2561. Rearranging Fruits

Hard Greedy Array Hash Table   Leetcode Link

Problem Description

In this problem, we are given two fruit baskets with exactly `n` fruits each. These fruit baskets are represented by two integer arrays `basket1` and `basket2` where each element indicates the cost of the respective fruit. Our task is to make both baskets contain the exact same fruits in terms of their costs, which means that if we sort both baskets by their fruits' costs, they should look identical.

However, there are rules on how we can make the two baskets equal:

1. We can choose to swap the `i-th` fruit from `basket1` with the `j-th` fruit from `basket2`.
2. The cost incurred for swapping these two fruits is `min(basket1[i], basket2[j])`.

The goal is to find the minimum total cost required to make both baskets equal through such swapping operations. If it's not possible to make both baskets equal, we should return `-1`.

Intuition

To solve this problem, we need to follow a series of logical steps:

1. **Frequency Counting:** First, we use a frequency counter to understand the difference between the baskets in terms of fruit costs. For each matching pair of fruits (same cost in both baskets), there's no action required. If a fruit cost in `basket1` is not counter-balanced by the same cost in `basket2`, then alternations are needed.

2. **Balancing Counts:** We need to check if each fruit's imbalanced count is even. An odd imbalance means we cannot fully swap fruits to balance the baskets, so we return `-1`. For instance, an extra three fruits of cost 5 in `basket1` cannot be balanced by swaps since we would always end up with an uncoupled fruit.

3. **Calculating Minimum Swap Cost:** Since only pairs of unbalanced fruits need swapping, we sort the unbalanced fruits by their costs. We can then calculate the cost to swap half of these unbalanced fruits (each swap involves two fruits, hence 'half'). We choose either the cost of the current fruit or two times the minimum fruit cost for each swap, whichever is lesser, to ensure the lowest possible swap cost.

4. **Summation of Costs:** Summing up the swap costs will result in the minimum cost required to balance the baskets.

By following the code provided according to the intuition above, we ensure that we're swapping fruits in the most cost-effective way possible, resulting in the minimum cost to make both baskets equal.

Solution Approach

Step-by-Step Implementation:

1. **Create a Counter Object**: Using the `Counter` class from the `collections` module we create a `cnt` object to track the balance of fruits between `basket1` and `basket2`. A positive value in `cnt` means there are more fruits of that cost in `basket1` and vice versa.

2. **Zip and Update Counts**: We iterate through `basket1` and `basket2` in tandem using `zip`:

   ```
   for a, b in zip(basket1, basket2):
       cnt[a] += 1
       cnt[b] -= 1
   ```

   For each pair `(a, b)` of fruits, we increment the count of fruit `a` in `cnt` because it's from `basket1`, and decrement for fruit `b` because it's from `basket2`.

3. **Find the Minimum Cost Fruit**: We find the minimum cost fruit that can be used for calculating the swap cost using `min(cnt)`.

4. **Prepare List of Imbalanced Fruits**: Iterate over the `cnt` to find the imbalances. If there's an odd imbalance, we cannot make a swap to balance the fruit, and therefore the task is impossible and we return `-1`:

   ```
   nums = []
   for x, v in cnt.items():
       if v % 2:
           return -1
       nums.extend([x] * (abs(v) // 2))
   ```

   We repeat the fruit cost value `abs(v) // 2` times because we need pairs for swapping.

5. **Sort the List**: Sorting the list of costs ensures that when we choose fruits to swap, we always consider the least costly.

   ```
   nums.sort()
   ```

6. **Calculate the Cost for Half the Swaps**: We only need to make swaps for half of the imbalanced fruits. For each imbalanced fruit cost, choose the lower of the fruit's cost itself or double the minimum cost fruit:

   ```
   m = len(nums) // 2
   total_cost = sum(min(x, mi * 2) for x in nums[:m])
   ```

7. **Return the Minimum Total Cost**: Finally, we return the sum which represents the minimum cost to make both the baskets contain the same fruits.

By using a `Counter` to maintain the frequency of the fruits' costs, determining the least cost fruit for swap calculations, and leveraging sorting to guide our swapping strategy, we establish an efficient approach to determine the minimum swapping cost required—or recognize that balancing the baskets is infeasible.

Example Walkthrough

Let's consider a small example to illustrate the solution approach with two fruit baskets `basket1` and `basket2`, each with `n=5` fruits:

```
basket1 = [1, 3, 3, 2, 5]
basket2 = [2, 1, 4, 3, 4]
```

We want to find the minimum total cost to make the fruits in both baskets have the same costs after swapping.

