42. Trapping Rain Water

June 1, 2017 | 408.1K views

Average Rating: 4.73 (369 votes)

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it is able to trap after raining.

The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped. **Thanks Marcos** for contributing this image!

Example:

```
Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
```

Solution

Approach 1: Brute force

Intuition

Do as directed in question. For each element in the array, we find the maximum level of water it can trap after the rain, which is equal to the minimum of maximum height of bars on both the sides minus its own height.

Algorithm

- Initialize $ans = 0$
- Iterate the array from left to right:
- Initialize $\text{left\_max} = 0$ and $\text{right\_max} = 0$
- Iterate from the current element to the beginning of array updating:
  - $\text{left\_max} = \max(\text{left\_max}, \text{height}[j])$
- Iterate from the current element to the end of array updating:
  - $\text{right\_max} = \max(\text{right\_max}, \text{height}[j])$
- Add $\min(\text{left\_max}, \text{right\_max}) - \text{height}[i]$ to ans

Complexity Analysis

- Time complexity: $O(n^2)$. For each element of array, we iterate the left and right parts.
- Space complexity: $O(1)$ extra space.

Approach 2: Dynamic Programming

Intuition

In brute force, we iterate over the left and right parts again and again just to find the highest bar size upto that index. But, this could be stored. Voila, dynamic programming.

The concept is illustrated as shown:

