1744. Can You Eat Your Favorite Candy on Your Favorite Day

Medium Array Prefix Sum Leetcode Link

Problem Description

In this problem, we are required to simulate a game of eating candies with some specific rules:

1. We are given an array `candiesCount` where each element `candiesCount[i]` denotes the number of candies of type `i`.
2. We also have a list of queries. Each query is given as an array with 3 elements: `[favoriteType, favoriteDay, dailyCap]`.
3. The game starts on day `0` and we must eat at least one candy every day.
4. We must finish eating all candies of type `i` before we can start on type `i+1`.
5. Our goal is to determine for each query if it is possible to eat at least one candy of our `favoriteType` on the `favoriteDay` without exceeding a maximum number of candies defined by `dailyCap`.

The essence of the problem is to check if our eating strategy, adhering to the rules, allows us to satisfy all the conditions described in a query.

Intuition

The intuition behind the solution leverages the understanding that to satisfy the queries, we must figure out the minimum and maximum number of candies we could have eaten by `favoriteDay`. These two values create a range, and if our `favoriteType` candy falls within that range, the answer for the query is true. Otherwise, it's false.

Here's how we arrive at the solution:

1. First, we use prefix sums to calculate the total number of candies eaten up to each type. Prefix sums help us quickly determine the total count without repeatedly summing individual counts.
2. For each query, we calculate the minimum number of candies possibly eaten by `favoriteDay` by assuming we eat only one candy per day, which is the same as the `favoriteDay` itself because we start at day `0`.
3. We also calculate the maximum number of candies by multiplying the `favoriteDay` plus one (since we start at day `0`) by the `dailyCap`.
4. The `favoriteType` candy can be eaten if it falls within the range created by our minimum and maximum possible candies. This means the following two conditions must hold true:
   - The number of candies eaten up to `favoriteType - 1` (thus `s[favoriteType]`) must be less than the most we could eat by `favoriteDay` (which we've calculated).
   - And, the number of candies we could have started eating by `favoriteDay` (`least` + 1) must be less than or equal to the total candy count up to including `favoriteType` (`s[favoriteType + 1]`).

The given solution neatly calculates this for every query, recording the answers in an array `ans` to return the results.

Solution Approach

The solution is implemented in Python using the following concepts:

1. **Prefix Sums**: We use the `accumulate` function from Python's `itertools` module to generate a list `s` that contains the cumulative sum of the candies. The `initial=0` parameter ensures that there is a `0` prefixed to the accumulated list, which facilitates calculating the number of candies eaten before a certain type.

2. **Array Manipulation**: The primary operation in the solution is element-wise manipulation and comparison within arrays or lists. These include:

   - Accessing the sums of candies before the current type (`s[t]`).
   - Calculating the sums of candies including the current type (`s[t + 1]`).
3. **Iterating Over Queries**: Each query is represented as a list `[t, day, mx]`, and we loop through the `queries` list while calculating the corresponding boolean value for each query to determine if the eating plan for a given `favoriteType` is feasible on the `favoriteDay`.

4. **Calculating Minimum and Maximum Candy Range**: For every query, we calculate the minimum (`least`) and maximum (`most`) number of candies that could have been eaten by the `favoriteDay`.

   - `least` is the day number itself, which assumes that we eat one candy per day.
   - `most` is the number `(day + 1) * mx`, reflecting the maximum consumption rate according to the `dailyCap`.
5. **Range Checking**: Using the values in `s`, we check if `least` is less than `s[t+1]`, which signifies that the least number of candies we could eat by `favoriteDay` is enough to allow us to start eating the `favoriteType`. We also check that `most` is greater than `s[t]`, which ensures we haven't already surpassed the amount of the `favoriteType` of candy.

6. **Appending Results**: If both conditions hold true for a query, we append `True` to our `ans` list; otherwise, we append `False`. This list is what is eventually returned.

The implementation can be summarized in the following steps:

- Calculate the prefix sums of `candiesCount` and store it in `s`.
- For each `query` in `queries` list:
  - Extract `favoriteType` (`t`), `favoriteDay` (`day`), and `dailyCap` (`mx`) from the query.
  - Calculate the `least` and `most` number of candies that could have been eaten by the `favoriteDay`.
  - Append `True` or `False` to the `ans` list based on if the ranges overlap with the amount available for the `favoriteType`.
- Return the `ans` list once all queries have been processed.

Example Walkthrough

Let's go through a small example to illustrate the solution approach provided.

