

Problem 2079: Watering Plants

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

You want to water

n

plants in your garden with a watering can. The plants are arranged in a row and are labeled from

0

to

$n - 1$

from left to right where the

i

th

plant is located at

$x = i$

. There is a river at

$x = -1$

that you can refill your watering can at.

Each plant needs a specific amount of water. You will water the plants in the following way:

Water the plants in order from left to right.

After watering the current plant, if you do not have enough water to

completely

water the next plant, return to the river to fully refill the watering can.

You

cannot

refill the watering can early.

You are initially at the river (i.e.,

$x = -1$

). It takes

one step

to move

one unit

on the x-axis.

Given a

0-indexed

integer array

plants

of

n

integers, where

plants[i]

is the amount of water the

i

th

plant needs, and an integer

capacity

representing the watering can capacity, return

the

number of steps

needed to water all the plants

.

Example 1:

Input:

plants = [2,2,3,3], capacity = 5

Output:

Explanation:

Start at the river with a full watering can: - Walk to plant 0 (1 step) and water it. Watering can has 3 units of water. - Walk to plant 1 (1 step) and water it. Watering can has 1 unit of water. - Since you cannot completely water plant 2, walk back to the river to refill (2 steps). - Walk to plant 2 (3 steps) and water it. Watering can has 2 units of water. - Since you cannot completely water plant 3, walk back to the river to refill (3 steps). - Walk to plant 3 (4 steps) and water it. Steps needed = $1 + 1 + 2 + 3 + 3 + 4 = 14$.

Example 2:

Input:

plants = [1,1,1,4,2,3], capacity = 4

Output:

30

Explanation:

Start at the river with a full watering can: - Water plants 0, 1, and 2 (3 steps). Return to river (3 steps). - Water plant 3 (4 steps). Return to river (4 steps). - Water plant 4 (5 steps). Return to river (5 steps). - Water plant 5 (6 steps). Steps needed = $3 + 3 + 4 + 4 + 5 + 5 + 6 = 30$.

Example 3:

Input:

plants = [7,7,7,7,7,7,7], capacity = 8

Output:

49

Explanation:

You have to refill before watering each plant. Steps needed = $1 + 1 + 2 + 2 + 3 + 3 + 4 + 4 + 5 + 5 + 6 + 6 + 7 = 49$.

Constraints:

$n == \text{plants.length}$

$1 \leq n \leq 1000$

$1 \leq \text{plants}[i] \leq 10$

6

$\max(\text{plants}[i]) \leq \text{capacity} \leq 10$

9

Code Snippets

C++:

```
class Solution {
public:
    int wateringPlants(vector<int>& plants, int capacity) {

    }
};
```

Java:

```
class Solution {
    public int wateringPlants(int[] plants, int capacity) {

    }
}
```

Python3:

```
class Solution:
    def wateringPlants(self, plants: List[int], capacity: int) -> int:
```

Python:

```

class Solution(object):
    def wateringPlants(self, plants, capacity):
        """
        :type plants: List[int]
        :type capacity: int
        :rtype: int
        """

```

JavaScript:

```

/**
 * @param {number[]} plants
 * @param {number} capacity
 * @return {number}
 */
var wateringPlants = function(plants, capacity) {

};

```

TypeScript:

```

function wateringPlants(plants: number[], capacity: number): number {

};

```

C#:

```

public class Solution {
    public int WateringPlants(int[] plants, int capacity) {

    }
}

```

C:

```

int wateringPlants(int* plants, int plantsSize, int capacity) {

}

```

Go:

```

func wateringPlants(plants []int, capacity int) int {

```

```
}
```

Kotlin:

```
class Solution {  
    fun wateringPlants(plants: IntArray, capacity: Int): Int {  
  
    }  
}
```

Swift:

```
class Solution {  
    func wateringPlants(_ plants: [Int], _ capacity: Int) -> Int {  
  
    }  
}
```

Rust:

```
impl Solution {  
    pub fn watering_plants(plants: Vec<i32>, capacity: i32) -> i32 {  
  
    }  
}
```

