1431. Kids With the Greatest Number of Candies

Easy Array Leetcode Link

Problem Description

In this problem, we are presented with `n` kids, each with a certain number of candies. The `candies` array represents the count of candies every kid has. We are also given `extraCandies`, which is a number of additional candies available to distribute to the kids. Our task is to determine, for each kid, whether giving them all the `extraCandies` would result in them having the greatest number of candies compared to all other kids. It's important to note that more than one kid can hold the title of having the most candies if they end up with the same number of candies after distributing `extraCandies`. We need to return a boolean array, where each element corresponds to a kid. The value is `true` if the kid can have the most candies after receiving `extraCandies` and `false` otherwise.

Intuition

To solve this problem, we look to determine the highest number of candies any kid currently has, which we can do by finding the maximum value in the `candies` array. Once we know this maximum value, the intuition is simple: for each kid, we add `extraCandies` to their current number of candies and compare it with the maximum number of candies. If the sum is greater than or equal to the maximum, it means that by receiving the `extraCandies`, this kid could potentially have the greatest number of candies. We can do this comparison for each kid, resulting in an array of boolean values that indicate whether each kid is enabled to reach or surpass the maximum candy count with the extra candies in hand.

Solution Approach

In the provided solution, we use a simple algorithm to accomplish the task. The implementation involves the following steps:

1. Find the maximum number of candies that any kid has by utilizing the built-in `max()` function, which is efficient for this purpose as it iterates through the list `candies` once and retrieves the highest value. This step is represented by the line of code:

   ```
   mx = max(candies)
   ```

2. We then iterate through the list of `candies` using a list comprehension—a concise way to generate a new list by applying an expression to each item in an iterable. For each element (representing each kid's candy count) in the `candies` list, we add the number of `extraCandies` and check if the new total is greater than or equal to the maximum candy count found in the first step. This step effectively checks whether distributing the `extraCandies` to each kid in turn would make their total equal to or greater than the current maximum.

3. The comparison `candy + extraCandies >= mx` yields a boolean result (`True` or `False`) which indicates whether after receiving `extraCandies`, that particular kid will have candies not less than the kid with the most candies.

4. The list comprehension returns a new list of boolean values—a direct outcome of the comparisons—which answers the problem's question for each kid. The line of code for this step looks like this:

   ```
   return [candy + extraCandies >= mx for candy in candies]
   ```

Using this approach, the algorithm achieves a time complexity of O(n), where n is the number of elements in the `candies` list, since the operations involve a single scan to find the maximum and another to evaluate the condition for each kid. As for space complexity, it is also O(n), as the output is a list of n boolean values.

The utilized data structure is a simple list, and the pattern applied is a straightforward linear scan for comparison. The beauty of this solution lies in its simplicity and direct approach to determining the outcome for each kid in relation to the others.

Example Walkthrough

Let's walk through a small example to demonstrate the solution approach. Assume we have five kids with the following number of candies: `candies = [2, 3, 5, 1, 3]`, and we have `extraCandies = 3` to distribute.

1. Find the maximum number of candies that any kid has:

   ```
   mx = max(candies)  # mx = max([2, 3, 5, 1, 3]) = 5
   ```

   The maximum number of candies with any kid is 5.

2. With the `extraCandies`, we need to see if each kid can reach or surpass the maximum count of 5 candies. We do this by adding `extraCandies` to the current number of candies each kid has and checking if it's equal to or greater than 5.

3. Generate the new list of boolean values by comparing the current number of candies plus `extraCandies` to the maximum number:

   ```
   result = [candy + extraCandies >= mx for candy in candies]
   # result = [2+3 >= 5, 3+3 >= 5, 5+3 >= 5, 1+3 >= 5, 3+3 >= 5]
   # result = [True, True, True, False, True]
   ```

   Kid 1: Starts with 2 candies; adding 3 extra candies gives them 5, which equals the maximum. Therefore, the first element is `True`.

   Kid 2: Starts with 3 candies; adding 3 extra candies gives them 6, which is more than the maximum. Therefore, the second element is `True`.

   Kid 3: Already has the maximum of 5 candies; adding 3 more makes it 8, hence the third element is `True`.

