LeetCode Q42 - Trapping Water

#LeetCode #monotonic_stack #code

Link to Q: https://leetcode.com/problems/trapping-rain-water/
Topics: Monotonic Stack

Question

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

Example 1:


```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) 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.
```

(finally they give a good illustration of the problem)

Workings

Initial observation

- First we notice that for water to be trapped, say in index `i`, the height at index `i-x` and `i+y` must be greater than the height at `i`
- This suggests it may be possible to solve it using a two-pointer approach.
- However this gets quite complicated when the heights get nested (see the "T" shape in the example).

Monotonic Stack

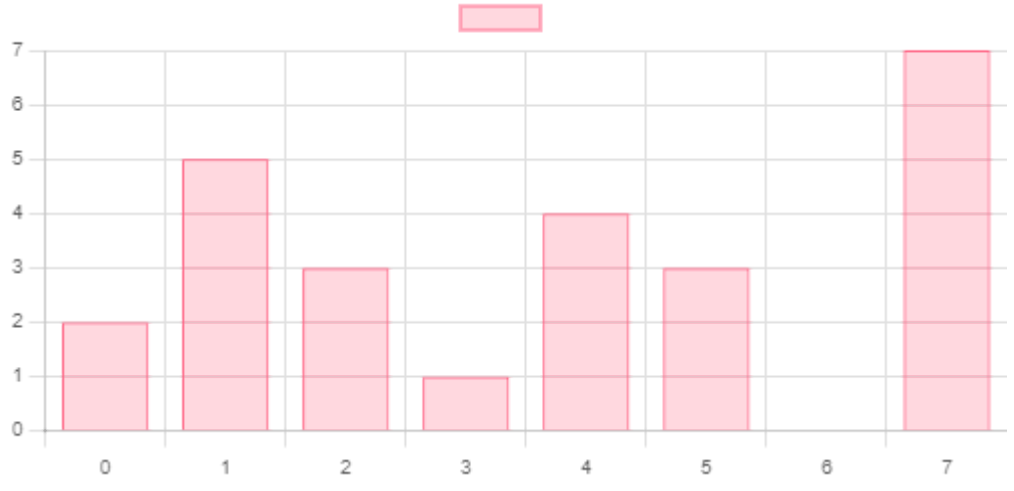
- A better solution is to use a monotonic stack
- We can observe from the example, if the heights keep increasing, no water would be trapped. So a decreasing monotonic stack might help!

The Approach:

- iterate through the list
- Add current height and index to the stack, if the height is decreasing
- If the current height is larger than the height in the last item of the stack, we pop from the stack.
  - Then we compute the water trapped between the popped height and the current height
  - We keep track of the last height that we popped, as it avoids overcounting the trapped water (illustrated below)

Example

Let's consider an example:




```
index:   0 1 2 3 4 5 6 7
heights: 2 5 3 1 4 3 0 7
```

We notate the current stack as `stack` and current water trapped as `ans`

- First we add (2,0) to the stack (2 is the height and 0 is the index)
  - `stack` : [(2,0)]
  - `ans` : 0
- Next, current height 5 is larger than 2 (last height in the stack), so we have to pop from the stack
  - We compute the trapped water by: **(current index - popped index - 1)$\times$popped height**
    - In this case: $(1-0-1)\times 2=0$ (the water trapped between index 0 and 1, there is no width so of course it is 0)
  - There is nothing else to pop, so we add the current item to the stack
  - `stack` : [(5,1)]
  - `ans` : 0
- Next, current height is 3, which is smaller than 5, so we just add it to the stack
  - `stack` : [(5,1), (3,2)]
  - `ans` : 0
- Next, current height is 1, which is smaller than 3, so we just add it to the stack
  - `stack` : [(5,1), (3,2), (1,3)]
  - `ans` : 0
- Next, current height 4 is larger than 1, so we have to pop from the stack
  - The popped height and index are 1 & 3
  - The trapped water is $(4-3-1)\times 1=0$ (the water trapped between 1 and 4)
  - `stack` : [(5,1), (3,2)]
  - `ans` : 0
- Noted the current height 4 is still larger than the last item height 3, so we continue popping
- The popped height and index are 3 & 2
- If we follow our previous formula, the trapped water would be:
  - $(4-2-1)\times 3=3$
  - But the actual water trapped should be 2 only since there was a height of 1 between the 4 and 3, which effectively reduced the trapped water
  - In order to solve this problem, we can keep track of the height of the previous popped height (initialize as 0), and modify our formula slightly:
  - **(current index - popped index - 1)$\times$(popped height - previous popped height)**
  - In this case, previous popped height is 1, so the trapped water becomes:
  - $(4-2-1)\times(3-1)=2$, which is correct
- So we append the current item after this operation since the remaining item in the stack has a height of 5 > 4:
- `stack` : [(5,1), (4,4)]
- `ans` : 2
- But is this really completed?

    - Obviously not, looking at the graph, water trapped between index 1 & 4 (height 5 & 4) should be 4 in total (like a "7" shape)
    - We have missed the part higher than height 3 between these indices
    - We can resolve the problem by simply applying the formula once again at the end of popping before actually appending our current height to it
    - Since we could not pop any further, that means, if the stack is not empty, the height of the last item must be greater than our current height, that also means, some water are trapped
  - Since we keep track of the previous popped height, we can just compute the trapped water between these two indices by:
    - **(current index - last item's index in stack - 1)$\times$(current height - previous popped height)**
    - Noted that we use current height here, as the trapped water is always the smaller height * width, and in this case, the smaller height is our current height, unlike the popped height in the above
    - So for this case, trapped water = $(4-1-1)\times(4-3)=2$ (Note that the previous popped height is 3)
  - So we are finally done with this item of 4
  - `stack` : [(5,1), (4,4)]
  - `ans` : 4
- Similarly, we keep adding the next two items as they are decreasing
  - `stack` : [(5,1), (4,4), (3,5), (0,6)]
  - `ans` : 4
- For the 7, we can see that we need to pop the whole stack
  - pop (0,6), trapped water = ... = 0
  - pop (3,5), trapped water = $(7-5-1)\times(3-0)=3$
  - pop (4,4), trapped water = $(7-4-1)\times(4-3)=2$ (prev height = 3 in this case)
  - pop (5,1), trapped water = $(7-1-1)\times(5-4)=5$
  - So the whole operation would increase our `ans` by 10

  So the final answer is 14

Code

```
class Solution:
    def trap(self, height: List[int]) -> int:
        stack=[] #monotonic decreasing stack (val, index)
        ans=0

        for index, val in enumerate(height):
            if stack and val>=stack[-1][0]:
                #pop until is monotonic
                cur=0 #bottom layer height
                while stack and stack[-1][0]<=val:
                    prevval, previndex=stack.pop()
                    ans+=(index-previndex-1)*(prevval-cur)
                    cur=prevval

            #add value one more time with previous term if stack
            if stack:
                ans+=(index-stack[-1][1]-1)*(val-cur)
        stack.append((val, index))

    return ans
```