LeetCode 42

https://leetcode.com/problems/trapping-rain-water/description/

Yifeng Zeng

Description

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.

Idea Report

The primitive idea is to calculate the how much water in any position i for all positions and sum them together. To calculate the water at position i, we just need to find the larger number on its left (leftH), and the larger number on its right (rigthH). Based on the Wooden Bucket Theory we use the smaller number between leftH and rightH and substract H[i], we can get how much water it can trap at locaiton i. For each i, to get leftH and rightH we need O(n) time, so this algorithm takes O(n^2) time.

Code

```
class Solution {
    // AC
    public int trap(int[] height) {
        if (height == null || height.length < 3) {
            return 0;
        }

        int sum = 0;
```

```
        for (int i = 1; i < height.length - 1; i++) {
            int leftH = height[i];
            int leftIndex = i;
            for (int left = i - 1; left >= 0; left--) {
                if (height[left] > leftH) {
                    leftH = height[left];
                    leftIndex = left;
                }
            }
            int rightH = height[i];
            int rightIndex = i;
            for (int right = i + 1; right < height.length; right++) {
                if (height[right] > rightH) {
                    rightH = height[right];
                    rightIndex = right;
                }
            }
            int h = height[i];
            sum += Math.max(Math.min(leftH, rightH) - h, 0);
        }

        return sum;
    }
}
```

How do we speed up? We have redundancy when we look for leftH, rightH. We can pre scan the input array and store the leftH, rightH for each location i. And we scan the input array again to calculate how much water can be trapped at position i. Store leftH use O(n), store rightH use O(n), and last scan use O(n), so the algorithm taks O(3n) which is O(n) time.

Code

```
class Solution {
    // AC
    public int trap(int[] height) {
        if (height == null || height.length < 3) {
            return 0;
        }

        int len = height.length;
        int[] left = new int[len];
        int[] right = new int[len];

        left[0] = height[0];
        for (int i = 1; i < len; i++) {
            left[i] = Math.max(left[i - 1], height[i]);
        }
```

```
        right[len - 1] = height[len - 1];
        for (int i = len - 2; i >= 0; i--) {
            right[i] = Math.max(right[i + 1], height[i]);
        }

        int sum = 0;
        for (int i = 1; i < len - 1; i++) {
            sum += Math.max(0, Math.min(left[i], right[i]) - height[i]);
        }
        return sum;
    }
}
```

And we can have some optimzation to achive space O(1). We can have two pointers i start from left and j start from right. And we have "left" to store the max height from left, and "right" to store max height from right. If left < right, we can handle i first because based on Wooden Bucket Theory, the water can be trapped at locaiton i is based on smaller number (left). Otherwise we handle j, until i and j meets.

Code

```
class Solution {
    public int trap(int[] height) {
        if (height == null || height.length < 3) {
            return 0;
        }

        int left = 0;
        int right = 0;
        int i = 0;
        int j = height.length - 1;
        int sum = 0;

        while (i <= j) {
            if (left < right) {
                sum += Math.max(0, left - height[i]);
                left = Math.max(left, height[i]);
                i++;
            } else {
                sum += Math.max(0, right - height[j]);
                right = Math.max(right, height[j]);
                j--;
            }
        }
        return sum;
    }
```

```
}
```

Summary

- Store max first, use space to trade time.