2107. Number of Unique Flavors After Sharing K Candies

Medium Array Hash Table Sliding Window Leetcode Link

Problem Description

You have an array of integers named `candies` where each element represents the flavor of a candy. The position of the flavored candy in the list is its index, starting from 0. You are tasked with sharing a certain number `k` of consecutive candies with your sister. The goal is to retain the highest number of unique candy flavors for yourself after you share `k` consecutive candies with her. You must figure out the maximum number of unique flavors you can keep after giving away `k` candies.

Intuition

The key to solving this problem lies in understanding that when you give away a range of consecutive candies, the unique flavors you have left are the ones not included in the range you gave away. To maximize the unique flavors, you want to give away the segment which has the least number of unique flavors. To find the solution, you can use a sliding window approach that moves across the `candies` array. This sliding window will represent the candies you will give to your sister.

- Starting after the first `k` candies, count the unique flavors you have (all the candies except the first `k` ones).
- Then, for each subsequent candy, adjust the count by removing the flavor going out of the window and adding the flavor coming into the window at the end.
- Note that if after removing a candy from the count its frequency drops to zero, remove it from the count to maintain an accurate representation of unique flavors.
- Update and keep track of the maximum count of unique flavors after each step.

By iterating through the array and adjusting the window of candies given to your sister, we can determine the maximum number of unique flavors that can be kept.

Solution Approach

The solution uses a sliding window approach alongside Python's `Counter` class (from `collections`) to keep track of the frequency of each flavor of candy that we can keep (i.e., candies not included in the `k` consecutive candies given to the sister).

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

1. Begin by counting the unique flavors of the candies that are initially outside the range of the first `k` candies using `Counter(candies[k:])`. This gives us the starting set of flavors we have excluding the first `k` to be given away.

2. Initialize the count of unique flavors (`ans`) to the length of the initial count, representing the number of unique flavors after giving away the first segment of `k` candies.

3. Iterate over the candies starting from index `k` up to the end:

   - For each candy at index `i`, decrement the count of the flavor encountered since it now becomes part of the segment to give away.
   - Increment the count of the flavor `i-k`, which is now coming into the window of candies we can keep, as the sliding window moves forward.
   - After adjusting the counts, check if the flavor we just decremented has a count of zero, indicating there are no more of that flavor to keep, and if so, remove it from the count.
   - Update the maximum count of unique flavors (`ans`) with the larger of the current `ans` or the current number of unique flavors after the adjustments.

The use of `Counter` makes it efficient to keep track of the frequency of each flavor and its updates, and the `max()` function assists in retaining the highest number of unique flavors after each iteration. By the end of the loop, `ans` holds the maximum number of unique flavors that can be kept.

```python
class Solution:
    def shareCandies(self, candies: List[int], k: int) -> int:
        cnt = Counter(candies[k:])
        ans = len(cnt)
        for i in range(k, len(candies)):
            cnt[candies[i]] -= 1
            cnt[candies[i - k]] += 1
            if cnt[candies[i]] == 0:
                cnt.pop(candies[i])
            ans = max(ans, len(cnt))
        return ans
```

This code snippet effectively finds the optimal number of unique flavors that can be retained, utilizing the efficiency of `Counter` for frequency tracking within the sliding window operated by the loop.

Example Walkthrough

Let's consider the `candies` array as `[1, 2, 3, 2, 2, 1, 3, 3]` and `k` equals `3`. Now, let's apply the solution step by step:

1. We start with the initial `Counter(candies[k:])` which excludes the first `k` candies: `Counter([2, 2, 1, 3, 3])`, resulting in a count of `{2: 2, 1: 1, 3: 2}`. Here, you have two unique flavors of #2, one of #1, and two of #3. Therefore, the initial `ans` (maximum number of unique flavors) is `3`, as there are three unique flavors in this count.

2. We now iterate over `candies` starting from the `k`-th element, which is the 3rd index (using zero-based indexing) in the original `candies` array.

