Cracking Coding Interviews

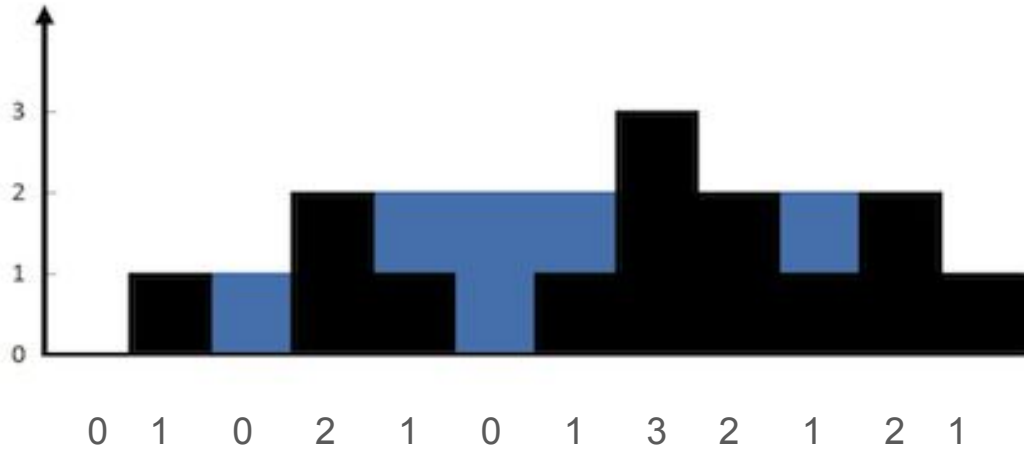
Trapping Rain Water

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

Leetcode 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 can trap** after raining.
- Input ⇒ Output
  - [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1] ⇒ 6
- Signature
  - C++: int trap(vector<int>& height)
  - Python: def trap(height) -> int
  - Java: public int trap(int[] height)



Your turn

- Can we approach it with brute-force? If so, how?

Brute-force it!

- We can use brute-force but we mainly
- For each each index
  - Compute the max trapped water for this index
  - But what is that value? It doesn't seem easy to apply brute-force
  - Do some analysis for some numbers/cases to decide

Brute-force it!

- Cell’s water is trapped IFF there are **taller walls** to both left and right sides
- The max water?
  - It depends on the **max left** wall (max value among all the left values)
  - It depends on the **max right** wall
  - The value of the cell we are evaluating relative to these walls.
  - E.g. (left = 10, current = 2, right = 15) = min(10, 15) = 10
  - E.g. (left = 10, current = 10, right = 15) = 0 as the current cell equal to one of them
  - E.g. (left = 10, current = 12, right = 15) = 0 as the current cell is bigger than 10 & 15!
  - So generally, If the current cell is >= one of its 2 sides, nothing will be 'trapped'

Brute-force it: Optimizations

- For each idx
  - Loop to compute max left of current
  - Loop to compute max left of current
- Overall O(n^2) time and O(1) memory. Can you optimize?

- We can use **prefix** left and **suffix** right arrays
- Then we simply iterate per idx and compute the trapped water!
- O(n) time and O(n) memory

Brute-force it: Optimizations

```cpp
// mx[i]: max in range {0, i}
void left_max(vector<int>& nums, vector<int>& mx) {
    mx = nums;
    for (int i = 1; i < (int) nums.size(); ++i)
        mx[i] = max(mx[i - 1], nums[i]);
}

// mx[i]: max in range {i, size-1}
void right_max(vector<int>& nums, vector<int>& mx) {
    mx = nums;
    for (int i = (int) nums.size() - 2; i >= 0; --i)
        mx[i] = max(mx[i + 1], nums[i]);
}
```

Brute-force it: Optimizations

```
int trap(vector<int> heights) { // O(n) time & memory
    int n = heights.size();
    if (n <= 2)
        return 0;

    vector<int> left_mx, right_mx;
    left_max(heights, left_mx);
    right_max(heights, right_mx);

    int traped = 0;
    for (int i = 0; i < n; i++)
        traped += min(left_mx[i], right_mx[i]) - heights[i];

    return traped;
}
```

Your turn

- Interviewer: Can you find O(n) time but O(1) space?
- Hint 1:
  - The O(n) memory comes from the prefix/suffix arrays
  - We can compute one of them in O(1), but still the 2nd remain O(n)
  - How can we do both max prefix and max suffix to be O(1)?
- Hint 2:
  - Assume max_idx represents the index of the maximum value in the array
  - How could that help you?

