2105. Watering Plants II

Medium | Array | Two Pointers | Simulation

Leetcode Link

Problem Description

Alice and Bob are watering plants numbered from 0 to `n - 1` in a row. Each plant requires a specific amount of water. Alice starts watering from the left (plant 0) moving to the right, while Bob begins from the right (plant `n - 1`) moving to the left. Both Alice and Bob are equipped with a watering can which is initially full, and they start watering simultaneously.

The key rules for watering the plants are as follows:

1. The time to water each plant is the same, regardless of the water needed.
2. They can only water a plant if their can has enough water for the whole plant. If not, they refill their can instantly and then water the plant.
3. If Alice and Bob meet at the same plant, the one with more water will water it. If they have the same amount, Alice does it.

The goal is to find out the total number of times Alice and Bob have to refill their watering cans in order to water all the plants.

The input to the problem is an array `plants` containing the amount of water each plant needs, and two integers, `capacityA` and `capacityB`, indicating the capacity of Alice's and Bob's watering cans, respectively.

Intuition

To solve this problem, we can simulate the process of watering the plants. Alice and Bob move towards each other from opposite ends of the row. They each water the plants according to their capacities. We track the remaining water in their cans after watering each plant. If a can doesn't have enough water to fully water the current plant:

- We increment a counter representing the number of refills.
- We refill the can, subtracting the water needed for the current plant from the full capacity.

We continue this process until Alice and Bob meet at the same plant or pass by each other, meaning all plants have been watered. At the moment they reach the same plant, we compare their remaining water and make a decision based on the rule.

The intuition behind this approach is to mimic the real-world actions Alice and Bob would take, updating values and counters as necessary. By simulating the watering process step by step, we avoid missing any cases where a refill might be necessary and ensure we account for every plant.

Solution Approach

The solution adopts a two-pointer approach, with one pointer (`i`) starting from the beginning of the array (Alice's side), and another pointer (`j`) starting from the end of the array (Bob's side). These pointers represent the current position of Alice and Bob, respectively. As they move toward each other, we calculate the number of refills needed based on the rules given.

To implement the solution, the following steps are followed:

1. Initialize two variables `a` and `b` to represent the full capacities of Alice's and Bob's watering cans (`capacityA` and `capacityB`) as they'll be refilled to these values.

2. Initialize the pointer `i` to `0` and `j` to `len(plants) - 1`.

3. Initialize a counter `ans` to `0` for counting the total number of refills made by Alice and Bob.

4. Use a while loop to iterate as long as `i <= j` (meaning Alice and Bob are not past each other):

   - Check if `i == j`, which means Alice and Bob are at the same plant.
     - If so, check if their can capacity (`max(capacityA, capacityB)`) is less than the water needed for this plant (`plants[i]`). If it is, increment the `ans` counter as this will require a refill.
     - Then break the loop as all plants have been watered.
   - For Alice:
     - If the current water (`capacityA`) is less than what the current plant needs (`plants[i]`), refill Alice's can and increment the `ans` counter.
     - Otherwise, reduce Alice's current water by the amount needed for the plant.
     - Move to the next plant by incrementing `i`.
   - For Bob (similar to Alice):
     - If Bob's current water (`capacityB`) is less than what the current plant needs (`plants[j]`), refill Bob's can and increment the `ans` counter.
     - Otherwise, reduce Bob's current water by the amount needed for the plant.
     - Move to the previous plant by decrementing `j`.
5. After the while loop, return the `ans` counter which now holds the total number of refills required.

The two-pointer technique allows us to effectively simulate the action of both individuals as they meet in the middle, considering both directions simultaneously. This approach ensures that both capacities and refill requirements are checked at every step, thus finding the minimum number of refills needed to water all plants.

Example Walkthrough

Let's illustrate the solution approach using a small example:

Suppose we have an array `plants = [1, 2, 4, 2, 3]`, and `capacityA` (Alice's can capacity) is 5, and `capacityB` (Bob's can capacity) is 6.

Here is a step-by-step walkthrough:

- Initialize Alice's current water `a = capacityA = 5` and Bob's current water `b = capacityB = 6`.
- Initialize the pointers for Alice and Bob `i = 0` and `j = 4` respectively, pointing at the start and end of the `plants` array.
- Initialize the number of refills counter `ans = 0`.

