2779. Maximum Beauty of an Array After Applying Operation

Medium Array Binary Search Sorting Sliding Window

Problem Description

You are provided with an array `nums` which is 0-indexed. This means the first element of the array is at position 0, the next element is at position 1, and so on. A non-negative integer `k` is also given to you.

There is a set of operations you can perform on `nums` to potentially increase its beauty. An array's beauty is defined as the length of the longest subsequence where all elements are equal. A subsequence is any subset of the array that is not necessarily contiguous but maintains the original order of elements.

Each operation follows two steps:

1. Choose an index `i` from the array which has not been chosen before.
2. Replace the element at `nums[i]` with any number within the range `[nums[i] - k, nums[i] + k]`.

Your goal is to execute these operations as many times as you see fit to maximize the beauty of the array, but you can only choose each index once. The question asks for the maximum beauty that can be achieved through such operations.

Intuition

The problem is, in essence, about finding the largest group of numbers in the `nums` array that can be made equal through the allowed operations. If we can increase or decrease each number by up to `k`, we can think of each number as having a "reach" of `2k + 1` (from `nums[i] - k` to `nums[i] + k`), where all numbers in this reach can be made equal to `nums[i]`.

One approach to solving this problem is to use a difference array, which is an efficient method for applying updates over a range of indices in an array. The general idea is to build an augmented array, `d`, where each element represents the difference between successive elements in the `nums` array. This array `d` is then used to accumulate the total possible count of each value in the original array after applying all possible operations.

The array `d` is initialized with zeros and should have enough capacity to accommodate the largest number after an operation (which is `max(nums) + 2 * k + 1`) plus an additional element. For each element in `nums`, we can perform the following in the array `d`:

1. Increment the value at index `x` (the original number) because you can always choose not to change the number, contributing to its count.
2. Decrement the value at index `x + 2 + k + 1`, which is beyond the reach of the operations from `nums[i]`, thus ending the impact of this specific number.

Once the difference array `d` is fully populated, we iterate over it, accumulating the counts and keeping track of the maximum value found, which corresponds to the maximum beauty of the array.

Here, the variable `ans` is used to track the maximum beauty, and `s` is the running sum while iterating through the augmented array `d`. In each step, `s` is incremented by the current value in `d`, and `ans` is updated if `s` is greater than the current `ans`.

Overall, this approach effectively allows assessing the impact of all operations in a cumulative fashion, facilitating the computation of the maximum beauty in a time-efficient manner.

Solution Approach

The provided solution uses a `difference array` as a key data structure. A difference array is an array that allows updates over intervals in the original array in `O(1)` time and querying in `O(n)` time, where `n` is the number of elements. This turns out to be very efficient for certain types of problems — such as this one, where we repeatedly perform a fixed change to a range of elements.

Here's a step-by-step breakdown of the `maximumBeauty` function's implementation:

1. Calculate the size `m` for the difference array by taking the maximum element in `nums`, adding twice `k`, and then adding 2. This guarantees the difference array can cover all the values after all the operations. The actual value is `m = max(nums) + k * 2 + 2`.
2. Initialize a difference array `d` with size `m` filled with zeros. This array will track the potential increases and decreases over the ranges from all `nums[i]`.
3. Loop through each number `x` in `nums` and:
   - Increment the count at `d[x]` to indicate the potential start point of numbers in the range `[x-k, x+k]` to be adjusted to `x`.
   - Decrement the count at `d[x + k + 2 + 1]` to indicate the end point of the range.
4. Initialize `ans`, which will store the maximum beauty of the array, and `s`, which will be used as the running sum to apply the effect of the difference array when traversing it.
5. Traverse the difference array `d`, accumulating the total effect in `s` at each step, and update `ans` if the new sum `s` is greater than the current `ans`. This traversal allows us to collect the effective total count for each number as though all operations have been applied.
6. Finally, return `ans`, which is now the longest length of a subsequence that can be created through the allowed modifications — the maximum beauty of the array `nums`.

To sum up, this solution is based on the insight that we can use a difference array to efficiently simulate all the operations and directly compute the result instead of performing each operation individually.

Example Walkthrough

Let's consider a small example to illustrate the solution approach.

Suppose we have an array `nums = [1, 3, 4]` and `k = 1`.