Solution

- We can split the array into 2 parts
  - A = from idx 0 to max_idx
  - B = from max_idx to the end of array
- The logic?
  - For A, we always know what the right_max is
  - For B, we always know what the left_max is

Solution

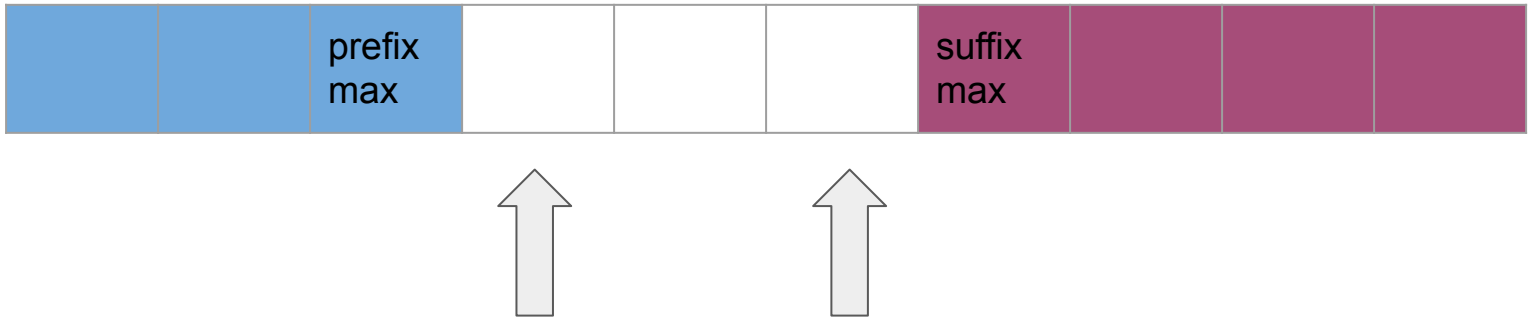
```
int ans = 0;
int max_idx = 0;    // max index block the 2 sides
for (int i = 1; i < (int) height.size(); i++)
    if (height[max_idx] < height[i])
        max_idx = i;

int left_max = 0;
for (int i = 0; i < max_idx; i++) {
    left_max = max(left_max, height[i]);
    ans += left_max - height[i];
}

int right_max = 0;
for (int i = (int) height.size() - 1; i > max_idx; i--) {
    right_max = max(right_max, height[i]);
    ans += right_max - height[i];
}
```

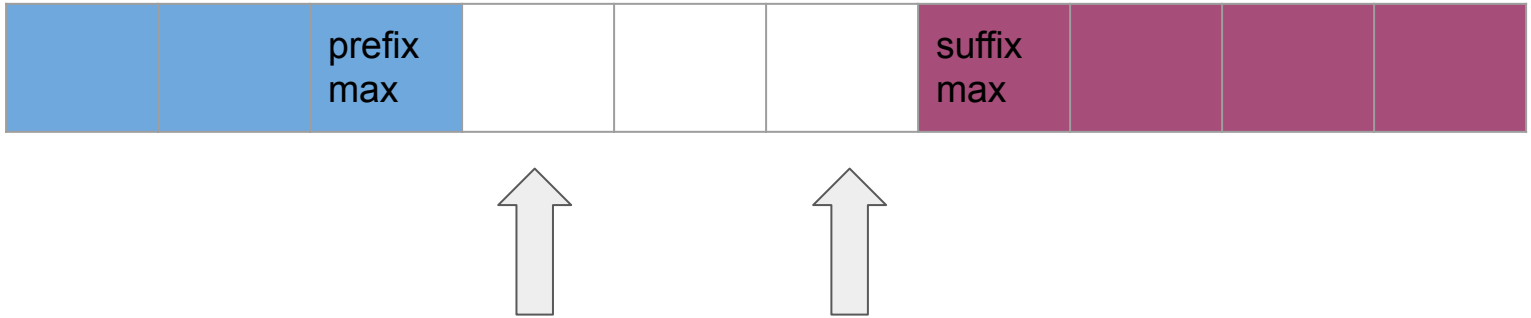
2-Pointers solution

- The editorial has O(n) 2-pointers solution
  - Left = 0, Right = sz-1 (shrink style)
  - Can you try to get it?
    - Part of the solution is to maintain the current left_max and right_max for the 2 pointers



2-Pointers solution

- The editorial has a hard O(n) 2-pointers solution
  - Left = 0, Right = sz-1 (shrink style)
  - The logic: based on the current heights and left/right max we can decide which side to process such that the unknown values in the middle doesn't affect our decision
    - if (left_max < max(right_max, height[right]))
    - The editorial uses a different logic that affects coding (< vs <=)
      - *Side note (for me): their solution always end at the max height in the array*



The logic

- Assume our indices are: left = 5 and right = 50 where N = 100
  - left_max so far = 20
  - right_max so far = 50
  - height[left] = 10 and height[right] = 70
  - For left index
    - left_max = 20
    - right_max = max(70, 50) = 70
    - Unknown values for indices from left+1 to right-1
    - 20 < 70 so min(20, 70) = 20
      - If the max(unknown values) > 70 ⇒ we don’t care as we are trapped by 20
        - The min(unknown values) is irrelevant for this problem
      - This means we can safely process the left side based on left_max
      - Otherwise, the same logic for right_max side
- Overall: interesting but challenging observation

Code!

- Tip: In 2-pointers shrinking style, always think about the condition
  - Left <= right
  - Left < right

```
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) { // with < fails at [5, 0, 5]
        if (left_max < max(right_max, height[right])) {
            left_max = max(left_max, height[left]);
            ans += left_max - height[left];
            left = left + 1;
        } else {
            right_max = max(right_max, height[right]);
            ans += right_max - height[right];
            right = right - 1;
        }
    }
    return ans;
}
```

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."