# 题目

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

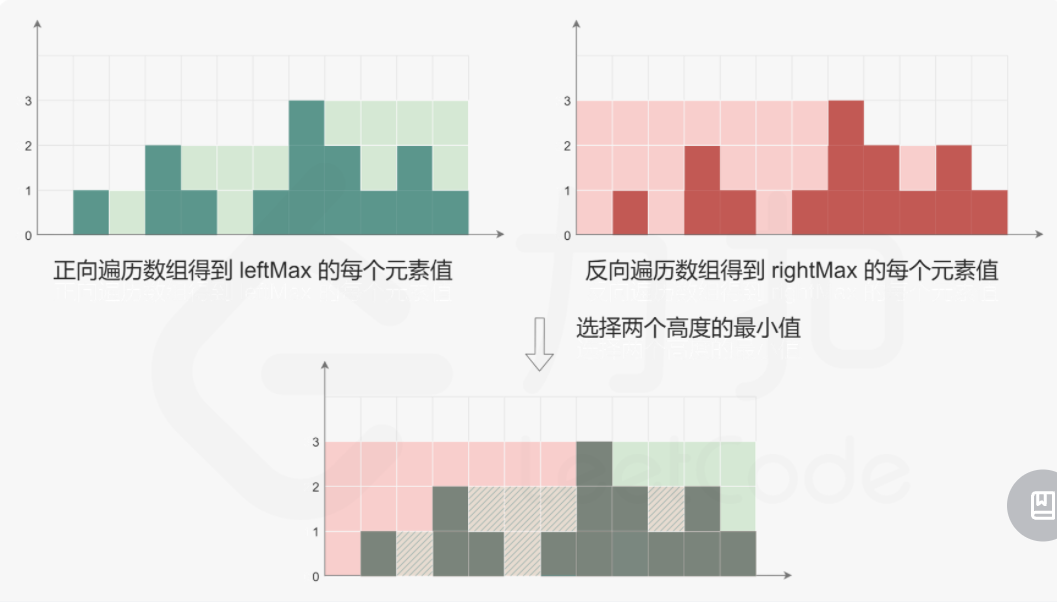
0 <= n <= 3 \* 10^4

0 <= height[i] <= 10^5

# 分析

## 方法一：动态规划

**思路：**



**代码：**

class Solution {

public:

int trap(vector<int>& height) {

int n = height.size();

if (n == 0) {

return 0;

}

vector<int> leftMax(n);

leftMax[0] = height[0];

for (int i = 1; i < n; ++i) {

leftMax[i] = max(leftMax[i - 1], height[i]);

}

vector<int> rightMax(n);

rightMax[n - 1] = height[n - 1];

for (int i = n - 2; i >= 0; --i) {

rightMax[i] = max(rightMax[i + 1], height[i]);

}

int ans = 0;

for (int i = 0; i < n; ++i) {

ans += min(leftMax[i], rightMax[i]) - height[i];

}

return ans;

}

};

**复杂度分析：**

时间复杂度：O(n)，其中n是数组height的长度。计算数组leftMax和rightMax的元素值各需要遍历数组height一次，计算能接的雨水总量还需要遍历一次。

空间复杂度：O(n)，其中n是数组height的长度。需要创建两个长度为n的数组leftMax和rightMax。

## 方法二：单调栈

思路：

使用单调栈来实现解题思路如下：

1、创建一个栈 st，用于存储柱子的索引。

2、遍历数组 height，对于每个柱子 height[i]：

- 如果栈为空或者当前柱子高度小于等于栈顶柱子高度，则将当前柱子索引入栈。

- 否则，说明当前柱子可能会形成积水，需要计算当前柱子和栈顶柱子之间的积水量。

- 弹出栈顶元素 top，计算当前柱子和栈顶柱子之间的距离 width = i - st.top() - 1。

- 计算高度差 h = min(height[i], height[st.top()]) - height[top]，即当前柱子和栈顶柱子之间的有效高度。

- 将积水量累加到结果中，ans += width \* h。

- 处理完当前柱子后，将当前柱子索引入栈。

3、遍历完整个数组后，计算完成所有可能的积水量。

具体实现如下：

class Solution {

public:

int trap(vector<int>& height) {

int n = height.size();

if (n == 0) return 0;

stack<int> st;

int ans = 0;

for (int i = 0; i < n; ++i) {

while (!st.empty() && height[i] > height[st.top()]) {

int top = st.top();

st.pop();

if (st.empty()) break;

int width = i - st.top() - 1;

int h = min(height[i], height[st.top()]) - height[top];

ans += width \* h;

