904. Fruit Into Baskets

Medium Array Hash Table Sliding Window

Problem Description

You are tasked to collect fruits from a series of fruit trees lined up in a row, with each tree bearing a certain type of fruit. You have two baskets to collect the fruits, and each basket can only hold one type of fruit, but as much of it as you'd like. Starting from any tree, you pick one fruit from each tree and move to the right, filling up your baskets with the fruits. The moment you encounter a tree with a fruit type that doesn't fit in either basket (since you can only carry two types), you must stop collecting. The goal is to maximize the amount of fruit you can collect under these constraints. The problem provides you with an array `fruits`, where `fruits[i]` denotes the type of fruit the ith tree produces. Your task is to determine the maximum number of fruits you can collect.

Intuition

This problem is a variation of the sliding window algorithm, which is useful for keeping track of a subset of data in a larger set. In this case, the subset is the range of trees from which we can collect fruits to fill our two baskets.

The intuition behind the solution is to maintain a window that slides over the array of fruit trees. The window should encompass the longest sequence of trees that only includes up to two types of fruits, satisfying the basket constraint. As we move through the array, we count the number of each type of fruit within the window using a counter. If adding a new fruit type to our window pushes us over the two-type limit, we shrink the window from the left until we are back within the limit, ensuring we always consider the maximum range of trees at each point.

The process is as follows:

- Start with an empty counter and a pointer at the beginning of the array.
- Iterate through the array one tree at a time, adding the fruit type to the counter.
- When the counter contains more than two types of fruits (i.e., we have more than two fruit types in our current window), we remove the leftmost fruit type from the window by decrementing its count in the counter. If its count reaches zero, we remove it from the counter.
- The window's size is adjusted accordingly throughout the process, and the maximum size of the window at any point during the iteration will be our answer.

Since we only ever add each element once and potentially remove each element once, this solution approach is efficient and works within O(n) time complexity, where n is the number of trees.

Solution Approach

The solution employs a few key concepts: sliding window, hash map (through Python's `Counter` class), and two pointers to optimize the process.

Here's a step-by-step walkthrough of how the `totalFruit` function operates:

1. **Initialize Counter**: The `Counter` from Python's `collections` module is used to keep track of the number of each type of fruit within our current window. It's essentially a hash map tying fruit types to their counts.

2. **Initialize Pointers**: Two pointers are initialized. The `j` pointer indicates the start of our sliding window, and the `for` loop index (`x` in the loop) acts as the end of the sliding window, moving from the first to the last tree.

3. **Iterate Over Fruit Trees**: The `for` loop begins iterating over the fruit trees. For each fruit type encountered, add it to our counter and increment its count.

```
for x in fruits:
    cnt[x] += 1
```

4. **Exceeding Basket Limit**: After adding the new fruit type, we check if we have more than two types of fruits in our counter (`len(cnt) > 2`). If we do, it means our current window of trees has exceeded the basket constraint, and we must shrink the window from the left.

```
if len(cnt) > 2:
    y = fruits[j]
    cnt[y] -= 1
    if cnt[y] == 0:
        cnt.pop(y)
    j += 1
```

In this block, we reduce the count of the leftmost fruit type by one (by getting the fruit at index `j`) and then increment the `j` pointer to effectively remove the tree from the window. If the leftmost fruit count drops to zero, it means there are no more trees of that fruit type in our current window, so we remove it from the counter.

5. **Maximum Fruits**: In each iteration, our window potentially encapsulates a valid sequence of at most two types of fruits. Since we're passing through all trees only once, and the `j` pointer never steps backward, the length of the maximum window will be found naturally. By subtracting `j` from the total number of fruits (`len(fruits)`), we get the length of this maximum window, which also represents the maximum number of fruits we can collect.

```
return len(fruits) - j
```

This algorithm effectively finds the longest subarray with at most two distinct elements, which corresponds to the largest quantity of fruit we can collect in our two baskets. The use of the sliding window technique allows for an efficient O(n) time complexity because each tree is evaluated only once, and each time the start of the window moves to the right, it never moves back.

