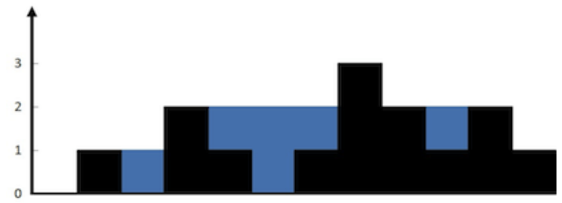
42. Trapping Rain Water (/problems/trapping-rain-water/)

June 1, 2017 | 37.1K views 42. Trapping Rain Water

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.

For example,
Given `[0,1,0,2,1,0,1,3,2,1,2,1]` , return `6` .



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!

Solution

Approach #1 Brute force [Accepted]

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

C++

Copy

```cpp
int trap(vector<int>& height)
{
    int ans = 0;
    int size = height.size();
    for (int i = 1; i < size - 1; i++) {
        int max_left = 0, max_right = 0;
        for (int j = i; j >= 0; j--) { //Search the left part for max bar size
            max_left = max(max_left, height[j]);
        }
        for (int j = i; j < size; j++) { //Search the right part for max bar size
            max_right = max(max_right, height[j]);
        }
        ans += min(max_left, max_right) - height[i];
    }
    return ans;
}
```

Complexity Analysis

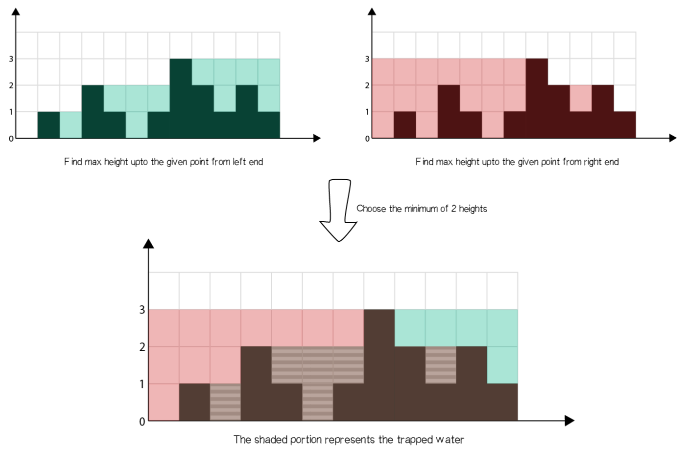
- Time complexity: $O(n^2)$. For each element of array, we iterate the left and right parts.
- Space complexity: $O(1)$ extra space.

Approach #2 Dynamic Programming [Accepted]

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:



3
2
1
0
Find max height upto the given point from left end
Find max height upto the given point from right end
Choose the minimum of 2 heights
The shaded portion represents the trapped water

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 $\min(\text{max\_left}[i], \text{max\_right}[i]) - \text{height}[i]$ 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 [Accepted]

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 $\text{bounded\_height} = \min(\text{height[current]}, \text{height[st.top()]}) - \text{height[top]}$
    - Add resulting trapped water to answer $\text{ans} += \text{distance} * \text{bounded\_height}$
  - Push current index to top of the stack
  - Move current to the next position