Suppose we have an input array, `candiesCount`, and a list of queries given as follows:

```
candiesCount = [7, 4, 5, 3]
queries = [[0, 2, 2], [1, 6, 1], [2, 5, 10]]
```

In this scenario:

- We have 7 candies of type 0, 4 of type 1, 5 of type 2, and 3 of type 3.
- The queries detail scenarios where we want to know if we can eat our favorite type of candy on the favorite day without exceeding a set daily cap.

Calculate Prefix Sums

Firstly, we calculate prefix sums of `candiesCount`:

```
s = [0, 7, 11, 16, 19]  # Calculated using prefix sums; we added a 0 at the start.
```

Process Each Query

Now, let's process each query using our example:

Query 1: `[0, 2, 2]`

- Favorite type: `0`
- Favorite day: `2`
- Daily cap: `2`

We calculate the `least` number of candies possibly eaten by day `2`, which is `2`, and the `most`, which is `(2 + 1) * 2 = 6`.

Since `s[0] = 0` and `s[1] = 7`, the number of candies of type `0` we could have started eating by day `3` is less than `7`. Thus, `0 < 7 <= 6` does not hold true, so our answer is `False`.

Query 2: `[1, 6, 1]`

- Favorite type: `1`
- Favorite day: `6`
- Daily cap: `1`

Calculating `least` as `6` and `most` as `(6 + 1) * 1 = 7`.

The number of candies eaten before type `1` (`s[1] = 7`) should be less than `most = 7`, and the number we can start eating by day `6` (`least + 1 = 7`) should be less than or equal to the candies of type `1` available (`s[2] = 11`), which means `7 <= 11`. Now, `6 < 11` and `7 <= 11` both hold true, hence the response is `True`.

Query 3: `[2, 5, 10]`

- Favorite type: `2`
- Favorite day: `5`
- Daily cap: `10`

`least` is `5`, and `most` now is `(5 + 1) * 10 = 60`.

Considering `s[2] = 11` and `s[3] = 16`, we have not eaten type `2` candies by day `5`, since `11 <= 60`. We could also start eating them the same day since `5 + 1 = 6` is less than `16`. Thus, the conditions `11 <= 60` and `6 <= 16` are satisfied, resulting in `True`.

Construct Answer Array

Finally, we have processed all queries and our answer array `ans` is `[False, True, True]`, indicating the feasibility of each query.

Complete Process

To apply this approach in an actual Python solution, follow the summarized steps outlined in the content provided. Calculate the prefix sums, iterate over every query to determine the range in which we can eat candies, check the range conditions, and then append the result to the answer list which is then returned.

Python Solution

```python
from itertools import accumulate

class Solution:
    def canEat(self, candies_count: List[int], queries: List[List[int]]) -> List[bool]:
        # Use accumulate function with initial=0 to create a prefix sum array
        # where s[i] represents the total number of candies up to (but not including) index i
        prefix_sum = list(accumulate(candies_count, initial=0))

        # Initialize an empty list to store the answer to each query
        answer_list = []

        # Iterate through each query in the queries list
        for candy_type, day, max_candies_per_day in queries:
            # Calculate the least number of candies the user could eat
            least_candies_eaten = day
            # Calculate the most number of candies the user could eat
            most_candies_eaten = (day + 1) * max_candies_per_day

            # Determine if the user can eat at least one candy of type 'candy_type'
            # by checking if they would still have candies left on the 'day'
            # and also making sure they won't run out of candies before the day
            can_eat = least_candies_eaten < prefix_sum[candy_type + 1] and most_candies_eaten > prefix_sum[candy_type]

            # Add the result to the answer list
            answer_list.append(can_eat)

        # Return the list of answers for each query
        return answer_list

```

Java Solution

```java
class Solution {
    public boolean[] canEat(int[] candiesCount, int[][] queries) {
        int typesOfCandies = candiesCount.length;
        long[] cumulativeCandies = new long[typesOfCandies + 1];

        // Calculate the cumulative sum of candies to determine the total
        // number of candies up to a certain type
        for (int i = 0; i < typesOfCandies; ++i) {
            cumulativeCandies[i + 1] = cumulativeCandies[i] + candiesCount[i];
        }

        int numberOfQueries = queries.length;
        boolean[] canEatOnDay = new boolean[numberOfQueries];

        // Iterate over each query to determine if you can eat the candies
        for (int i = 0; i < numberOfQueries; ++i) {
            int type = queries[i][0]; // Type of candy
            int day = queries[i][1]; // Day number
            int maxCandies = queries[i][2]; // Maximum number of candies you can eat

            // The earliest amount of candies you could eat by that day
            // is equal to the day number if you eat 1 candy per day

            // Whereas the most amount of candies you could eat is if
            // you eat 'maxCandies' on each day including 'day'

            long leastCandiesYouCouldEat = day;
            long mostCandiesYouCouldEat = (long) (day + 1) * maxCandies;

            // Check if there's any overlap between the range of candies you could eat
            // and the range of candies available for that type. You can eat the candy
            // on that day if at least one candy of that type is within your eating range.
            canEatOnDay[i] = leastCandiesYouCouldEat < cumulativeCandies[type + 1] && mostCandiesYouCouldEat > cumulativeCandies[type
        }

        return canEatOnDay;
    }
}

