2079. Watering Plants

Medium Array

Problem Description

In this problem, we have `n` plants arranged in a row, numbered from `0` to `n - 1`. Each plant requires a specific amount of water. We have a watering can with a finite capacity and a river located at `x = -1` where we can refill the can. The goal is to find out the minimum number of steps needed to water all the plants by following these rules:

1. We must water the plants in order from left to right.
2. If the watering can does not have enough water to fully water the next plant, we must go back to the river to refill the can to its full capacity before watering that plant.
3. We are not allowed to refill the can before it is completely empty.
4. We start at the river (at `x = -1`), and each step equates to moving one unit on the x-axis.

The problem asks us to calculate the total number of steps we must take to water all the plants when we are given an array `plants`, where `plants[i]` represents the amount of water needed by the `i`th plant, and an integer `capacity`, which is the total capacity of the watering can.

Intuition

The intuitive approach to solve this problem is by a simple simulation of the watering process, keeping track of the current position, and the amount of water left in the can.

1. Start from the river with the can filled to capacity.
2. Move towards the plants and water them in sequence until the can is depleted.
3. When the can doesn't have enough water for the next plant, calculate the number of steps to go back to the river, refill the can, and return to the current plant.
4. Each watering step and each walking step is counted to calculate the number of total steps.

Solution Approach

The solution is implemented as a function `wateringPlants` within the `Solution` class. It takes two arguments: `plants`, which is a list of integers where each integer represents the water requirement of a plant, and `capacity`, which is an integer representing the full capacity of the watering can. The function returns an integer that is the total number of steps needed to water all the plants.

Here's a step-by-step explanation of the solution's implementation:

1. We initialize a variable `ans` to store the total number of steps needed, and set it to `0`. We also create a variable `cap` to keep track of the current water level in the can and initialize it to `capacity`.
2. We use a `for` loop that goes through each plant (and its index `i`) by enumerating over `plants`. The `enumerate` function is useful here as it provides both the index and the value from the list.
3. For each plant, we check if the current water level (`cap`) is sufficient to water the plant (`x`):
   - If `cap >= x`, it means we have enough water for the current plant. We water the plant by subtracting `x` from `cap` and increment `ans` by `1` to account for the step taken to water the plant.
   - If `cap < x`, we do not have enough water and need to refill the can. Before refilling, we calculate the number of steps needed to go back to the river and return to the current plant. This is `i * 2` (double the distance from the river to the plant) plus `1` more step to water the plant. We update `ans` with these additional steps. We then reset `cap` to `capacity - x` since we refill the can and use `x` amount of water to water the current plant.
4. When the loop is completed, all plants have been watered, and `ans` contains the total number of steps required. We return `ans` as the final answer.

The algorithm's time complexity is O(n), where n is the number of plants since every plant is visited at most twice (once while moving forward and once while moving backward if a refill is needed). The space complexity is O(1) as we only use a fixed amount of additional memory (variables `ans` and `cap`).

Here is the core implementation encapsulated by the `for` loop:

```
for i, x in enumerate(plants):
    if cap >= x:
        cap -= x
        ans += 1
    else:
        cap = capacity - x
        ans += i * 2 + 1
```

This approach straightforwardly solves the problem efficiently and effectively without the need for complex data structures or patterns.

Example Walkthrough

Let's assume we have 5 plants with the following water requirements given in the array `plants = [2, 4, 5, 1, 2]` and a watering can with a capacity of `6` units of water. Using the solution approach, let's walk through the process to determine the minimum number of steps needed to water all the plants.

1. Start with the can at full capacity (6 units of water) at the river location which is at `x=-1`.
2. Move to plant 0 (1 step). The plant requires 2 units of water, and we have enough water. Water the plant (`cap = 6 - 2 = 4`, `ans = 1`).
3. Move to plant 1 (1 step). The plant requires 4 units of water, and we have enough water. Water the plant (`cap = 4 - 4 = 0`, `ans = 2`).
4. Since the watering can is now empty, move back to the river to refill the can (2 steps back, +2 steps forward to return to plant 1, total 4 steps). Refill the can to full capacity and water plant 2 which requires 5 units of water (`cap = 6 - 5 = 1`, `ans = 6` after refilling and watering).
5. Move to plant 3 (1 step). The plant requires 1 unit of water, and we have enough water. Water the plant (`cap = 1 - 1 = 0`, `ans = 7`).
6. Again, the watering can is empty, so move back to the river and refill the can (3 steps back, +3 steps forward to return to plant 3, total 6 steps). Water plant 4 which requires 2 units of water (`cap = 6 - 2 = 4`, `ans = 13` after refilling and watering).

Now all plants have been watered, and the total number of steps taken is `13`. Therefore, the `wateringPlants` function would return `13` for this example.

The steps taken for each action are summarized in the following sequence:

- Start `[can=6]`
- Water plant 0 `[can=4, steps=1]`
- Water plant 1 `[can=0, steps=2]`
- Refill at river, water plant 2 `[can=1, steps=6]`
- Water plant 3 `[can=0, steps=7]`
- Refill at river, water plant 4 `[can=4, steps=13]`
- Done

The answer to how many steps are needed to water all plants is 13 steps.