Start the while loop:

- **Step 1**: Alice at `i = 0`, Bob at `j = 4`
  - Alice has enough water for plant `0` (needs 1 water), so `capacityA = 5 - 1 = 4`. No refills needed.
  - Bob has enough water for plant `4` (needs 3 water), so `capacityB = 6 - 3 = 3`. No refills needed.
  - Move Alice to `i = 1` and Bob to `j = 3`.
- **Step 2**: Alice at `i = 1`, Bob at `j = 3`
  - Alice has enough water for plant `1` (needs 2 water), so `capacityA = 4 - 2 = 2`. No refills needed.
  - Bob has enough water for plant `3` (needs 2 water), so `capacityB = 3 - 2 = 1`. No refills needed.
  - Move Alice to `i = 2` and Bob to `j = 2`.
- **Step 3**: Alice at `i = 2`, Bob at `j = 2` (they meet at the same plant)
  - The current plant `2` requires 4 water.
  - Alice only has `capacityA = 2`, which is not enough. She refills her can and `capacityA` is now `5 - 4 = 1`, and `ans` is incremented to `1`.
  - Bob doesn't water since Alice has already watered the plant.
  - Since Alice and Bob meet at the same plant and have gone through all plants, the while loop ends.

The total number of refills `ans` is `1`. Therefore, Alice and Bob needed to refill their cans only once combined to water all the plants.

Python Solution

```python
from typing import List

class Solution:
    def minimum_refill(self, plants: List[int], capacity_a: int, capacity_b: int) -> int:
        # Funtion to calculate the minimum number of refills needed.

        i, j = 0, len(plants) - 1  # Initialize pointers for Alice and Bob respectively.
        num_refills = 0  # Counter for the number of refills.
        current_a, current_b = capacity_a, capacity_b  # Current water capacities for Alice and Bob.

        while i <= j:
            if i == j:  # If Alice and Bob reach the same plant.
                if max(current_a, current_b) < plants[i]:  # If both can't water the plant, one needs to refill.
                    num_refills += 1
                break

            # Alice waters the plants from the beginning.
            if current_a < plants[i]:  # If Alice doesn't have enough water for the plant.
                current_a = capacity_a - plants[i]  # Refill minus what is needed for the current plant.
                num_refills += 1  # Increment refill counter.
            else:
                current_a -= plants[i]  # Subtract the amount of water used for the plant.

            # Bob waters the plants from the end.
            if current_b < plants[j]:  # If Bob doesn't have enough water for the plant.
                current_b = capacity_b - plants[j]  # Refill minus what is needed for the current plant.
                num_refills += 1  # Increment refill counter.
            else:
                current_b -= plants[j]  # Subtract the amount of water used for the plant.

            i += 1  # Move Alice to the next plant.
            j -= 1  # Move Bob to the previous plant.

        return num_refills  # Return the total number of refills needed.

# Example usage:
solution = Solution()
print(solution.minimum_refill([2, 4, 5, 1, 2], 5, 7))  # Example input to test the function.
```