```

C++ Solution

```cpp
#include <vector>

// Type alias for long long for ease of use
using ll = long long;

class Solution {
public:
    // Function to determine if you can eat all your favorite candies
    vector<bool> canEat(vector<int>& candiesCount, vector<vector<int>>& queries) {

        // Get the number of different types of candies
        int numTypes = candiesCount.size();

        // Prefix sum array to store the total candies up to index i
        vector<ll> prefixSum(numTypes + 1);
        for (int i = 0; i < numTypes; ++i)
            prefixSum[i + 1] = prefixSum[i] + candiesCount[i];

        // Result vector to store whether each query is true or false
        vector<bool> results;

        // Process each query
        for (auto& query : queries) {
            // Type of favorite candy for this query
            int favoriteType = query[0];
            // Day on which you plan to eat the favorite candy
            int day = query[1];
            // Maximum number of candies you can eat each day
            int maxCandiesPerDay = query[2];

            // Least number of candies you could have eaten until 'day'
            ll leastCandiesEaten = day;
            // Most number of candies you could have eaten until 'day'
            ll mostCandiesEaten = 1ll * (day + 1) * maxCandiesPerDay;

            // Check if it is possible to eat some favorite candies on that day
            // We can eat our favorite candy only if we have not eaten all of them before that day
            // and we would have been able to reach to our favorite candy type by that day.
            results.emplace_back(leastCandiesEaten < prefixSum[favoriteType + 1] &&
                                 mostCandiesEaten > prefixSum[favoriteType]);
        }

        // Return result vector
        return results;
    }
};

```

Typescript Solution

```typescript
// Define a type alias for number to represent large counts
type ll = number;

// Array to store counts for each type of candy
let candiesCount: number[];

// Matrix to store queries, where each query is an array containing:
// 0. Type of favorite candy
// 1. Day to eat the favorite candy
// 2. Maximum number of candies that can be eaten in one day
let queries: number[][];

/**
 * Function to determine if you can eat all your favorite candies based on queries
 * @param candiesCount - Array with count of each type of candy
 * @param queries - Array of queries with details [favoriteType, day, maxCandiesPerDay]
 * @returns Array of boolean values where each value corresponds to the possibility of a query
 */
function canEat(candiesCount: number[], queries: number[][]): boolean[] {

    // Get the number of different types of candies
    const numTypes: number = candiesCount.length;

    // Array to store the prefix sum for total candies up to index i
    const prefixSum: ll[] = new Array(numTypes + 1).fill(0);
    for (let i = 0; i < numTypes; ++i) {
        prefixSum[i + 1] = prefixSum[i] + candiesCount[i];
    }

    // Array to store the results of the queries
    const results: boolean[] = [];

    // Iterate through each query and process it
    for (const query of queries) {
        // Extract the query details for readability
        const favoriteType: number = query[0];
        const day: number = query[1];
        const maxCandiesPerDay: number = query[2];

        // Calculate the least number of candies that could have been eaten until the 'day'
        const leastCandiesEaten: ll = day;

        // Calculate the most number of candies that could have been eaten until the 'day'
        const mostCandiesEaten: ll = (day + 1) * maxCandiesPerDay;

        // Determine if it's possible to eat some of the favorite candies on that day
        const canEatFavorite: boolean = leastCandiesEaten < prefixSum[favoriteType + 1]
                                      && mostCandiesEaten > prefixSum[favoriteType];

        // Add the result to the results array
        results.push(canEatFavorite);
    }

    // Return the array of results from the queries
    return results;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n + q)`, where `n` is the length of the `candiesCount` array, and `q` is the number of queries. The `accumulate()` function is called once on the `candiesCount` array, and this operation has a time complexity of `O(n)` for calculating the prefix sums. Then the algorithm processes each query in constant time, so the time complexity for processing all queries is `O(q)`. Therefore, the combined time complexity is the sum of the two, which results in `O(n + q)`.

The space complexity of the provided code is `O(n)`, as the `s` array stores the prefix sums of the `candiesCount` array, which is the only additional significant space used by the algorithm.

Please note that the space taken by the output array `ans` is not included in this analysis, as it's usually considered the space required to store the output of the function, and is typically not counted towards the additional space complexity of an algorithm.