Problem 755: Pour Water

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

You are given an elevation map represents as an integer array

heights

where

heights[i]

representing the height of the terrain at index

i

. The width at each index is

1

. You are also given two integers

volume

and

k

.

volume

units of water will fall at index

k

.

Water first drops at the 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 to its current position.

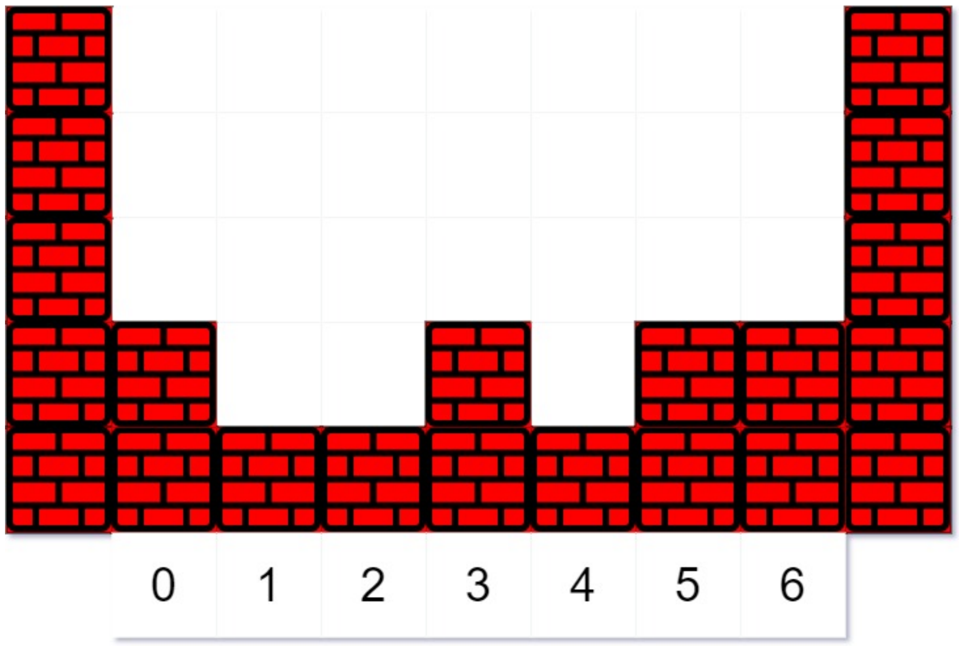
Here,

"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 is 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 one grid block, and each unit of water has to be in exactly one block.

Example 1:



Input:

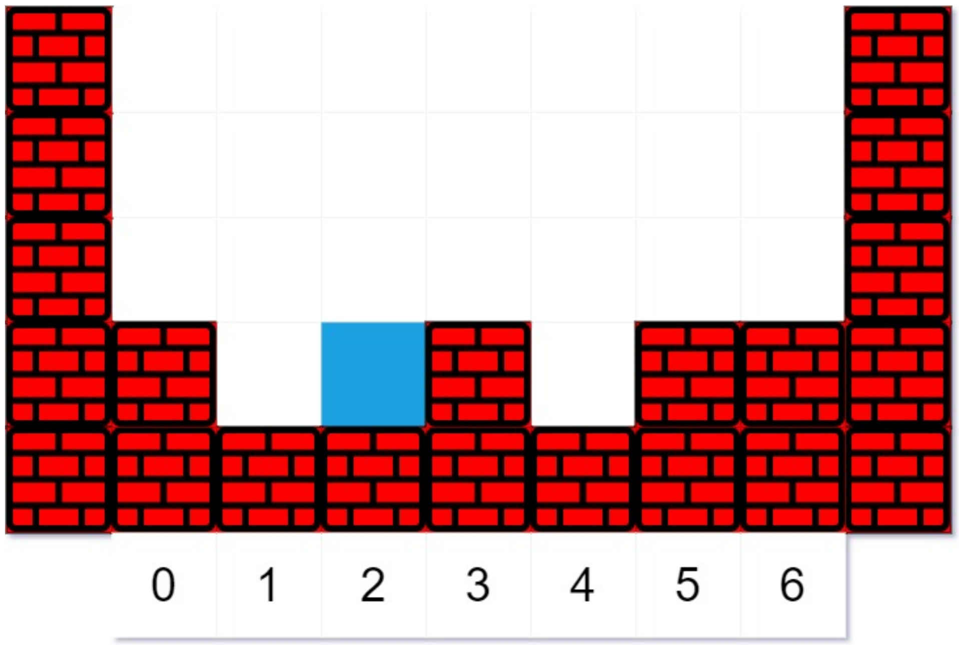
heights = [2,1,1,2,1,2,2], volume = 4, k = 3

