = 0;

int max\_left = 0;

int[] max\_right = new int[height.length];

for (int i = height.length - 2; i >= 0; i--) {

max\_right[i] = Math.max(max\_right[i + 1], height[i + 1]);

}

for (int i = 1; i < height.length - 1; i++) {

max\_left = Math.max(max\_left, height[i - 1]);

int min = Math.min(max\_left, max\_right[i]);

if (min > height[i]) {

sum = sum + (min - height[i]);

}

}

return sum;

}

我们成功将 max\_left 数组去掉了。但是会发现我们不能同时把 max\_right 的数组去掉，因为最后的 for 循环是从左到右遍历的，而 max\_right 的更新是从右向左的。

所以这里要用到两个指针，left 和 right，从两个方向去遍历。

那么什么时候从左到右，什么时候从右到左呢？根据下边的代码的更新规则，我们可以知道

Java

max\_left = Math.max(max\_left, height[i - 1]);

height [ left - 1] 是可能成为 max\_left 的变量， 同理，height [ right + 1 ] 是可能成为 right\_max 的变量。

只要保证 height [ left - 1 ] < height [ right + 1 ] ，那么 max\_left 就一定小于 max\_right。

因为 max\_left 是由 height [ left - 1] 更新过来的，而 height [ left - 1 ] 是小于 height [ right + 1] 的，而 height [ right + 1 ] 会更新 max\_right，所以间接的得出 max\_left 一定小于 max\_right。

反之，我们就从右到左更。

Java

public int trap(int[] height) {

int sum = 0;

int max\_left = 0;

int max\_right = 0;

int left = 1;

int right = height.length - 2; // 加右指针进去

for (int i = 1; i < height.length - 1; i++) {

//从左到右更

if (height[left - 1] < height[right + 1]) {

max\_left = Math.max(max\_left, height[left - 1]);

int min = max\_left;

if (min > height[left]) {

sum = sum + (min - height[left]);

}

left++;

//从右到左更

} else {

max\_right = Math.max(max\_right, height[right + 1]);

int min = max\_right;

if (min > height[right]) {

sum = sum + (min - height[right]);

}

right--;

}

}

return sum;

}

时间复杂度： O(n)O(n)。

空间复杂度： O(1)O(1)。

解法五：栈

说到栈，我们肯定会想到括号匹配了。我们仔细观察蓝色的部分，可以和括号匹配类比下。每次匹配出一对括号（找到对应的一堵墙），就计算这两堵墙中的水。

我们用栈保存每堵墙。

当遍历墙的高度的时候，如果当前高度小于栈顶的墙高度，说明这里会有积水，我们将墙的高度的下标入栈。

如果当前高度大于栈顶的墙的高度，说明之前的积水到这里停下，我们可以计算下有多少积水了。计算完，就把当前的墙继续入栈，作为新的积水的墙。

总体的原则就是，

当前高度小于等于栈顶高度，入栈，指针后移。

当前高度大于栈顶高度，出栈，计算出当前墙和栈顶的墙之间水的多少，然后计算当前的高度和新栈的高度的关系，重复第 2 步。直到当前墙的高度不大于栈顶高度或者栈空，然后把当前墙入栈，指针后移。

我们看具体的例子。

首先将 height [ 0 ] 入栈。然后 current 指向的高度大于栈顶高度，所以把栈顶 height [ 0 ] 出栈，然后栈空了，再把 height [ 1 ] 入栈。current 后移。

然后 current 指向的高度小于栈顶高度，height [ 2 ] 入栈，current 后移。

然后 current 指向的高度大于栈顶高度，栈顶 height [ 2 ] 出栈。计算 height [ 3 ] 和新的栈顶之间的水。计算完之后继续判断 current 和新的栈顶的关系。

current 指向的高度大于栈顶高度，栈顶 height [ 1 ] 出栈，栈空。所以把 height [ 3 ] 入栈。currtent 后移。

然后 current 指向的高度小于栈顶 height [ 3 ] 的高度，height [ 4 ] 入栈。current 后移。

然后 current 指向的高度小于栈顶 height [ 4 ] 的高度，height [ 5 ] 入栈。current 后移。

然后 current 指向的高度大于栈顶 height [ 5 ] 的高度，将栈顶 height [ 5 ] 出栈，然后计算 current 指向的墙和新栈顶 height [ 4 ] 之间的水。计算完之后继续判断 current 的指向和新栈顶的关系。此时 height [ 6 ] 不大于栈顶 height [ 4 ] ，所以将 height [ 6 ] 入栈。current 后移。

