Pour Water

We are given an elevation map, `heights[i]` representing the height of the terrain at that index. The width at each index is 1. After `V` units of water fall at index `K`, how much water is at each index?

Water first drops at index `K` and rests on top of the highest terrain or water at that index. Then, it flows according to the following rules:

- If the droplet would eventually fall by moving left, then move left.
- Otherwise, if the droplet would eventually fall by moving right, then move right.
- Otherwise, rise at it's current position.

Here, "eventually fall" means that the droplet will eventually be at a lower level if it moves in that direction. Also, "level" means the height of the terrain plus any water in that column.

We can assume there's infinitely high terrain on the two sides out of bounds of the array. Also, there could not be partial water being spread out evenly on more than 1 grid block - each unit of water has to be in exactly one block.

Example 1:

```
Input: heights = [2,1,1,2,1,2,2], V = 4, K = 3
Output: [2,2,2,3,2,2,2]
Explanation:
#       #
#       #
##  # ###
#########
 0123456    <- index

The first drop of water lands at index K = 3:

#       #
#   w   #
##  # ###
#########
 0123456

When moving left or right, the water can only move to the same level or a lower level
.
(By level, we mean the total height of the terrain plus any water in that column.)
Since moving left will eventually make it fall, it moves left.
(A droplet "made to fall" means go to a lower height than it was at previously.)

#       #
#       #
## w# ###
#########
 0123456

Since moving left will not make it fall, it stays in place.  The next droplet falls:

#       #
#   w   #
## w# ###
```

```
#########
 0123456

Since the new droplet moving left will eventually make it fall, it moves left.
Notice that the droplet still preferred to move left,
even though it could move right (and moving right makes it fall quicker.)

#       #
#  w    #
## w# ###
#########
 0123456

#       #
#       #
##ww# ###
#########
 0123456

After those steps, the third droplet falls.
Since moving left would not eventually make it fall, it tries to move right.
Since moving right would eventually make it fall, it moves right.

#       #
#   w   #
##ww# ###
#########
 0123456

#       #
#       #
##ww#w###
#########
 0123456

Finally, the fourth droplet falls.
Since moving left would not eventually make it fall, it tries to move right.
Since moving right would not eventually make it fall, it stays in place:

#       #
#   w   #
##ww#w###
#########
 0123456

The final answer is [2,2,2,3,2,2,2]:

    #
 #######
 #######
 0123456
```

Example 2:

Input: heights = [1,2,3,4], V = 2, K = 2
Output: [2,3,3,4]
Explanation:
The last droplet settles at index 1, since moving further left would not cause it to eventually fall to a lower height.

Example 3:

Input: heights = [3,1,3], V = 5, K = 1
Output: [4,4,4]

Note:

1. `heights` will have length in `[1, 100]` and contain integers in `[0, 99]`.
2. `V` will be in range `[0, 2000]`.
3. `K` will be in range `[0, heights.length - 1]`.

Solution 1

```
public int[] pourWater(int[] heights, int V, int K) {
        if (heights == null || heights.length == 0 || V == 0) {
            return heights;
        }
        int index;
        while (V > 0) {
            index = K;
            for (int i = K - 1; i >= 0; i--) {
                if (heights[i] > heights[index]) {
                    break;
                } else if (heights[i] < heights[index]) {
                    index = i;
                }
            }
            if (index != K) {
                heights[index]++;
                V--;
                continue;
            }
            for (int i = K + 1; i < heights.length; i++) {
                if (heights[i] > heights[index]) {
                    break;
                } else if (heights[i] < heights[index]) {
                    index = i;
                }
            }
            heights[index]++;
            V--;
        }
        return heights;
    }
```

written by satan945 original link here

Solution 2

Java

```
class Solution {
    public int[] pourWater(int[] heights, int V, int K) {
        while (V-- > 0)
            drop(heights, K, heights[K] + 1, true, true);
        return heights;
    }
   private boolean drop(int[] h, int i, int j, boolean l, boolean r) {
        if (l && i > 0            && h[i - 1] <= h[i] && drop(h, i - 1, h[i], l, fa
lse)) return true;
        if (r && i < h.length - 1 && h[i + 1] <= h[i] && drop(h, i + 1, h[i], false
, r)) return true;

        if (h[i] == j) return false;
        h[i]++;
        return true;
    }
}
```

C++

```
class Solution {
public:
    vector<int> pourWater(vector<int>& heights, int V, int K) {
        while (V-- > 0)
            drop(heights, K, heights[K] + 1, 1, 1);
        return heights;
    }
private:
    bool drop(vector<int>& h, int i, int j, bool l, bool r) {
        if (l && i > 0          && h[i - 1] <= h[i] && drop(h, i - 1, h[i], l, 0))
return true;
        if (r && i < h.size()-1 && h[i + 1] <= h[i] && drop(h, i + 1, h[i], 0, r))
return true;

        if (h[i] == j) return false;
        h[i]++;
        return true;
    }
};
```

written by alexander original link here

Solution 3

The straightforward simulation solution has runtime O(nV), where n is size of heights, and V is total water drops. In case of big V, such as V = n^2, this solution may not be optimal, and we can improve it to O(V) by preprocessing.