Output:

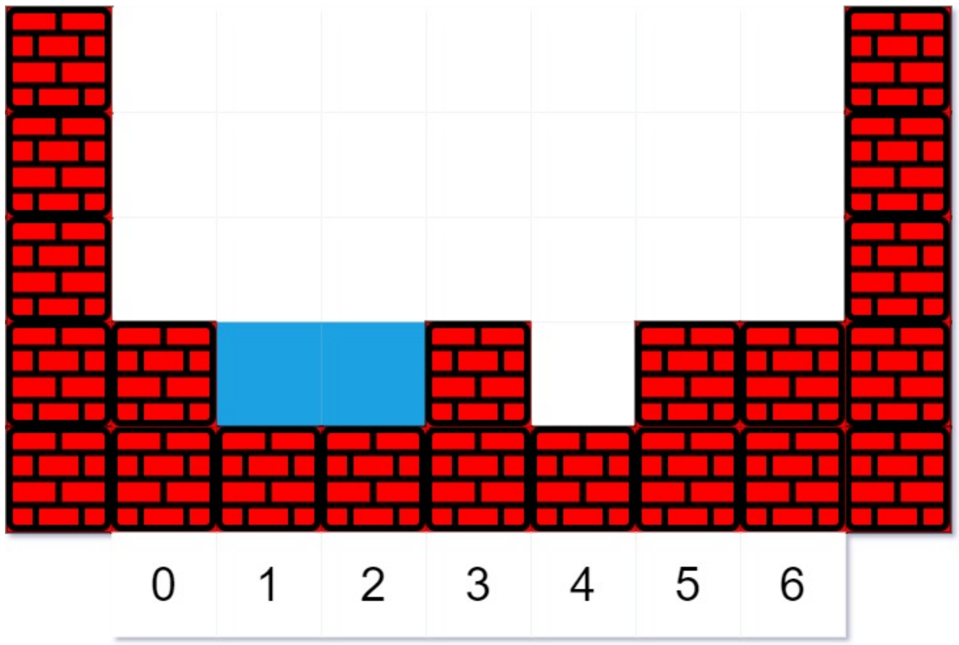
[2,2,2,3,2,2,2]

Explanation:

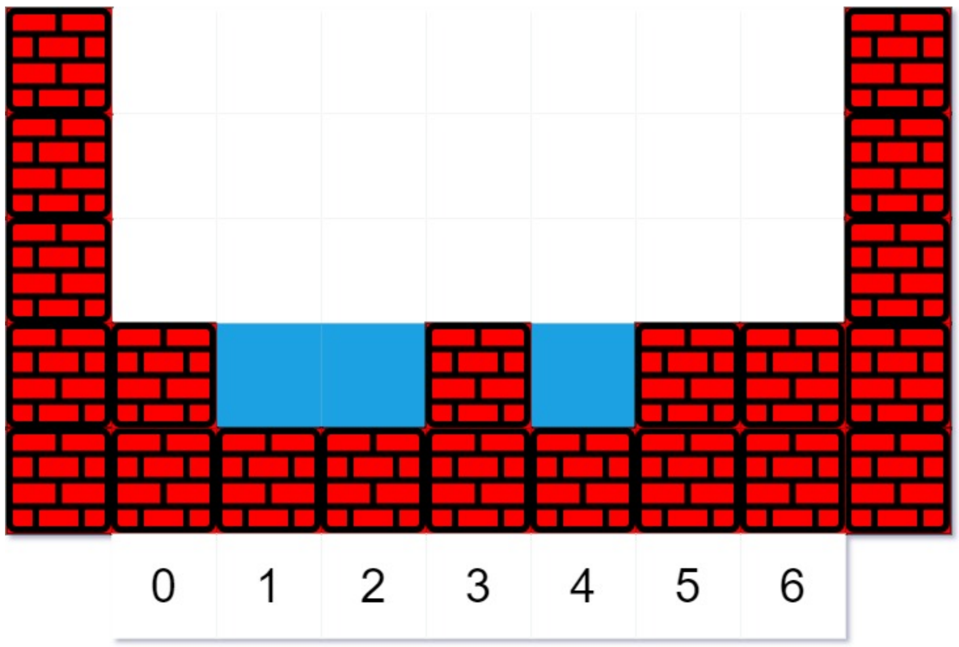
The first drop of water lands at index k = 3. When moving left or right, the water can only move to the same level or a lower level. (By level, we mean the total height of the terrain plus any water in that column.) Since moving left will eventually make it fall, it moves left. (A droplet "made to fall" means go to a lower height than it was at previously.) Since moving left will not make it fall, it stays in place.