然后 current 指向的高度大于栈顶高度，将栈顶 height [ 6 ] 出栈。计算和新的栈顶 height [ 4 ] 组成两个边界中的水。然后判断 current 和新的栈顶 height [ 4 ] 的关系，依旧是大于，所以把 height [ 4 ] 出栈。计算 current 和 新的栈顶 height [ 3 ] 之间的水。然后判断 current 和新的栈顶 height [ 3 ] 的关系，依旧是大于，所以把 height [ 3 ] 出栈，栈空。将 current 指向的 height [ 7 ] 入栈。current 后移。

其实不停的出栈，可以看做是在找与 7 匹配的墙，也就是 3 。

而对于计算 current 指向墙和新的栈顶之间的水，根据图的关系，我们可以直接把这两个墙当做之前解法三的 max\_left 和 max\_right，然后之前弹出的栈顶当做每次遍历的 height [ i ]。水量就是 Min ( max \_ left ，max \_ right ) - height [ i ]，只不过这里需要乘上两个墙之间的距离。可以看下代码继续理解下。

Java

public int trap6(int[] height) {

int sum = 0;

Stack<Integer> stack = new Stack<>();

int current = 0;

while (current < height.length) {

//如果栈不空并且当前指向的高度大于栈顶高度就一直循环

while (!stack.empty() && height[current] > height[stack.peek()]) {

int h = height[stack.peek()]; //取出要出栈的元素

stack.pop(); //出栈

if (stack.empty()) { // 栈空就出去

break;

}

int distance = current - stack.peek() - 1; //两堵墙之前的距离。

int min = Math.min(height[stack.peek()], height[current]);

sum = sum + distance \* (min - h);

}

stack.push(current); //当前指向的墙入栈

current++; //指针后移

}

return sum;

}

时间复杂度：虽然 while 循环里套了一个 while 循环，但是考虑到每个元素最多访问两次，入栈一次和出栈一次，所以时间复杂度是 O(n)O(n)。

空间复杂度：O(n)O(n)。栈的空间。

总结:

解法二到解法三，利用动态规划，空间换时间，解法三到解法四，优化动态规划的空间，这一系列下来，让人心旷神怡。

下一篇：单调栈O(n)解决，动图预警🎶🎵

© 著作权归作者所有

103

条评论

最热

编辑

预览

评论

精选评论(3)

suing

9 个月前

比官方题解好太多了，官方那是什么鬼，压根没看懂，赞一个

129

踩

回复

CrazyTianC

8 个月前

双指针那段我有点没看明白，后来想了一下，看看我的理解能不能帮助到。 假设一开始left-1大于right+1，则之后right会一直向左移动，直到right+1大于left-1。在这段时间内right所遍历的所有点都是左侧最高点maxleft大于右侧最高点maxright的，所以只需要根据原则判断maxright与当前高度的关系就行。反之left右移，所经过的点只要判断maxleft与当前高度的关系就行。

37

踩

查看 4 条回复

回复

emacser

3 个月前

汝之秀，吾五体投地也

12

踩

查看 3 条回复

回复

评论(103)

sybil63

8 天前

https://leetcode-cn.com/problems/trapping-rain-water/solution/jie-yu-shui-by-leetcode/327718

我觉得你讲双指针那里完全没讲清楚，官方题解下的这个评论倒是讲得很好。

5

踩

回复

长弓告水

8 天前

解一逐层计算惊艳到了，怒点一赞。

5

踩

回复

suan-tou-wang-ba

8 天前

双指针

class Solution {

public int trap(int[] height) {

/\*

使用双指针（左右两边各两个指针）

我们使用一根一根柱子计算装水量的方法

left 表示左边当前遍历的柱子（即左边我们需要计算能够装多少水的柱子）

left\_max 表示 left 的左边最高的柱子长度（不包括 left）

right 表示右边当前遍历的柱子

right\_max 表示 right 的右边最高的柱子长度（不包括 right）

我们有以下几个公式：

当 left\_max < right\_max 的话，那么我们就判断 left\_max 是否比 left 高

因为根据木桶效应，一个桶装水量取决于最短的那个木板，这里也一样，柱子能否装水取决于左右两边的是否都存在比它高的柱子

因为 left\_max < right\_max 了，那么我们只需要比较 left\_max 即可

如果 left\_max > left，那么装水量就是 left\_max - left

如果 left\_max <= left，那么装水量为 0，即 left 装不了水

当 left\_max >= right\_max 的话，同理如上，比较 right\_max 和 right

？？？？ 为什么 right\_max 和 left 隔这么远我们还可以使用 right\_max 来判断？

前提：left\_max < right\_max

right\_max 虽然跟 left 离得远，但有如下两种情况：

1、left 柱子和 right\_max 柱子之间，没有比 right\_max 柱子更高的柱子了，

那么情况如下： left 能否装水取决于 left\_max 柱子是否比 left 高

|

| |

| | |

↑ ↑ ↑

l\_m l r\_m

2、left 柱子和 right\_max 柱子之间存在比 right\_max 柱子更高的柱子

那么情况如下：因为存在了比 right\_max 更高的柱子，那么我们仍然只需要判断 left\_max 是否比 left 高，因为右边已经存在比 left 高的柱子

|

| |

| | |

| | | |

↑ ↑ ↑ ↑

l\_m l mid r\_m

初始化指针：

left = 1;

right = len - 2;

left\_max = 0;

right\_max = len - 1;