C++ Copy

Articles >

```
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

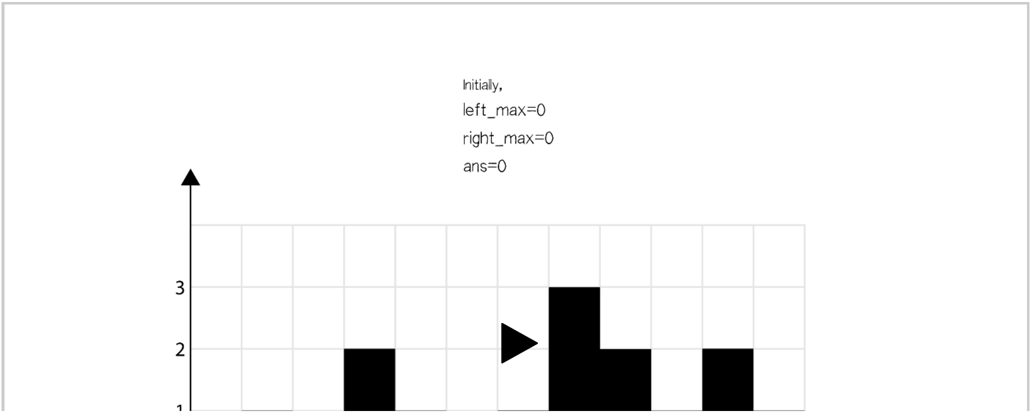
- Time complexity: $O(n)$.
  - Single iteration of $O(n)$ in which each bar can be touched at most twice(due to insertion and deletion from stack) and insertion and deletion from stack takes $O(1)$ time.
- Space complexity: $O(n)$. Stack can take upto $O(n)$ space in case of stairs-like or flat structure.

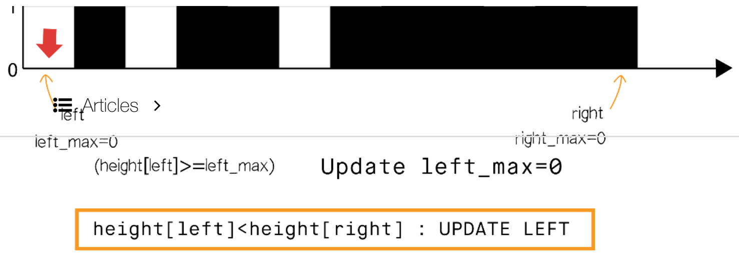
Approach #4 Using 2 pointers [Accepted]

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 $\text{height[left]} >= \text{left\_max}$, update left_max
    - Else add $\text{left\_max} - \text{height[left]}$ to ans
    - Add 1 to left.
  - Else
    - If $\text{height[right]} >= \text{right\_max}$, update right_max
    - Else add $\text{right\_max} - \text{height[right]}$ to ans
    - Subtract 1 from right.





Articles ›

1 / 11

C++ Copy

```
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.

Analysis written by @abhinavbansal0 (https://leetcode.com/abhinavbansal0).

Rate this article:
(/ratings/107/150/?return=/articles/trapping-rain-water/) (/ratings/107/150/?return=/articles/trapp

Previous (/articles/find-peak-element/)

Next (/articles/find-duplicate/)

Join the conversation

Login to Reply

K
(https://discuss.leetcode.com/user/kimgea)

kimgea commented 2 weeks ago

Kind of similar to #2, but without extra space usage and less iterations. But still running one unnecessary time on right side of max. So not as good as #4. But it might be more cache and branch prediction friendly friendly? Not sure.

Complexity:
Time = O(N) - worst 2N, best N, average 1.5N
Space = O(1)

```
class Solution {
public:
        int trap(const std::vector<int>& heights) const {

                int water = 0;

        int water_at_max = 0;
        int max_h = 0;
        auto max_idx = 0u;

        auto counter = 0u;

        for (auto h : heights)
        {
            if (max_h <= h)
            {
                max_h = h;
                max_idx = counter;
                water_at_max = water;
            }

            water += max_h - h;
            ++counter;
        }

        water = 0;
        max_h = 0;
        for (auto itr = heights.rbegin(); itr != heights.rend() - max_idx; ++itr)
        {
            if (max_h <= *itr)
                max_h = *itr;

            water += max_h - *itr;
        }

        return water_at_max + water;
        }
};
```

A
(https://discuss.leetcode.com/user/ark-kun)

Ark-kun commented 3 weeks ago

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?

V
(https://discuss.leetcode.com/user/vcshui)

vcshui commented 2 months ago

@kickasowen (https://discuss.leetcode.com/uid/5011) #11 you're right
@abhinavbansal0 (https://discuss.leetcode.com/uid/52366) subproblems in approach 2# are disjoint, it's also a divided-and-counter method

K
(https://discuss.leetcode.com/user/kickasowen)

kickasowen commented 2 months ago

#2 is not DP actually, because the problem is not divided into 2 sub-problems, but 2 *partial* problems. It's just memoization. Although memoization is often used with DP, they're different concepts. Don't be misleading.