1. **Create a Counter Object**: Create a counter to track the balance of fruits:

   ```
   cnt = Counter()
   ```

2. **Zip and Update Counts**: We compare `basket1` and `basket2` simultaneously and update our counter:

   ```
   cnt[1] += 1   # From basket1
   cnt[2] -= 1   # From basket2 (counterpart for basket1[1])
   cnt[3] += 1   # From basket1
   cnt[1] -= 1   # From basket2 (counterpart for basket1[3])
   cnt[3] += 1   # From basket1
   cnt[4] -= 1   # From basket2 (counterpart for basket1[3])
   cnt[2] += 1   # From basket1
   cnt[3] -= 1   # From basket2 (counterpart for basket1[2])
   cnt[5] += 1   # From basket1
   cnt[4] -= 1   # From basket2 (counterpart for basket1[5])

   After updating counts, `cnt` looks like this:
   cnt = {1: 0, 3: 1, 2: 0, 4: -2, 5: 1}
   ```

3. **Find the Minimum Cost Fruit**: The minimum fruit cost from the imbalanced fruits (`cnt`) is 3.

4. **Prepare List of Imbalanced Fruits**: Check the imbalances and gather the fruits that need to be swapped:

   ```
   nums = [3, 5]  # We need to swap one fruit with cost 3 and one with cost 5
   ```

   Since there are no odd values in `cnt`, we proceed.

5. **Sort the List**: We sort the imbalances to ensure we consider the least costly fruits first:

   ```
   nums.sort()
   nums = [3, 5]
   ```

6. **Calculate the Cost for Half the Swaps**: There are two imbalanced fruits, so we need one swap:

   ```
   mi = 3  # minimum cost from imbalanced fruits
   m = 1   # number of swaps needed
   total_cost = min(3, 3 * 2) for the first fruit
   ```

   In this case, we choose the fruit's own cost, which is 3, since it's not greater than double the minimum cost fruit.

7. **Return the Minimum Total Cost**: The sum is the minimum cost for making both the baskets equal:

   ```
   total_cost = 3
   ```

The minimum total cost required to make both baskets contain the same fruits in terms of cost is thus `3`. This would involve swapping one of the fruits with cost `3` from `basket1` with one of the fruits with cost `4` from `basket2`.

Python Solution

```python
from collections import Counter

class Solution:
    def minCost(self, basket1: List[int], basket2: List[int]) -> int:
        # Create a counter to track the frequency of each fruit type
        fruit_counter = Counter()

        # Iterate over both baskets simultaneously
        for fruit_type_a, fruit_type_b in zip(basket1, basket2):
            # Increment the count for fruit type from basket1
            fruit_counter[fruit_type_a] += 1
            # Decrement the count for fruit type from basket2
            fruit_counter[fruit_type_b] -= 1

        # Get the minimum fruit type value from our counter
        min_fruit_type = min(fruit_counter)

        # Prepare a list to count how many exchanges are needed
        exchange_list = []

        # Check if exchange is possible given the counts
        for fruit_type, count in fruit_counter.items():
            if count % 2:
                # If count is odd, return -1 (impossible to exchange)
                return -1
            # Add the fruit types for exchange to the list
            exchange_list.extend([fruit_type] * (abs(count) // 2))

        # Sort the list to facilitate minimum cost calculation
        exchange_list.sort()

        # Find the middle point of our sorted list
        mid_point = len(exchange_list) // 2

        # Calculate and return the cost of exchanges
        # By taking minimum exchange cost between the fruit type and double the minimum fruit type
        return sum(min(fruit_type, min_fruit_type * 2) for fruit_type in exchange_list[:mid_point])

```

Java Solution

```java
class Solution {
    public long minCost(int[] basket1, int[] basket2) {
        int n = basket1.length; // Length of the baskets
        Map<Integer, Integer> fruitCountMap = new HashMap<>(); // A map to store the count difference of fruits between baskets

        // Count the difference between the two baskets
        for (int i = 0; i < n; ++i) {
            fruitCountMap.merge(basket1[i], 1, Integer::sum);  // Increment count for the current fruit in basket1
            fruitCountMap.merge(basket2[i], -1, Integer::sum); // Decrement count for the current fruit in basket2
        }

        int minFruitValue = Integer.MAX_VALUE; // Initialize the minimum fruit value
        List<Integer> fruitDifferences = new ArrayList<>(); // List to store absolute differences

        // Analyze the map to find out the absolute difference and minimum fruit value
        for (var entry : fruitCountMap.entrySet()) {
            int fruit = entry.getKey(), count = entry.getValue();
            if (count % 2 != 0) { // If count is odd, there's no way to balance, return -1
                return -1;
            }
            for (int i = Math.abs(count) / 2; i > 0; --i) {
                fruitDifferences.add(fruit);  // Add the fruit differences
            }
            minFruitValue = Math.min(minFruitValue, fruit); // Update the minimum fruit value if necessary
        }

        Collections.sort(fruitDifferences); // Sort the list of differences

        int m = fruitDifferences.size(); // Size of the list of differences
        long totalCost = 0; // Initialize the total cost

        // Calculate the minimum cost of balancing the baskets
        for (int i = 0; i < m / 2; ++i) {
            totalCost += Math.min(fruitDifferences.get(i), minFruitValue * 2); // Take the minimum of the current fruit difference ar
        }

        return totalCost; // Return the total minimum cost
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // This method calculates the minimum cost to make two baskets identical
    long long minCost(vector<int>& basket1, vector<int>& basket2) {
        int n = basket1.size();
        unordered_map<int, int> countMap; // Map to store the difference in counts of fruits

        // Calculate the count differences for each fruit
        for (int i = 0; i < n; ++i) {
            countMap[basket1[i]]++;
            countMap[basket2[i]]--;
        }

        int minFruit = INT_MAX; // To store the minimum fruit value encountered
        vector<int> disparities; // Vector to store fruits which have disparities

        // Loop through the countMap to find disparities
        for (auto& [fruit, count] : countMap) {
            // If the count is odd, we cannot make the baskets identical so return -1
            if (count % 2) {
                return -1;
            }

            // Add the absolute half of the count of each fruit to the disparities vector
            for (int i = abs(count) / 2; i > 0; --i) {
                disparities.push_back(fruit);
            }

            // Update the minimum fruit value
            minFruit = min(minFruit, fruit);
        }

        // Sort the disparities vector in ascending order
        sort(disparities.begin(), disparities.end());

        int m = disparities.size(); // Size of the disparities vector
        long long totalCost = 0; // To store the total cost required to make the baskets identical

        // Loop to calculate the minimum cost required
        for (int i = 0; i < m / 2; ++i) {
            // Cost is the minimum between swapping with the cheapest fruit twice or the current disparity
            totalCost += min(disparities[i], minFruit * 2);
        }

        // Return the total cost calculated
        return totalCost;
    }
};