（因为第一个柱子和最后一个柱子肯定不能装水，因为不作为装水柱子，而是作为左边最高柱子和右边最高柱子）

\*/

int len = height.length;

int left = 1;

int right = len - 2;

int left\_max = 0;

int right\_max = len - 1;

int res = 0;

while(left <= right){

//比较

if(height[left\_max] < height[right\_max]){

if(height[left\_max] > height[left]){

res += height[left\_max] - height[left];

}else{

left\_max = left;

}

left++;

}else{

if(height[right\_max] > height[right]){

res += height[right\_max] - height[right];

}else{

right\_max = right;

}

right--;

}

}

return res;

}

}

3

踩

查看 1 条回复

回复

76111118101马里奥

（编辑过）2 天前

对于双指针法, 一直没有看懂这里. 按照这位大佬的描述: 1.height [ left - 1] 是可能成为 max\_left 的变量， 同理，height [ right + 1 ] 是可能成为 right\_max 的变量。 那么height [ left - 1 ] < height [ right + 1 ] ， max\_left 就一定小于 max\_right

这里不对啊. 当height [ left - 1 ] < height [ right + 1 ]时, 可能max\_left 远大于height [ left - 1 ]. 导致max\_left > max\_right. 这里就矛盾了

赞

踩

查看 1 条回复

回复

夜是故乡明

1 个月前

估计大家看到双指针空间优化有点绕，我改变了下逻辑，抛砖引玉解释：

max\_left记录left左边最大值，max\_right记录right左边最大值，

若left\_max < right\_max,而right\_max小于等于left右边最大值，因此left所在雨滴以left\_max计算

同理，反之right所在雨滴以right\_max计算

def trap(self, height: List[int]) -> int:

if not height or len(height) <= 2:

return 0

left = 1

right = len(height) - 2

left\_max = height[0]

right\_max = height[len(height) - 1]

res = 0

while left <= right:

if left\_max < right\_max:

if left\_max > height[left]:

res += left\_max - height[left]

left\_max = max(left\_max, height[left])

left += 1

else:

if right\_max > height[right]:

res += right\_max - height[right]

right\_max = max(right\_max, height[right])

right -= 1

return res

5

踩

回复

啦啦啦

12 天前

栈算积水好难 崩溃

1

踩

回复

曾浩然

1 个月前

看完感觉你就是小天才！

5

踩

回复

Hungry Clarke

8 天前

通俗易懂，看一眼就明白了

赞

踩

查看 1 条回复

回复

白衣

（编辑过）15 天前

双指针怎么也看不懂，看了看官方的题解。

原来是从两边开始，分别找出两边的最高高度 max\_left 和 max\_right，积水只会受到这两个中低的高度 （min）的影响，当前列的积水就等于 min - 当前高度。

这种解法（双指针）有种二维的感觉，简直妙啊：左右两个最高高度就是按行遍历，跨度就是高度（差？），所以比解法一跨度为1的按行快，左右两个指针就是按列遍历，跨度为1。

所以两个指针相遇就算遍历完成了，楼主的 for 循环 i 没用到，且感觉两个指针会交错，但意外的能过，看不懂，真是头大。

赞

踩

查看 2 条回复

回复

mecury

（编辑过）1 个月前

关于双指针的个人理解： 当maxleft<maxright，满足了maxleft一定是当前left的左右两边高墙之中较低的一个，所以可以用maxleft-h[left]计算，再使left右移。 当maxright<maxleft，满足了maxright一定是当前right的左右两边高墙的较低的一个，所以可以用maxright-h[right]计算，再使right左移。 left和right不断向中间靠拢，直到left>=right。

2

踩

回复

auscar

6 个月前

的确是比官方题解要好太多了，通俗易懂。官解压根没看懂。

6

踩

回复

分享

举报

rulcy天天都要练算法

8 个月前

left-1<right+1，max\_left一定小于max\_right。 就我一个人在这里花了大功夫吗...属实把我整晕了...

作者：windliang

链接：https://leetcode-cn.com/problems/trapping-rain-water/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-by-w-8/

来源：力扣（LeetCode）

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。