RF commented 3 months ago

(https://discuss.leetcode.com/user/rf)

```
class Solution(object):
  # This solution has time complexity O(n**2), space complextity O(n), it has Time Limit Exce
eded error
  def trapBruteForce(self, height):
    """
    :type height: List[int]
    :rtype: int
    """
    result = 0
    for i in range(1,len(height)-1):
      l_max = max(height[:i+1])
      r_max = max(height[i:])
      h = min(l_max, r_max)
      result += h-height[i]
    return result

  # This solution has time complexity O(n), space complexity o(n)
  def trapDP(self, height):
    N = len(height)
    if N < 3: return 0

    result = 0
    l_max, r_max = [0]*N, [0]*N
    l_max[0] = height[0]

    for i in range(1, N):
      l_max[i] = max(height[i], l_max[i-1])

    r_max[N-1] = height[N-1]
    for i in range(N-2, -1, -1):
      r_max[i] = max(height[i], r_max[i+1])

    for i in range(1, N-1):
      result += min(l_max[i], r_max[i]) - height[i]

    return result

  # This solution has time complexity O(n), space complexity o(n)
  def trapStack(self, height):
    result, i = 0, 0
    st = []
    while i < len(height):
      while st and height[st[-1]] < height[i]:
        t = st.pop()
        if not st: break;
        dist = i - st[-1] -1
        h = min(height[i], height[st[-1]]) - height[t]
        result += h * dist
      st.append(i)
      i += 1
    return result

  # This solution has time complexity O(n), space complexity o(1)
  def trapTwoPointers(self, height):
    N = len(height)
    if N < 3: return 0
    result, l, r = 0, 1, N-2
    l_max = height[0]
    r_max = height[N-1]
    while (l <= r):
      if l_max <= r_max:
        l_max = max(l_max, height[l])
        result += l_max - height[l]
        l += 1
      else:
        r_max = max(r_max, height[r])
        result += r_max - height[r]
        r -= 1
    return result
```

metal4people commented 4 months ago

(https://discuss.leetcode.com/user/metal4people)

Among these solutions, stack solution is the least obvious to me.

M

meganlee commented 4 months ago

(https://discuss.leetcode.com/user/meganlee)

```
// 2 pointers, shrinking window. fix Math.max(leftMax, rightMax) side, move the other side
```

```
    public int trap4(int[] A) {
        // input validation
        if (A == null || A.length < 3) {
            return 0;
        }

        // 2 pointers, sliding window
        int sum = 0, leftMax = 0, rightMax = 0;
        for (int lo = 0, hi = A.length - 1; lo < hi; ) {
            // update leftMax and rightMax
            leftMax  = Math.max(leftMax,  A[lo]);
            rightMax = Math.max(rightMax, A[hi]);
            // fix left side, move right side
            if (leftMax > rightMax) {
                sum += rightMax - A[hi];
                hi--;
            // fix right side, shrink left side
            } else {
                sum += leftMax - A[lo];
                lo++;
            }
        }
        return sum;
```

H

hliu94 commented 5 months ago

I did approach 4, but with 2 arrays instead of 2 pointers.

(https://discuss.leetcode.com/user/hliu94)

L

l1203627122 commented 5 months ago

its so hard

(https://discuss.leetcode.com/user/l1203627122)

R

rulai.hu commented 7 months ago

(https://discuss.leetcode.com/user/rulai-hu)

Articles ›

The DP solution certainly does not need O(n) space.

1. We iterate forwards as usual, keeping the index of the last maximal height. Let this index be called peak
2. We iterate backwards until peak, keeping track of the last seen maximum, and subtracting excess rainwater

Where excess = peak - currentMaximum

```
/**
 * @param {number[]} height
 * @return {number}
 */
var trap = function(height) {
    if (height.length < 1) return 0
    peak = 0
    currMax = height[0]
    rain = 0
    for (i = 0; i < height.length; i++) {
        currHeight = height[i]

        if (currHeight < currMax) {
            rain += currMax - currHeight
        } else {
            currMax = currHeight
            peak = i
        }
    }

    currMax = height[height.length - 1]

    for (i = height.length - 1; i > peak; i--) {
        currHeight = height[i]
        if (currHeight > currMax) {
            currMax = currHeight
        }

        console.log('subtracting', (height[peak] - currMax), 'at', i)
        rain -= height[peak] - currMax
    }

    return rain
};```
```

View original thread (https://discuss.leetcode.com/topic/91027)

Load more comments...