Solution Implementation

Python

```python
from typing import List  # Import the List type from the typing module.

class Solution:
    def wateringPlants(self, plants: List[int], capacity: int) -> int:
        # Initialize steps to zero and current_capacity to the input capacity.
        steps, current_capacity = 0, capacity

        # Iterate through the plants with their indices.
        for i, plant in enumerate(plants):
            # If the current_capacity is sufficient to water the plant:
            if current_capacity >= plant:
                current_capacity -= plant  # Decrease the current_capacity by plant's need.
                steps += 1  # Increment the steps by one (one step forward).
            else:
                # Refill the watering can to full capacity then water the plant.
                current_capacity = capacity - plant
                # Add steps to go back to river (i steps back) and return (i steps forward).
                # Plus one more step to water the current plant.
                steps += (i + 1) * 2  # Total steps for back and forth.

        # Return the total number of steps taken to water all plants.
        return steps

# Example usage:
# solution = Solution()
# print(solution.wateringPlants([2, 4, 5, 1, 2], 6))  # Output would be 17
```

Java

```java
class Solution {
    public int wateringPlants(int[] plants, int capacity) {
        int steps = 0; // This will hold the total number of steps taken
        int currentCapacity = capacity; // This holds the current water capacity in the can

        // Loop through all the plants
        for (int i = 0; i < plants.length; i++) {
            // If there's enough water left to water the current plant
            if (currentCapacity >= plants[i]) {
                currentCapacity -= plants[i]; // Water the plant and decrease the can's capacity
                steps++; // One step to water the plant
            } else {
                // If there isn't enough water capacity:
                // Steps to go back to the river to refill (i steps)
                // and return back to this plant (i + 1 steps)
                steps += 2 * i + 1;
                currentCapacity = capacity - plants[i]; // Refill the can minus the water needed for current plant
            }
        }
        return steps; // Return the total number of steps taken
    }
}
```

C++

```cpp
class Solution {
public:
    int wateringPlants(vector<int>& plants, int capacity) {
        int steps = 0; // Variable to store the total number of steps taken.
        int remainingCapacity = capacity; // Variable to keep track of the remaining water capacity.

        // Loop through all the plants.
        for (int i = 0; i < plants.size(); ++i) {
            // Check if there's enough water left to water the current plant.
            if (remainingCapacity >= plants[i]) {
                // If there is, water the plant: decrement remaining capacity.
                remainingCapacity -= plants[i];
                // And increment the step counter because a step is taken to water the plant.
                ++steps;
            } else {
                // If there's not enough water left, go back to the river to refill.
                // This takes 2 steps for every plant passed (back and forth), plus one to water the plant.
                steps += i * 2 + 1;
                // Refill the can to full capacity, minus the water needed for the current plant.
                remainingCapacity = capacity - plants[i];
            }
        }

        return steps; // Return the total number of steps taken.
    }
};
```

TypeScript

```typescript
function wateringPlants(plants: number[], capacity: number): number {
    // n holds the total number of plants
    const plantCount = plants.length;

    // steps represents the total number of steps taken so far
    let steps = 0;

    // currentWater represents the current water capacity in the can
    let currentWater = capacity;

    // Looping through each plant
    for (let i = 0; i < plantCount; i++) {
        // If current water is less than what the current plant needs,
        // the gardener must refill the water can
        if (currentWater < plants[i]) {
            // Steps to go back to the river (i steps), refill (1 step), and return to the plant (i steps)
            steps += i * 2 + 1;

            // Refill the can to full capacity minus the amount of water needed for the current plant
            currentWater = capacity - plants[i];
        } else {
            // If enough water is present for the current plant,
            // simply water the plant, which takes 1 step
            steps++;

            // Subtract the amount of water used for the current plant
            currentWater -= plants[i];
        }
    }

    // Return the total number of steps taken to water all plants
    return steps;
}
```


```python
from typing import List  # Import the List type from the typing module.

class Solution:
    def wateringPlants(self, plants: List[int], capacity: int) -> int:
        # Initialize steps to zero and current_capacity to the input capacity.
        steps, current_capacity = 0, capacity

        # Iterate through the plants with their indices.
        for i, plant in enumerate(plants):
            # If the current_capacity is sufficient to water the plant:
            if current_capacity >= plant:
                current_capacity -= plant  # Decrease the current_capacity by plant's need.
                steps += 1  # Increment the steps by one (one step forward).
            else:
                # Refill the watering can to full capacity then water the plant.
                current_capacity = capacity - plant
                # Add steps to go back to river (i steps back) and return (i steps forward).
                # Plus one more step to water the current plant.
                steps += (i + 1) * 2  # Total steps for back and forth.

        # Return the total number of steps taken to water all plants.
        return steps

# Example usage:
# solution = Solution()
# print(solution.wateringPlants([2, 4, 5, 1, 2], 6))  # Output would be 17
```


Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the number of plants. This is because the code iterates through each plant exactly once.

The space complexity of the code is `O(1)` since a fixed amount of extra space is used regardless of the input size. Additional variables `ans` and `cap` are used, but their use does not scale with the number of plants.