Example Walkthrough

Let's illustrate the solution approach using a simple example. Consider the following series of fruit trees and their corresponding fruits:

```
Trees Index: 0  1  2  3  4  5  6
Fruits Type: A  B  A  A  C  B  B
```

We will walk through the algorithm step by step:

1. **Initialize Counter**: We will use a `Counter` to keep track of the types of fruits we currently have in our baskets.

2. **Initialize Pointers**: We will initialize two pointers, `j` and `x`. `j` will start at 0 and indicates the beginning of our window.

3. **Iterate Over Fruit Trees**: As we iterate over the trees with our `x` pointer, we'll perform the following steps:

```
x = 0, fruit = A
Counter = {'A': 1}
Window = [A]

x = 1, fruit = B
Counter = {'A': 1, 'B': 1}
Window = [A, B]

x = 2, fruit = A
Counter = {'A': 2, 'B': 1}
Window = [A, B, A]

x = 3, fruit = A
Counter = {'A': 3, 'B': 1}
Window = [A, B, A, A]

x = 4, fruit = C
Counter = {'A': 3, 'B': 1, 'C': 1}
Current window exceeds the allowed number of fruit types (more than 2).
```

4. **Exceeding Basket Limit**: Once we hit the third type of fruit, in this case, 'C', we need to shrink our window from the left side:

```
Counter after adding 'C' = {'A': 3, 'B': 1, 'C': 1} (more than 2 types, must remove one)
We remove the leftmost fruit type 'A' by one unit.
j = j + 1 () = 1 now)

Counter = {'A': 2, 'B': 1, 'C': 1}
We still have three types of fruits. We need to remove 'A' completely to satisfy the basket constraint.
So we remove 'A' one more time.
j = j + 1 () = 2 now)

Counter = {'A': 1, 'B': 1, 'C': 1}
Still more to do. We remove 'A' completely.
j = j + 1 () = 3 now)

Counter = {'B': 1, 'C': 1}
Now we have exactly 2 types of fruit in the basket.
Window = [A, C, B, B]
```

5. **Maximum Fruits**: We continue this process until the end. In the end, our window will look like this:

```
Final Window = [C, B, B]
Length of Final Window = 3
```

The length of the final window indicates the maximum number of fruits we can collect, which is 3 in this example.

Thus, using our algorithm, we determine that the longest sequence of trees where we can collect fruits without breaking the rules (up to 2 types of fruits) is `[C, B, B]`, and the total amount of fruit collected is 3.

Python Solution

```
from collections import Counter

class Solution:
    def totalFruit(self, fruits: List[int]) -> int:
        # Initialize a counter to keep track of the count of each type of fruit
        fruit_counter = Counter()
        # Initialize a variable to keep track of the starting index of the current window
        start_index = 0
        # Iterate over the list of fruits
        for fruit in fruits:
            # Increment the count for the current fruit
            fruit_counter[fruit] += 1
            # If the counter has more than two types of fruits, we shrink the window
            if len(fruit_counter) > 2:
                # The fruit at the start index needs to be removed or decremented
                start_fruit = fruits[start_index]
                fruit_counter[start_fruit] -= 1
                # Remove the fruit from counter if its count drops to 0
                if fruit_counter[start_fruit] == 0:
                    del fruit_counter[start_fruit]
                # Move the start index forward
                start_index += 1
        # Calculate the maximum length of the subarray with at most two types of fruits
        max_length = len(fruits) - start_index
        return max_length
```