3. At index `3` (`candies[3]` is `2`), we would give away the second flavor candy, which was in the initial segment to be kept. So, we decrement the count of flavor `2` from `{2: 2, 1: 1, 3: 2}` to `{2: 1, 1: 1, 3: 2}`.

4. We also include the flavor at index `0` (because the window is sliding forward), so flavor `1` is to be added back into our count. The count remains the same because flavor `1` is already in our count `{2: 1, 1: 1, 3: 2}`.

5. The current count of unique flavors is still `3`, and thus `ans` remains `3`.

6. Continue the iteration:

   - Move to index `4`: Remove one occurrence of flavor `2` and add back flavor `1`. Count is now `{2: 0, 1: 2, 3: 2}`. After removing the flavor `2` count which is zero, the count becomes `{1: 2, 3: 2}`. `ans` remains `3`.

   - Move to index `5`: Remove flavor `1` and add back flavor `2`. New count is `{1: 1, 3: 2}` (since `2` was not in the count, it doesn't get added). `ans` remains `3`.

   - Move to index `6`: Remove flavor `3` and there's no flavor at `i-k` (since we're now at the end of the array). Current count is `{1: 1, 3: 1}`. `ans` does not change and remains `3`.

By the end of the process, we have iterated through the array and adjusted the window properly. The final `ans` remains `3`, which signifies the maximum number of unique flavors we can retain after sharing `k` consecutive candies with the sister. The sliding window approach made it efficient to find this out.

Python Solution

```python
from collections import Counter

class Solution:
    def shareCandies(self, candies, k):
        # Initialize the counter with the candies excluding the first 'k' candies
        candy_counter = Counter(candies[k:])
        # Calculate initial number of unique candies
        max_unique = len(candy_counter)

        # Iterate over the list starting from the 'k-th' candy to the end
        for i in range(k, len(candies)):
            # Decrease the count of the outgoing candy (as the window slides)
            candy_counter[candies[i - k]] += 1
            # Increase the count of the incoming candy
            candy_counter[candies[i]] -= 1

            # Remove the outgoing candy from counter if its count falls to zero
            if candy_counter[candies[i]] == 0:
                del candy_counter[candies[i]]

            # Update the result with the maximum number of unique candies seen so far
            max_unique = max(max_unique, len(candy_counter))

        # Return the maximum number of unique candies that can be given to a sibling
        return max_unique

```

Java Solution

```java
class Solution {

    public int shareCandies(int[] candies, int k) {
        // This map will hold the count of each type of candy
        Map<Integer, Integer> candyCountMap = new HashMap<>();
        int totalCandies = candies.length;

        // Initialize the map with the candy count for the first window of size (totalCandies - k)
        for (int i = k; i < totalCandies; ++i) {
            candyCountMap.merge(candies[i], 1, Integer::sum);
        }

        // The initial answer is the number of distinct candies in the first window
        int maxDistinctCandies = candyCountMap.size();

        // Slide the window one candy at a time, updating the map accordingly
        for (int i = k; i < totalCandies; ++i) {
            // Decrement the count of the leftmost candy of the previous window
            // If its count drops to 0, remove it from the map
            if (candyCountMap.merge(candies[i - k], -1, Integer::sum) == 0) {
                candyCountMap.remove(candies[i - k]);
            }

            // Increment the count of the new candy entering the current window
            candyCountMap.merge(candies[i], 1, Integer::sum);

            // Update the max distinct candies seen so far
            maxDistinctCandies = Math.max(maxDistinctCandies, candyCountMap.size());
        }

        // Return the maximum number of distinct candies
        return maxDistinctCandies;
    }
}

```

C++ Solution

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

class Solution {
public:
    int shareCandies(vector<int>& candies, int k) {
        // Create a hash map to store the frequency count of each type of candy.
        unordered_map<int, int> candy_count;
        int total_candies = candies.size();

        // Populate the map with the frequencies of the candies starting from index 'k'.
        for (int i = k; i < total_candies; ++i) {
            ++candy_count[candies[i]];
        }