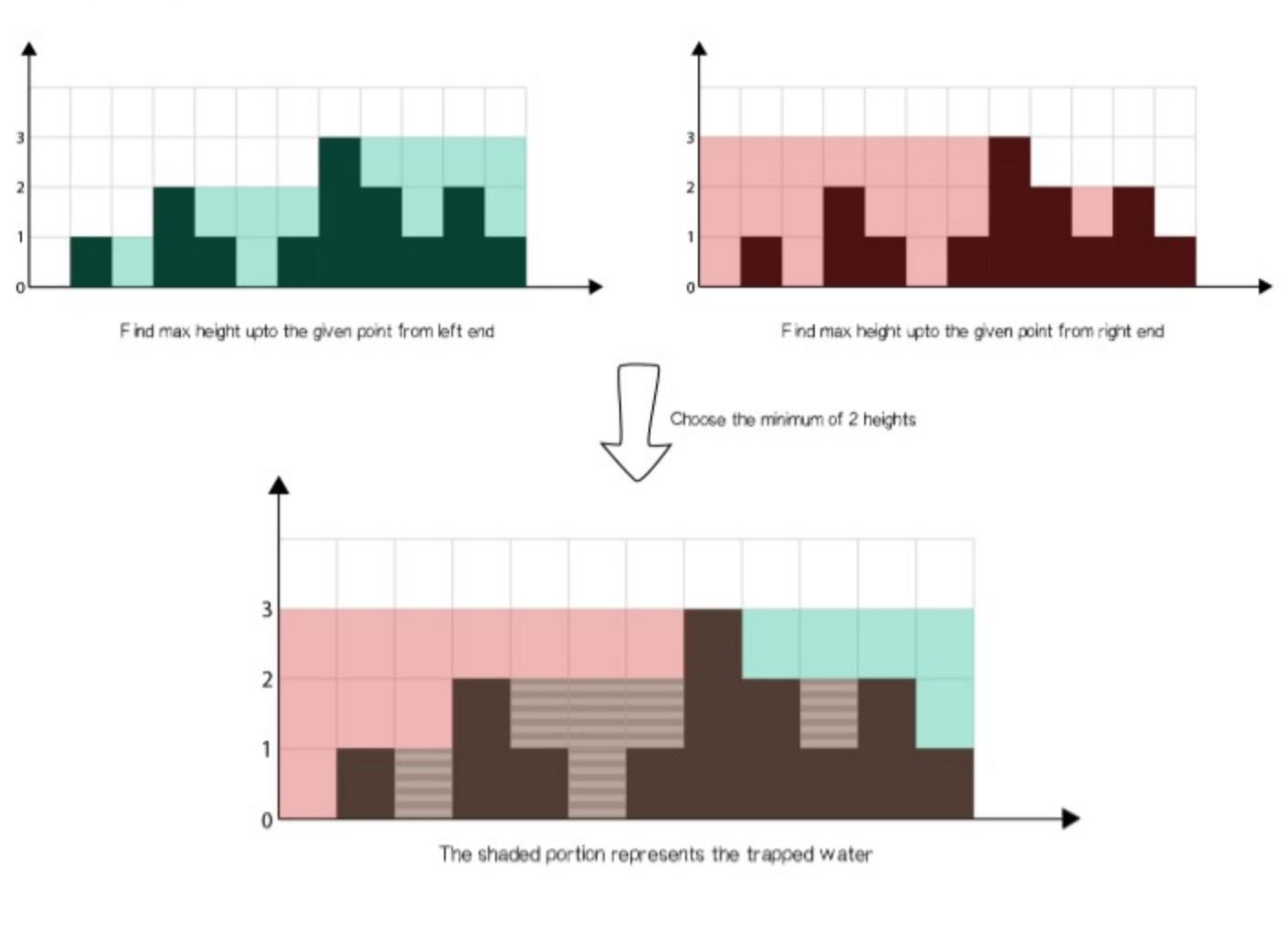

Fig: Dynamic Prgramming Approach

Algorithm

- Find maximum height of bar from the left end upto an index i in the array left_max.
- Find maximum height of bar from the right end upto an index i in the array right_max.
- Iterate over the height array and update ans:
  - Add $\min(\text{left\_max}[i], \text{right\_max}[i]) - \text{height}[i]$ to ans

C++ Copy

```cpp
int trap(vector<int>& height)
{
    if(height == null)
        return 0;
    int ans = 0;
    int size = height.size();
    vector<int> left_max(size), right_max(size);
    left_max[0] = height[0];
    for (int i = 1; i < size; i++) {
        left_max[i] = max(height[i], left_max[i - 1]);
    }
    right_max[size - 1] = height[size - 1];
    for (int i = size - 2; i >= 0; i--) {
        right_max[i] = max(height[i], right_max[i + 1]);
    }
    for (int i = 1; i < size - 1; i++) {
        ans += min(left_max[i], right_max[i]) - height[i];
    }
    return ans;
}
```

Complexity analysis

- Time complexity: $O(n)$.
  - We store the maximum heights upto a point using 2 iterations of $O(n)$ each.
  - We finally update ans using the stored values in $O(n)$.
- Space complexity: $O(n)$ extra space.
  - Additional $O(n)$ space for left_max and right_max arrays than in Approach 1.

Approach 3: Using stacks

Intuition

Instead of storing the largest bar upto an index as in Approach 2, we can use stack to keep track of the bars that are bounded by longer bars and hence, may store water. Using the stack, we can do the calculations in only one iteration.

We keep a stack and iterate over the array. We add the index of the bar to the stack if bar is smaller than or equal to the bar at top of stack, which means that the current bar is bounded by the previous bar in the stack. If we found a bar longer than that at the top, we are sure that the bar at the top of the stack is bounded by the current bar and a previous bar in the stack, hence, we can pop it and add resulting trapped water to ans.

Algorithm

- Use stack to store the indices of the bars.
- Iterate the array:
  - While stack is not empty and $\text{height}[current] > \text{height}[st.top()]$
    - It means that the stack element can be popped. Pop the top element as top.
    - Find the distance between the current element and the element at top of stack, which is to be filled. $\text{distance} = \text{current} - \text{st.top}() - 1$
    - Find the bounded height $\text{bounded\_height} = \min(\text{height}[\text{current}], \text{height}[\text{st.top}()]) - \text{height}[\text{top}]$
    - Add resulting trapped water to answer $\text{ans} \mathrel{+}= \text{distance} \times \text{bounded\_height}$
  - Push current index to top of the stack
  - Move current to the next position

C++ Copy

```cpp
int trap(vector<int>& height)
{
    int ans = 0, current = 0;
    stack<int> st;
    while (current < height.size()) {
        while (!st.empty() && height[current] > height[st.top()]) {
            int top = st.top();
            st.pop();
            if (st.empty())
                break;
            int distance = current - st.top() - 1;
            int bounded_height = min(height[current], height[st.top()]) - height[top];
            ans += distance * bounded_height;
        }
        st.push(current++);
    }
    return ans;
}
```

Complexity analysis

- Time complexity: $O(n)$.
  - Single iteration of $O(n)$ in which each bar can be touched at most twice(due to insertion and deletion from stack) and insertion and deletion from stack takes $O(1)$ time.
- Space complexity: $O(n)$. Stack can take upto $O(n)$ space in case of stairs-like or flat structure.

