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 1

Here is my idea: instead of calculating area by height*width, we can think it in a cumulative way. In other words, sum water amount of each bin(width=1). Search from left to right and maintain a max height of left and right separately, which is like a one-side wall of partial container. Fix the higher one and flow water from the lower part. For example, if current height of left is lower, we fill water in the left bin. Until left meets right, we filled the whole container.

```
class Solution {
public:
    int trap(int A[], int n) {
        int left=0; int right=n-1;
        int res=0;
        int maxleft=0, maxright=0;
        while(left<=right){
            if(A[left]<=A[right]){
                if(A[left]>=maxleft) maxleft=A[left];
                else res+=maxleft-A[left];
                left++;
            }
            else{
                if(A[right]>=maxright) maxright= A[right];
                else res+=maxright-A[right];
                right--;
            }
        }
        return res;
    }
};
```

written by mcrystal original link here

Solution 2

Keep track of the maximum height from both forward directions backward directions, call them leftmax and rightmax.

```
public int trap(int[] A){
    int a=0;
    int b=A.length-1;
    int max=0;
    int leftmax=0;
    int rightmax=0;
    while(a<=b){
        leftmax=Math.max(leftmax,A[a]);
        rightmax=Math.max(rightmax,A[b]);
        if(leftmax<rightmax){
            max+=(leftmax-A[a]);       // leftmax is smaller than rightmax, so th
e (leftmax-A[a]) water can be stored
            a++;
        }
        else{
            max+=(rightmax-A[b]);
            b--;
        }
    }
    return max;
}
```

written by yuyibestman original link here

Solution 3

Keep track of the already safe `level` and the total `water` so far. In each step, process and discard the `lower` one of the leftmost or rightmost elevation.

C

Changing the given parameters to discard the lower border. I'm quite fond of this one.

```
int trap(int* height, int n) {
    int level = 0, water = 0;
    while (n--) {
        int lower = *height < height[n] ? *height++ : height[n];
        if (lower > level) level = lower;
        water += level - lower;
    }
    return water;
}
```

Slight variation with two pointers (left and right).

```
int trap(int* height, int n) {
    int *L = height, *R = L+n-1, level = 0, water = 0;
    while (L < R) {
        int lower = *L < *R ? *L++ : *R--;
        if (lower > level) level = lower;
        water += level - lower;
    }
    return water;
}
```

C++

With left and right index.

```
int trap(vector<int>& height) {
    int l = 0, r = height.size()-1, level = 0, water = 0;
    while (l < r) {
        int lower = height[height[l] < height[r] ? l++ : r--];
        level = max(level, lower);
        water += level - lower;
    }
    return water;
}
```

With left and right iterator.

```
int trap(vector<int>& height) {
    auto l = height.begin(), r = height.end() - 1;
    int level = 0, water = 0;
    while (l != r + 1) {
        int lower = *l < *r ? *l++ : *r--;
        level = max(level, lower);
        water += level - lower;
    }
    return water;
}
```

written by StefanPochmann original link here

From Leetcoder.