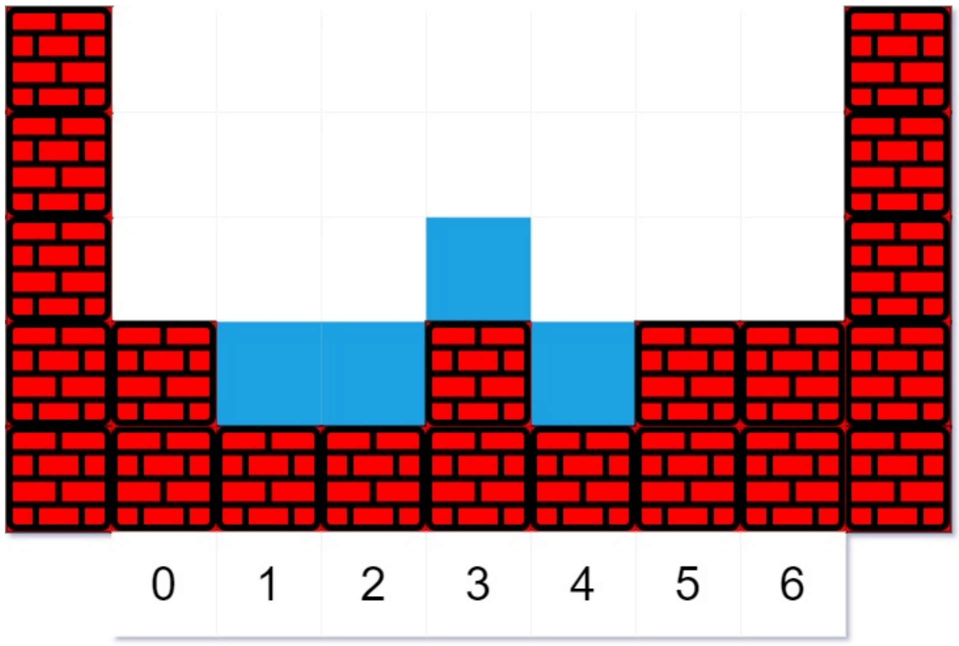
The next droplet falls at index k = 3. Since the new droplet moving left will eventually make it fall, it moves left. Notice that the droplet still preferred to move left, even though it could move right (and moving right makes it fall quicker.)



The third droplet falls at index k = 3. 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.



Finally, the fourth droplet falls at index k = 3. 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.



Example 2:

Input:

heights = [1,2,3,4], volume = 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], volume = 5, k = 1

Output:

[4,4,4]

Constraints:

1 <= heights.length <= 100

0 <= heights[i] <= 99

0 <= volume <= 2000

0 <= k < heights.length

Code Snippets

C++:

```
class Solution {
public:
vector<int> pourWater(vector<int>& heights, int volume, int k) {

}
};
```

Java:

```
class Solution {
public int[] pourWater(int[] heights, int volume, int k) {

}
}
```

Python3:

```
class Solution:
def pourWater(self, heights: List[int], volume: int, k: int) -> List[int]:
```