Approach 4: Using 2 pointers

Intuition

As in Approach 2, instead of computing the left and right parts seperately, we may think of some way to do it in one iteration. From the figure in dynamic programming approach, notice that as long as $\text{right\_max}[i] > \text{left\_max}[i]$ (from element 0 to 6), the water trapped depends upon the left_max, and similar is the case when $\text{left\_max}[i] > \text{right\_max}[i]$ (from element 8 to 11). So, we can say that if there is a larger bar at one end (say right), we are assured that the water trapped would be dependant on height of bar in current direction (from left to right). As soon as we find the bar at other end (right) is smaller, we start iterating in opposite direction (from right to left). We must maintain left_max and right_max during the iteration, but now we can do it in one iteration using 2 pointers, switching between the two.

Algorithm

- Initialize left pointer to 0 and right pointer to size-1
- While $\text{left} < \text{right}$, do:
  - If $\text{height}[\text{left}]$ is smaller than $\text{height}[\text{right}]$
    - If $\text{height}[\text{left}] \geq \text{left\_max}$, update left_max
    - Else add $\text{left\_max} - \text{height}[\text{left}]$ to ans
    - Add 1 to left.
  - Else
    - If $\text{height}[\text{right}] \geq \text{right\_max}$, update right_max
    - Else add $\text{right\_max} - \text{height}[\text{right}]$ to ans
    - Subtract 1 from right.

C++ Copy

```cpp
int trap(vector<int>& height)
{
    int left = 0, right = height.size() - 1;
    int ans = 0;
    int left_max = 0, right_max = 0;
    while (left < right) {
        if (height[left] < height[right]) {
            height[left] >= left_max ? (left_max = height[left]) : ans += (left_max - height[left]);
            ++left;
        }
        else {
            height[right] >= right_max ? (right_max = height[right]) : ans += (right_max - height[right]);
            --right;
        }
    }
    return ans;
}
```

Complexity analysis

- Time complexity: $O(n)$. Single iteration of $O(n)$.
- Space complexity: $O(1)$ extra space. Only constant space required for left, right, left_max and right_max.

Rate this article:

Previous Next

Comments: 88 Sort By

Type comment here... (Markdown is supported)

Preview Post

amanda2015w 202 August 4, 2018 3:21 AM

Below is also working but more intuitive: we're comparing leftMax and rightMax to decide which pointer to move:

```
class Solution {
    public int trap(int[] height) {
```

Read More

174 Share Reply

SHOW 12 REPLIES

kvmial 1006 October 18, 2018 8:35 AM Report

a little bit explanation about the 4th solution: Let's assume left,right,leftMax,rightMax are in positions shown in the graph below.

Read More

122 Share Reply

SHOW 8 REPLIES

narendra8 94 January 27, 2019 1:56 AM

If you don't understand the above explanation, please have a look at this video:
https://www.youtube.com/watch?v=HmBbcDiJapY
which has a nice explanation on how to solve this problem:

Read More

91 Share Reply

SHOW 4 REPLIES

zhengzhicong 294 November 13, 2018 8:02 AM

python3:

```
class Solution:
    def trap(self, height):
```

Read More

27 Share Reply

GoingMyWay 138 July 20, 2018 3:58 PM

For Approach 1: Brute force, the code is not Java code, it should be tagged with `C++`

24 Share Reply

terrible_whiteboard 626 May 19, 2020 6:24 PM Report

I made a video if anyone is having trouble understanding the solution (clickable link)
https://youtu.be/fTD6Se3ZtEo

Read More

20 Share Reply

Ark-kun 85 February 27, 2018 4:31 PM

There is an easier linear solution with constant memory: Find the global maximum. Find the last index of global maximum. Then do approach 2/4: scan right from start to last max, then scan left from end to last max. What can be easier?

20 Share Reply

SHOW 5 REPLIES

laiyinlg 139 February 5, 2019 11:21 AM Report

Simple Java two pointers:

```
class Solution {
    public int trap(int[] height) {
        int result = 0;
```

Read More

17 Share Reply

SHOW 2 REPLIES

haoyangfan 897 February 1, 2019 11:43 PM

Note that for approach 2 we can actually save 1 pass since we only need **one** array to keep track of maximum height on one side. For the other side, we can simply use a variable to keep track of maximum height so far and process it on-the-fly during the process when we calculate the volume of water in each puddle.

Read More

14 Share Reply

SHOW 1 REPLY

Pink_Strawberry 254 June 28, 2018 12:17 AM

优秀

16 Share Reply

SHOW 1 REPLY

‹ 1 2 3 4 5 6 ... 8 9 ›