        // Initialize the maximum number of unique candies as the current size of the map.
        int max_unique_candies = candy_count.size();

        // Use a sliding window of size 'k' to find the maximum number of unique candies
        // that can be shared.
        for (int i = k; i < total_candies; ++i) {
            // Decrease the count of the current candy as it is leaving the window.
            if (--candy_count[candies[i]] == 0) {
                // If the count becomes zero, remove it from the map.
                candy_count.erase(candies[i]);
            }

            // Increase the count for the new candy entering the window.
            ++candy_count[candies[i - k]];

            // Update the max_unique_candies with the current number of unique candies.
            max_unique_candies = std::max(max_unique_candies, static_cast<int>(candy_count.size()));
        }

        // Return the maximum number of unique candies that can be shared.
        return max_unique_candies;
    }
};

```

Typescript Solution

```typescript
// import the necessary modules. In TypeScript, you typically import modules instead of including headers.
import { max } from "lodash";

// Define the variables and methods globally as required.

// Function to find the share of candies.
// candies: The array of candy types.
// k: The number of candies to share.
function shareCandies(candies: number[], k: number): number {
    // Create a map to store the frequency count of each type of candy.
    const candyCount: Map<number, number> = new Map<number, number>();
    const totalCandies = candies.length;

    // Populate the map with the frequencies of the candies starting from index 'k'.
    for (let i = k; i < totalCandies; i++) {
        candyCount.set(candies[i], (candyCount.get(candies[i]) || 0) + 1);
    }

    // Initialize the maximum number of unique candies as the current size of the map.
    let maxUniqueCandies = candyCount.size;

    // Use a sliding window of size 'k' to find the maximum number of unique candies
    // that can be shared.
    for (let i = k; i < totalCandies; i++) {
        // Decrease the count of the current candy as it is leaving the window.
        const currentCandyCount = candyCount.get(candies[i])! - 1;
        if (currentCandyCount === 0) {
            // If the count becomes zero, remove it from the map.
            candyCount.delete(candies[i]);
        } else {
            // If count is not zero, then update the count in the map.
            candyCount.set(candies[i], currentCandyCount);
        }

        // Increase the count for the new candy entering the window.
        const newCandyCount = (candyCount.get(candies[i - k]) || 0) + 1;
        candyCount.set(candies[i - k], newCandyCount);

        // Update the maxUniqueCandies with the current number of unique candies.
        maxUniqueCandies = max([maxUniqueCandies, candyCount.size])!;
    }

    // Return the maximum number of unique candies that can be shared.
    return maxUniqueCandies;
}

// Note that TypeScript does not have a direct equivalent to `unordered_map` in C++.
// We use the `Map` object in TypeScript for the key-value mapping.
// Additionally, TypeScript requires explicit null-checks or 'non-null assertion operators' (!) when dealing with potentially null va

// Usage of lodash's max function requires it to be installed and imported, or you could write a utility function for that purpose.

```

Time and Space Complexity

Time Complexity

The initial setup of the Counter `cnt` with the slice `candies[k:]` is `O(n-k)` where `n` is the length of `candies`.

The loop runs for `n-k` iterations (since it starts from `k` and goes up to `n`). Within each iteration, the operations on the Counter object (`cnt[candies[i]] -= 1`, `cnt[candies[i - k]] += 1`, and `cnt.pop(candies[i])`) can be considered `O(1)` on average, as they involve updating the hash table.

Therefore, the total time complexity of the loop is `O(n-k)`. Since the setup time is also `O(n-k)` and these are done sequentially, the overall time complexity of the function is `O(n-k + n-k)`, which simplifies to `O(n)`, assuming `k` is much smaller than `n`.

Space Complexity

The space complexity is dominated by the space needed to store the Counter object `cnt`. In the worst case, if all `candies` elements are unique, the Counter will have up to `n-k` entries, leading to a space complexity of `O(n-k)`. In the best-case scenario, if all elements are the same, the Counter will have just one entry, but since `n` is the determining factor (assuming `k` doesn't scale with `n`), the space complexity remains `O(n)`.