}

st.push(i);

}

return ans;

}

};

这个算法的时间复杂度为 O(n)，空间复杂度为 O(n)，其中 n 是数组 `height` 的长度。

代码：

class Solution {

public:

int trap(vector<int>& height) {

int ans = 0;

stack<int> stk;

int n = height.size();

for (int i = 0; i < n; ++i) {

while (!stk.empty() && height[i] > height[stk.top()]) {

int top = stk.top();

stk.pop();

if (stk.empty()) {

break;

}

int left = stk.top();

int currWidth = i - left - 1;

int currHeight = min(height[left], height[i]) - height[top];

ans += currWidth \* currHeight;

}

stk.push(i);

}

return ans;

}

};

## 方法三：双指针

思路：

使用双指针的方法可以进一步优化空间复杂度，只需使用常数级别的额外空间。这种方法通过维护两个指针 left 和 right，分别指向数组的开头和结尾，并且维护两个变量 leftMax 和 rightMax，分别表示当前左侧和右侧的最大高度。

具体步骤如下：

1、初始化 left = 0，right = n - 1，leftMax = 0，rightMax = 0，ans = 0。

2、循环直到 left 大于等于 right：

- 如果 height[left] 小于等于 height[right]，则：

- 如果 height[left] 大于 leftMax，更新 leftMax = height[left]。

- 否则，计算当前位置的雨水量并将 ans 增加。

- 将 left 向右移动一位。

- 否则，即 height[left] 大于 height[right]，则：

- 如果height[right]大于 rightMax，更新 rightMax = height[right]。

- 否则，计算当前位置的雨水量并将 ans 增加。

- 将 right 向左移动一位。

3、返回 ans。

以下是使用双指针实现的代码：

class Solution {

public:

int trap(vector<int>& height) {

int n = height.size();

if (n == 0) return 0;

int left = 0, right = n - 1;

int leftMax = 0, rightMax = 0;

int ans = 0;

while (left < right) {

if (height[left] <= height[right]) {

if (height[left] > leftMax) {

leftMax = height[left];

} else {

ans += leftMax - height[left];

}

++left;

} else {

if (height[right] > rightMax) {

rightMax = height[right];

} else {

ans += rightMax - height[right];

}

--right;

}

}

return ans;

}

};

这个算法的时间复杂度为 O(n)，空间复杂度为 O(1)。

或：

class Solution {

public:

int trap(vector<int>& height) {

int ans = 0;

int left = 0, right = height.size() - 1;

int leftMax = 0, rightMax = 0;

while (left < right) {

leftMax = max(leftMax, height[left]);

rightMax = max(rightMax, height[right]);

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

ans += leftMax - height[left];

++left;

} else {

ans += rightMax - height[right];

--right;

}

}

return ans;

}

};

**复杂度分析：**

时间复杂度：O(n)，其中n是数组height 的长度。两个指针的移动总次数不超过n。

空间复杂度：O(1)。只需要使用常数的额外空间。

或：

可以通过维护两个数组来解决，一个数组用来记录当前位置左边的最大高度，另一个数组用来记录当前位置右边的最大高度。然后遍历每个位置，计算当前位置能接的雨水量。

class Solution {

public:

int trap(vector<int>& height) {

int n = height.size();

if (n == 0) return 0;

vector<int> leftMax(n, 0);

vector<int> rightMax(n, 0);

leftMax[0] = height[0];

for (int i = 1; i < n; ++i) {

leftMax[i] = max(leftMax[i - 1], height[i]);

}

rightMax[n - 1] = height[n - 1];

for (int i = n - 2; i >= 0; --i) {

rightMax[i] = max(rightMax[i + 1], height[i]);

}

int ans = 0;

for (int i = 0; i < n; ++i) {

ans += min(leftMax[i], rightMax[i]) - height[i];

}

return ans;

}

};