Java Solution

```
import java.util.HashMap; // Import HashMap class for usage

class Solution {
    public int totalFruit(int[] tree) {
        // Create a HashMap to keep track of the count of each type of fruit
        HashMap<Integer, Integer> fruitCount = new HashMap<>();
        int start = 0; // Start of the sliding window
        int maxFruits = 0; // Maximum number of fruits collected

        // Iterate through the array of fruits using the end of the sliding window
        for (int end = 0; end < tree.length; end++) {
            // Add the current fruit to the fruitCount map or update its count
            fruitCount.put(tree[end], fruitCount.getOrDefault(tree[end], 0) + 1);

            // If the map contains more than 2 types of fruit, shrink the window from the start
            while (fruitCount.size() > 2) {
                fruitCount.put(tree[start], fruitCount.get(tree[start]) - 1);
                // If the count of a fruit at the start of the window becomes 0, remove it
                if (fruitCount.get(tree[start]) == 0) {
                    fruitCount.remove(tree[start]);
                }
                start++; // Move the start of the window forward
            }

            // Calculate the maximum number of fruits in the current window
            maxFruits = Math.max(maxFruits, end - start + 1);
        }
        // Return the size of the largest contiguous subarray with 2 types of fruits
        return maxFruits;
    }
}
```

C++ Solution

```
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int totalFruit(vector<int>& fruits) {
        // Initialize a hash map to count the fruits
        unordered_map<int, int> fruitCounter;

        // Initialize the start of the current window
        int windowStart = 0;

        // Get the number of fruits
        int totalFruits = fruits.size();

        // Iterate over all fruits
        for (int windowEnd = 0; windowEnd < totalFruits; ++windowEnd) {
            // Increase the count for the current fruit
            fruitCounter[fruits[windowEnd]]++;

            // If there are more than 2 types of fruits in the current window
            while (fruitCounter.size() > 2) {
                // Decrease the count of the fruit at the start of the window
                fruitCounter[fruits[windowStart]]--;

                // If the count becomes zero, remove the fruit from the map
                if (fruitCounter[fruits[windowStart]] == 0) {
                    fruitCounter.erase(fruits[windowStart]);
                }

                // Move the window start forward
                ++windowStart;
            }
        }

        // The maximum number of fruits is the size of the array minus the start of the last valid window
        return totalFruits - windowStart;
    }
};
```

Typescript Solution

```
function totalFruit(fruits: number[]): number {
    // Initialize the length of the fruits array
    const fruitCount = fruits.length;

    // Create a map to keep track of the frequency of each type of fruit within the sliding window
    const fruitFrequencyMap = new Map<number, number>();

    // `startIndex` represents the beginning index of the sliding window
    let startIndex = 0;

    // Iterate over each fruit in the fruits array
    for (const fruit of fruits) {
        // Add the fruit to the map or update its frequency
        fruitFrequencyMap.set(fruit, (fruitFrequencyMap.get(fruit) ?? 0) + 1);

        // If we have more than 2 types of fruits, shrink the window from the left
        if (fruitFrequencyMap.size > 2) {
            // Get the fruit type at the start of the window for updating its frequency
            const fruitTypeAtStart = fruits[startIndex++];
            fruitFrequencyMap.set(fruitType, fruitFrequencyMap.get(fruitType) - 1);

            // If the frequency of the leftmost fruit type becomes 0, remove it from the map
            if (fruitFrequencyMap.get(fruitType) === 0) {
                fruitFrequencyMap.delete(fruitType);
            }
        }
    }

    // Calculate the maximum number of fruits collected in a contiguous subarray of size at most 2
    return fruitCount - startIndex;
}
```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the number of fruits in the input list, `fruits`. This is because the code uses a single loop to iterate through all the fruits exactly once.

The space complexity of the code is also `O(n)`, primarily due to the `Counter` data structure `cnt` which in the worst-case scenario could store a frequency count for each unique fruit in the input list if all fruits are different. However, in practical terms, since the problem is constrained to finding the longest subarray with at most two distinct integers, the space complexity can effectively be seen as `O(2)` or constant `O(1)`, as there will never be more than two unique fruits in the counter at any time.