Python:

```
class Solution(object):
def pourWater(self, heights, volume, k):
"""
:type heights: List[int]
:type volume: int
:type k: int
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} heights
* @param {number} volume
* @param {number} k
* @return {number[]}
*/
var pourWater = function(heights, volume, k) {

};
```

TypeScript:

```
function pourWater(heights: number[], volume: number, k: number): number[] {

};
```

C#:

```
public class Solution {
public int[] PourWater(int[] heights, int volume, int k) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* pourWater(int* heights, int heightsSize, int volume, int k, int*
returnSize) {

}
```

Go:

```
func pourWater(heights []int, volume int, k int) []int {

}
```

Kotlin:

```
class Solution {
fun pourWater(heights: IntArray, volume: Int, k: Int): IntArray {

}
}
```

Swift:

```
class Solution {
func pourWater(_ heights: [Int], _ volume: Int, _ k: Int) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn pour_water(heights: Vec<i32>, volume: i32, k: i32) -> Vec<i32> {

}
}
```

Ruby:

```ruby
# @param {Integer[]} heights
# @param {Integer} volume
# @param {Integer} k
# @return {Integer[]}
def pour_water(heights, volume, k)

end
```

PHP:

```php
class Solution {

/**
* @param Integer[] $heights
* @param Integer $volume
* @param Integer $k
* @return Integer[]
*/
function pourWater($heights, $volume, $k) {

}
}
```

Dart:

```dart
class Solution {
List<int> pourWater(List<int> heights, int volume, int k) {

}
}
```

Scala:

```scala
object Solution {
def pourWater(heights: Array[Int], volume: Int, k: Int): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec pour_water(heights :: [integer], volume :: integer, k :: integer) ::
[integer]
def pour_water(heights, volume, k) do

end
end
```

Erlang:

```
-spec pour_water(Heights :: [integer()], Volume :: integer(), K :: integer())
-> [integer()].
pour_water(Heights, Volume, K) ->
.
```

Racket:

```
(define/contract (pour-water heights volume k)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof
exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Pour Water
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class Solution {
public:
vector<int> pourWater(vector<int>& heights, int volume, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Pour Water
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int[] pourWater(int[] heights, int volume, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Pour Water
Difficulty: Medium
Tags: array

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def pourWater(self, heights: List[int], volume: int, k: int) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def pourWater(self, heights, volume, k):
"""
:type heights: List[int]
:type volume: int
:type k: int
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Pour Water
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} heights
* @param {number} volume
* @param {number} k
* @return {number[]}
*/
var pourWater = function(heights, volume, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Pour Water
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function pourWater(heights: number[], volume: number, k: number): number[] {

};
```

C# Solution:

```
/*
* Problem: Pour Water
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int[] PourWater(int[] heights, int volume, int k) {

}
}
```

C Solution:

```
/*
* Problem: Pour Water
* Difficulty: Medium
* Tags: array
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* pourWater(int* heights, int heightsSize, int volume, int k, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Pour Water
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func pourWater(heights []int, volume int, k int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun pourWater(heights: IntArray, volume: Int, k: Int): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func pourWater(_ heights: [Int], _ volume: Int, _ k: Int) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Pour Water
// Difficulty: Medium
// Tags: array
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn pour_water(heights: Vec<i32>, volume: i32, k: i32) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} heights
# @param {Integer} volume
# @param {Integer} k
# @return {Integer[]}
def pour_water(heights, volume, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $heights
* @param Integer $volume
* @param Integer $k
* @return Integer[]
*/
function pourWater($heights, $volume, $k) {

}
}
```

Dart Solution:

```
class Solution {
List<int> pourWater(List<int> heights, int volume, int k) {

}
}
```

Scala Solution:

```
object Solution {
def pourWater(heights: Array[Int], volume: Int, k: Int): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec pour_water(heights :: [integer], volume :: integer, k :: integer) ::
[integer]
def pour_water(heights, volume, k) do

end
end
```

Erlang Solution:

```
-spec pour_water(Heights :: [integer()], Volume :: integer(), K :: integer())
-> [integer()].
pour_water(Heights, Volume, K) ->
.
```

Racket Solution:

```
(define/contract (pour-water heights volume k)
(-> (listof exact-integer?) exact-integer? exact-integer? (listof
exact-integer?))
)
```