```

Typescript Solution

```typescript
// Importing required collections
import { HashMap } from "tstl/container/HashMap";
import { min } from "tstl/algorithm/math";

// This method calculates the minimum cost to make two baskets identical
function minCost(basket1: number[], basket2: number[]): number {
    const n: number = basket1.length;
    const countMap: HashMap<number, number> = new HashMap(); // Map to store the difference in counts of fruits

    // Calculate the count differences for each fruit
    for (let i = 0; i < n; ++i) {
        // Increase the count for the current fruit in basket1
        countMap.set(basket1[i], (countMap.get(basket1[i]) || 0) + 1);
        // Decrease the count for the current fruit in basket2
        countMap.set(basket2[i], (countMap.get(basket2[i]) || 0) - 1);
    }

    let minFruit: number = Number.MAX_VALUE; // To store the minimum fruit value encountered
    const disparities: number[] = []; // Array to store fruits which have disparities

    // Loop through the countMap to find disparities
    countMap.forEach((count, fruit) => {
        // If the count is odd, we cannot make the baskets identical so return -1
        if (count % 2) {
            return -1;
        }

        // Add the absolute half of the count of each fruit to the disparities array
        for (let i = Math.abs(count) / 2; i > 0; --i) {
            disparities.push(fruit);
        }

        // Update the minimum fruit value
        minFruit = min(minFruit, fruit);
    });

    // Sort the disparities array in ascending order
    disparities.sort((a, b) => a - b);

    const m: number = disparities.length; // Size of the disparities array
    let totalCost: number = 0; // To store the total cost required to make the baskets identical

    // Loop to calculate the minimum cost required
    for (let i = 0; i < m / 2; ++i) {
        // Cost is the minimum between swapping with the cheapest fruit twice or the current disparity
        totalCost += Math.min(disparities[i], minFruit * 2);
    }

    // Return the total cost calculated
    return totalCost;
}

```

Time and Space Complexity

Time Complexity:

The total time complexity of the given Python code is determined by multiple factors:

1. The loop where we zip `basket1` and `basket2` and update `cnt`: this loop runs for every pair of elements in the two lists, so if each list has `n` elements, this step is `O(n)`.

2. The `min(cnt)` operation: finding the minimum in the counter object depends on the number of unique elements in `cnt`, let's denote this number as `k`. In the worst case, all elements are unique so `k = n`. However, typically `k` is expected to be much less than `n`. This is a `O(k)` operation.

3. The loop to create `nums` list: the number of iterations is the sum of half the absolute values in `cnt` (since we're adding abs(v) // 2 elements for each x). In the worst case, this could be `O(n)` if every swap generates a new unique number in the counter.

4. The `sort()` call on `nums`: if we assume that `m` is the number of elements in `nums`, then sorting would take `O(m log m)`. In the worst case, where we have to add `n/2` elements to `nums`, this is `O(n log n)`.

5. The final loop where we sum the minimum of each element and `mi * 2`, running `m/2` times: this is `O(m)` which is `O(n)` in the worst case.

Adding these together, the time complexity is dominated by the sorting operation, resulting in `O(n log n)`.

Space Complexity:

The space complexity is also affected by multiple factors:

1. The Counter `cnt`, which has at most `k` unique elements, where `k` is the number of unique elements. In the worst case, `k = n`, so this is `O(n)`.

2. The `nums` list, which holds up to `n/2` elements in the worst case, making it `O(n)`.

3. The space required for the output of the `min()` operation (a single integer) and the final sum operation, both of which are constant, `O(1)`.

Therefore, the combined space complexity of the code is `O(n)`.