   Kid 4: Starts with 1 candy; adding 3 extra candies gives them only 4, which is less than the maximum. Therefore, the fourth element is `False`.

   Kid 5: Starts with 3 candies; adding 3 extra candies also gives them 6, resulting in the fifth element being `True`.

4. Returning this list gives us the final answer to the problem:

   ```
   result = [True, True, True, False, True]
   ```

Using this example, each kid, except for the fourth one, can have the most candies if they receive all `extraCandies`. This is exactly what the boolean array indicates and is in alignment with the problem description and solution approach.

Python Solution

```python
from typing import List

class Solution:
    def kidsWithCandies(self, candies: List[int], extraCandies: int) -> List[bool]:
        # Find the maximum number of candies that any child currently has
        max_candies = max(candies)

        # Create a list of boolean values, where each value indicates whether a child
        # can have the greatest number of candies by adding the extraCandies to their current amount
        can_have_most_candies = [
            (child_candies + extraCandies) >= max_candies for child_candies in candies
        ]

        # Return the list of boolean values
        return can_have_most_candies

```

Java Solution

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Solution {

    // Function to determine which kids can have the greatest number of candies
    // after they receive an additional amount of extraCandies.
    public List<Boolean> kidsWithCandies(int[] candies, int extraCandies) {

        // Find the maximum number of candies that any kid currently has.
        int maxCandies = Integer.MIN_VALUE;
        for (int candy : candies) {
            maxCandies = Math.max(maxCandies, candy);
        }

        // List to store the results, whether each kid can have the maximum number
        // of candies after receiving extraCandies.
        List<Boolean> result = new ArrayList<>();

        // Loop through each kid's candies to determine if they can reach maxCandies
        // with their current candies plus extraCandies.
        for (int candy : candies) {
            // If the current kid's candies plus extraCandies is greater than or
            // equal to maxCandies, add 'true' to the result list, otherwise add 'false'.
            result.add(candy + extraCandies >= maxCandies);
        }

        // Return the result list.
        return result;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to determine which kids can have the greatest number of candies
    // after receiving extraCandies
    vector<bool> kidsWithCandies(vector<int>& candies, int extraCandies) {
        // Find the maximum number of candies any kid currently has
        int maxCandies = *max_element(candies.begin(), candies.end());

        // Initialize a result vector to store boolean values indicating
        // whether each kid can have the maximum number of candies
        vector<bool> result;

        // Iterate through the number of candies each kid has
        for (int candyCount : candies) {
            // Check if giving extraCandies to the kid makes their total equal
            // or greater than maxCandies and add the result to the result vector
            result.push_back(candyCount + extraCandies >= maxCandies);
        }

        return result; // Return the completed result vector
    }
};

```

Typescript Solution

```typescript
// Function to determine which kids can have the greatest number of candies
// after being given extra candies.
// Parameters:
// candies: an array of integers representing the number of candies each kid has.
// extraCandies: an integer representing the number of extra candies to give.
// Returns an array of booleans indicating whether each kid can have the greatest
// number after receiving the extra candies.
function kidsWithCandies(candies: number[], extraCandies: number): boolean[] {

    // Find the maximum number of candies any kid currently has.
    const maxCandies = candies.reduce((max, current) => Math.max(max, current), 0);

    // Determine if each kid could have the most candies by adding the extraCandies to their current count.
    // The result is an array of boolean values corresponding to each kid.
    const canHaveMostCandies = candies.map(candyCount => candyCount + extraCandies >= maxCandies);

    return canHaveMostCandies;
}

```

Time and Space Complexity

Time Complexity

The provided function involves iterating over the list `candies` twice - once to find the maximum value with the `max` function, and once in the list comprehension to check each child's candy count.

1. `max(candies)`: This call takes `O(n)` time, where `n` is the number of elements in the `candies` list.
2. List comprehension: This also takes `O(n)` time as it iterates through the list `candies` once.

Adding both gives us a total time complexity of `O(n) + O(n)`, which simplifies to `O(n)`, because constant factors are ignored in big O notation.

Space Complexity

The space complexity of the function is determined by the additional space required for the output list:

1. List comprehension creates a new list with the same number of elements as the input list `candies`. This requires `O(n)` space.

There's no other significant additional space used in the function, so the total space complexity is `O(n)`.