Ruby:

```
# @param {Integer[]} plants  
# @param {Integer} capacity  
# @return {Integer}  
def watering_plants(plants, capacity)  
  
end
```

PHP:

```
class Solution {  
  
    /**  
     * @param Integer[] $plants  
     * @param Integer $capacity
```

```

* @return Integer
*/
function wateringPlants($plants, $capacity) {

}
}

```

Dart:

```

class Solution {
  int wateringPlants(List<int> plants, int capacity) {

  }
}

```

Scala:

```

object Solution {
  def wateringPlants(plants: Array[Int], capacity: Int): Int = {

  }
}

```

Elixir:

```

defmodule Solution do
  @spec watering_plants(plants :: [integer], capacity :: integer) :: integer
  def watering_plants(plants, capacity) do

  end
end

```

Erlang:

```

-spec watering_plants(Plants :: [integer()], Capacity :: integer()) ->
integer().
watering_plants(Plants, Capacity) ->
.

```

Racket:

```
(define/contract (watering-plants plants capacity)
  (-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
 * Problem: Watering Plants
 * 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 wateringPlants(vector<int>& plants, int capacity) {

    }
};
```

Java Solution:

```
/**
 * Problem: Watering Plants
 * 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 wateringPlants(int[] plants, int capacity) {

    }
}
```

```
}
```

Python3 Solution:

```
"""
Problem: Watering Plants
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 wateringPlants(self, plants: List[int], capacity: int) -> int:
        # TODO: Implement optimized solution
        pass
```

Python Solution:

```
class Solution(object):
    def wateringPlants(self, plants, capacity):
        """
        :type plants: List[int]
        :type capacity: int
        :rtype: int
        """
```

JavaScript Solution:

```
/**
 * Problem: Watering Plants
 * 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[]} plants
 * @param {number} capacity
 * @return {number}
 */
var wateringPlants = function(plants, capacity) {

};

```

TypeScript Solution:

```

/**
 * Problem: Watering Plants
 * 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 wateringPlants(plants: number[], capacity: number): number {

};

```

C# Solution:

```

/*
 * Problem: Watering Plants
 * 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 WateringPlants(int[] plants, int capacity) {

    }
}

```

```
}
```

C Solution:

```
/*
 * Problem: Watering Plants
 * 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
 */

int wateringPlants(int* plants, int plantsSize, int capacity) {

}
```

Go Solution:

```
// Problem: Watering Plants
// 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 wateringPlants(plants []int, capacity int) int {

}
```

Kotlin Solution:

```
class Solution {
    fun wateringPlants(plants: IntArray, capacity: Int): Int {

    }
}
```

Swift Solution:

```

class Solution {
    func wateringPlants(_ plants: [Int], _ capacity: Int) -> Int {

    }
}

```

Rust Solution:

```

// Problem: Watering Plants
// 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 watering_plants(plants: Vec<i32>, capacity: i32) -> i32 {

    }
}

```

Ruby Solution:

```

# @param {Integer[]} plants
# @param {Integer} capacity
# @return {Integer}
def watering_plants(plants, capacity)

end

```

PHP Solution:

```

class Solution {

    /**
     * @param Integer[] $plants
     * @param Integer $capacity
     * @return Integer
     */
    function wateringPlants($plants, $capacity) {

```

```
}  
}
```

Dart Solution:

```
class Solution {  
  int wateringPlants(List<int> plants, int capacity) {  
  
  }  
}
```

Scala Solution:

```
object Solution {  
  def wateringPlants(plants: Array[Int], capacity: Int): Int = {  
  
  }  
}
```

Elixir Solution:

```
defmodule Solution do  
  @spec watering_plants(plants :: [integer], capacity :: integer) :: integer  
  def watering_plants(plants, capacity) do  
  
  end  
end
```

Erlang Solution:

```
-spec watering_plants(Plants :: [integer()], Capacity :: integer()) ->  
integer().  
watering_plants(Plants, Capacity) ->  
.
```

Racket Solution:

```
(define/contract (watering-plants plants capacity)  
  (-> (listof exact-integer?) exact-integer? exact-integer?)  
  )
```