Java Solution

```java
class Solution {

    // Method to determine the number of refills needed to water all plants
    public int minimumRefill(int[] plants, int capacityA, int capacityB) {
        int leftIndex = 0; // Starting index for Alice
        int rightIndex = plants.length - 1; // Starting index for Bob
        int refills = 0; // Counter for the number of refills
        int remainingA = capacityA; // Remaining water in Alice's can
        int remainingB = capacityB; // Remaining water in Bob's can

        // Loop through the plants while both pointers do not cross each other
        while (leftIndex <= rightIndex) {

            // When both Alice and Bob reach the middle plant
            if (leftIndex == rightIndex) {
                // If the plant's needs exceed both capacities, a refill is needed
                if (Math.max(remainingA, remainingB) < plants[leftIndex]) {
                    refills++;
                }
                break; // We break since this is the last plant to consider
            }

            // Watering the plant with Alice's can
            if (remainingA < plants[leftIndex]) {
                remainingA = capacityA - plants[leftIndex]; // Refill Alice's can and water the plant
                refills++; // Increment refill counter for Alice
            } else {
                remainingA -= plants[leftIndex]; // Use existing water for the plant
            }

            // Watering the plant with Bob's can
            if (remainingB < plants[rightIndex]) {
                remainingB = capacityB - plants[rightIndex]; // Refill Bob's can and water the plant
                refills++; // Increment refill counter for Bob
            } else {
                remainingB -= plants[rightIndex]; // Use existing water for the plant
            }

            // Move towards the middle plant
            leftIndex++;
            rightIndex--;
        }

        // Return the total number of refills needed by both Alice and Bob
        return refills;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Function calculates the minimum number of refills required to water all plants.
    int minimumRefill(std::vector<int>& plants, int capacityA, int capacityB) {
        int leftIndex = 0; // Starting from the leftmost plant
        int rightIndex = plants.size() - 1; // Starting from the rightmost plant

        int refills = 0; // Counter for the number of refills
        int remainingA = capacityA; // Remaining water in A's can
        int remainingB = capacityB; // Remaining water in B's can

        // Water the plants from both ends until the paths of A and B meet or cross
        while (leftIndex <= rightIndex) {
            // Check if both A and B are at the same plant
            if (leftIndex == rightIndex) {
                // If the plant's requirement is higher than both A's and B's capacity, add a refill and end the loop
                if (std::max(remainingA, remainingB) < plants[leftIndex]) {
                    ++refills;
                }
                break;
            }
            // Watering from A's side
            if (remainingA < plants[leftIndex]) {
                remainingA = capacityA - plants[leftIndex]; // Refill A's can and use water for the current plant
                ++refills; // Count the refill
            } else {
                remainingA -= plants[leftIndex]; // Deduct the water used for the current plant
            }

            // Watering from B's side
            if (remainingB < plants[rightIndex]) {
                remainingB = capacityB - plants[rightIndex]; // Refill B's can and use water for the current plant
                ++refills; // Count the refill
            } else {
                remainingB -= plants[rightIndex]; // Deduct the water used for the current plant
            }

            ++leftIndex; // Move A to the next plant
            --rightIndex; // Move B to the previous plant
        }

        return refills; // Return the total number of refills required
    }
};
```

Typescript Solution

```typescript
// Define a function to calculate the minimum number of refills required to water all plants.
function minimumRefill(plants: number[], capacityA: number, capacityB: number): number {
    let leftIndex: number = 0; // Starting from the leftmost plant
    let rightIndex: number = plants.length - 1; // Starting from the rightmost plant

    let refills: number = 0; // Counter for the number of refills
    let remainingA: number = capacityA; // Remaining water in A's can
    let remainingB: number = capacityB; // Remaining water in B's can

    // Water the plants from both ends until the paths of A and B meet or cross
    while (leftIndex <= rightIndex) {
        // Check if A and B are at the same plant
        if (leftIndex === rightIndex) {
            // If the plant's requirement is higher than both A's and B's remaining capacity, add a refill
            if (Math.max(remainingA, remainingB) < plants[leftIndex]) {
                refills++;
            }
            break;
        }

        // Watering from A's side
        if (remainingA < plants[leftIndex]) {
            remainingA = capacityA; // Refill A's can
            refills++; // Count the refill
            remainingA -= plants[leftIndex]; // Use water for the current plant
        } else {
            remainingA -= plants[leftIndex]; // Deduct the water used for the current plant
        }

        // Watering from B's side
        if (remainingB < plants[rightIndex]) {
            remainingB = capacityB; // Refill B's can
            refills++; // Count the refill
            remainingB -= plants[rightIndex]; // Use water for the current plant
        } else {
            remainingB -= plants[rightIndex]; // Deduct the water used for the current plant
        }

        leftIndex++; // Move A to the next plant
        rightIndex--; // Move B to the previous plant
    }

    return refills; // Return the total number of refills required
}

// Variable and function usage example:

// Capacity of watering cans for person A and B
const capacityA: number = 5;
const capacityB: number = 7;

// Plants array where each element represents the amount of water required to water the plant
const plants: number[] = [2, 4, 5, 1, 2, 3, 2, 3];

// Calculate the minimum number of refills needed to water all plants
const minRefills: number = minimumRefill(plants, capacityA, capacityB);

console.log(minRefills); // Outputs the result
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `plants` list. This is because the code uses a single `while` loop to traverse the list from both ends towards the center, performing a constant number of calculations at each step. In the worst case, the loop runs for the entire length of the list if there's only one refill station (when `i` starts at 0 and `j` at `n-1`, incrementing and decrementing respectively until they meet), thus the time complexity is linear with respect to the number of plants.

Space Complexity

The space complexity of the code is `O(1)`. This is because the code uses a fixed amount of extra space (variables to keep track of the positions `i`, `j`, the remaining capacities `capacityA` and `capacityB`, and the counter `ans` for the number of refills). This space requirement does not grow with the size of the input (`plants` list), hence the constant space complexity.