The preprocessing consists of three steps:

1. if flowing left, the total water "sum" can be contained;
2. if flowing right, the total water can be contained;
3. increase k grid height by 1;
   When sum > V, return back to normal simulation.

In the second more complex version, the optimal runtime could be improved to O(n^2). Let L and R to be the boundaries of water, and grid k to be the drop point, after every round of preprocessing which is O(n), we can either push at least one of L and R further or move k to left or right by 1 grid. Once L = -1, R = n, we can solve the problem mathematically. The total move of L and R is n, and the total move of k is also n. So the runtime is O(n^2).

```
class Solution {
public:
    vector<int> pourWater(vector<int>& heights, int V, int K) {
        preprocess(heights, V, K);
        int n = heights.size();
        for (int i = 0; i < V; i++) {
            int next = K; // next water drop point is lowest terrain nearest to K
            for (int d = -1; d <= 1 && next == K; d += 2) {
                for (int j = K; j+d >= 0 && j+d < n && heights[j+d] <= heights[j];
j += d) {
                    if (heights[j+d] < heights[j]) next = j+d;
                }
            }
            heights[next]++;
        }
        return heights;
    }
private:
    void preprocess(vector<int>& heights, int& v, int k) {
        // flow towards left
        if (v == 0) return;
        int l = k, r = k, sum = 0, n = heights.size();
        while (l >= 0 && heights[l] <= heights[k])
            sum += heights[k]-heights[l--];
        if (sum > v) return;
        v -= sum;
        for (int i = l+1; i < k; i++)
            heights[i] = heights[k];
        // flow towards right
        sum = 0;
        while (r < n && heights[r] <= heights[k])
            sum += heights[k]-heights[r++];
        if (sum > v) return;
        v -= sum;
        for (int i = k+1; i < r; i++)
            heights[i] = heights[k];
        // increase grid k
        if (v > 0) {
           heights[k]++;
           v--;
        }
        preprocess(heights, v, k);
    }
};
```

More efficient preprocessing. I think O(n^2).

```
class Solution {
public:
    vector<int> pourWater(vector<int>& heights, int V, int K) {
        preprocess(heights, V, K);
        return heights;
    }
private:
    void preprocess(vector<int>& heights, int& v, int k) {
        if (v == 0) return;
        int l = k, r = k, sum = 0, n = heights.size();
        while (l >= 0 && heights[l] <= heights[k])
            sum += heights[k]-heights[l--];
        if (sum > v) {
            // move k to left by 1 grid
            preprocess(heights, v, k-1);
            return;
        }
        v -= sum;
        for (int i = l+1; i < k; i++)
            heights[i] = heights[k];
        sum = 0;
        while (r < n && heights[r] <= heights[k])
            sum += heights[k]-heights[r++];
        if (sum > v) {
            // move k to right by 1 grid
            preprocess(heights, v, k+1);
            return;
        }
        v -= sum;
        for (int i = k+1; i < r; i++)
            heights[i] = heights[k];
       // solve mathematically; fill until water is 1 grid lower than boundary, then
increase grid k by 1
        int leftH = l < 0? INT_MAX: heights[l], rightH = r == n? INT_MAX: heights[r
];
        int h0 = min(leftH, rightH);
        if (h0 != INT_MAX) {
            sum = (h0-heights[k]-1)*(r-l-1)+1;
            if (sum <= v) {
                v -= sum;
                for (int i = l+1; i < r; i++)
                    heights[i] = h0-1;
                heights[k]++;
                preprocess(heights, v, k);
                return;
            }
        }
        int row = v/(r-l-1), rem = v%(r-l-1);
        v = 0;
        for (int i = l+1; i < r; i++) heights[i] += row;
        for (int i = k; i > l && rem > 0; i--, rem--)
            heights[i]++;
        for (int i = k+1; i < r && rem > 0; i++, rem--)
            heights[i]++;
    }
};
```

written by zestypanda original link here