First, we follow these steps to maximize the beauty of the array:

1. Calculate the size for the difference array. With a `max(nums)` of `4`, `k * 2` is `2`, so `m` is `4 + 2 + 2` which gives us `8`.
2. Initialize the difference array `d` to be size `8`, filled with zeros: `d = [0, 0, 0, 0, 0, 0, 0, 0]`.
3. Loop through each number in `nums` and update the difference array:
   - For `x = 1`, increment `d[1]` and decrement `d[1 + 1 + 2 + 1]`, so `d` becomes `[0, 1, 0, -1, 0, 0, 0, 0]`.
   - For `x = 3`, increment `d[3]` and decrement `d[3 + 1 + 2 + 1]`, so `d` becomes `[0, 1, 0, 0, 0, -1, 0, 0]`.
   - For `x = 4`, increment `d[4]` and decrement `d[4 + 1 + 2 + 1]`, so `d` becomes `[0, 1, 0, 0, 1, -1, 0, -1]`.
4. Initialize `ans` to `0` and `s` to `0`.
5. Traverse the difference array `d`, updating `ans` as we accumulate `s`:
   - `s = 0` → `d[0] = 0`, ans remains `0`.
   - `s = 0 + 1` → `d[1] = 1`, update `ans` to `1`.
   - `s = 1 + 0` → `d[2] = 0`, ans remains `1`.
   - `s = 1 + 0` → `d[3] = 0`, ans remains `1`.
   - `s = 1 + 1` → `d[4] = 1`, update `ans` to `2`.
   - `s = 2 - 1` → `d[5] = -1`, ans remains `2`.
   - `s = 1 + 0` → `d[6] = 0`, ans remains `2`.
   - `s = 1 - 1` → `d[7] = -1`, ans remains `2`.
6. Finally, return `ans` which is `2`. This means the largest length of a subsequence where all elements are equal after the operations is `2`.

From the walkthrough, we can see with `nums = [1, 3, 4]` and `k = 1`, the maximum beauty that can be achieved is `2`, since we can transform the array to `[3, 3, 3]` by incrementing the first element `1` by `2` (which is within the range `[1 - 1, 1 + 1]`) and decrementing the last element `4` by `1` (which is within the range `[4 - 1, 4 + 1]`), resulting in two `3`s which creates the longest subsequence of equal numbers.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def maximumBeauty(self, flowers: List[int], steps: int) -> int:
        # Calculate the maximum constant for adjusting the range
        max_flower = max(flowers) + steps * 2 + 2

        # Initialize a difference array with the same range
        diff_array = [0] * max_flower

        # Construct the difference array to perform range updates
        for flower_count in flowers:
            diff_array[flower_count] += 1
            diff_array[flower_count + steps * 2 + 1] -= 1

        # Initialize the variables for tracking the maximum beauty and the running sum
        max_beauty = running_sum = 0

        # Apply the difference array technique to find the total at each point
        for count_diff in diff_array:
            running_sum += count_diff
            max_beauty = max(max_beauty, running_sum)

        # Return the maximum beauty value found
        return max_beauty
```

Java

```java
import java.util.Arrays;

public class Solution {

    public int maximumBeauty(int[] flowers, int additions) {
        // Calculate the maximum value after all possible additions.
        // We add extra space for fluctuations in the count within the k range.
        int maxValue = Arrays.stream(flowers).max().getAsInt() + additions * 2 + 2;
        int[] delta = new int[maxValue];

        // Build the difference array using the number of additions possible adding and subtracting at the range ends.
        for (int flower : flowers) {
            delta[flower]++;
            delta[flower + additions + 2 + 1]--;
        }

        int maxBeauty = 0; // This will hold the maximum beauty calculated so far.
        int runningSum = 0; // This will be used to compute the running sum from the difference array.

        // Compute the running sum and find the maximum value.
        for (int value : delta) {
            runningSum += value;
            maxBeauty = Math.max(maxBeauty, runningSum);
        }

        // Return the computed maximum beauty of the bouquet.
        return maxBeauty;
    }
}
```

C++

```cpp
class Solution {
public:
    int maximumBeauty(vector<int>& flowers, int k) {
        // Calculate the maximum value in the vector and set the range accordingly.
        // considering the range needs to be large enough to account for adding k to both sides.
        int maxValue = *max_element(flowers.begin(), flowers.end()) + k * 2 + 2;

        // Create a differential array with the size based on calculated range.
        vector<int> diffArray(maxValue, 0);

        // Iterate over the original array and update the differential array accordingly.
        for (int flower : flowers) {
            // Increase the count at the start of the window (flower's position).
            diffArray[flower]++;
            // Decrease the count at the end of the window.
            diffArray[flower + k + 2 + 1]--;
        }

        // Initialize variables to track the sum and the maximum beauty so far.
        int currentSum = 0, maxBeauty = 0;

        // Iterate over the differential array and calculate the prefix sum,
        // which gives us the running count of flowers.
        for (int count : diffArray) {
            // Update the running sum with the current count.
            currentSum += count;
            // Calculate the maximum beauty seen so far by taking the maximum of the
            // current running sum and the previous maxBeauty.
            maxBeauty = max(maxBeauty, currentSum);
        }

        // Return the maximum beauty that can be achieved.
        return maxBeauty;
    }
};
```

TypeScript

```typescript
function maximumBeauty(nums: number[], operationsAllowed: number): number {
    // Calculate a maximum boundary to create an array that is big enough to hold all possible values after operations.
    const maxBoundary = Math.max(...nums) + operationsAllowed * 2 + 2;
    // Initialize difference array to hold the frequency of numbers up to the maximum boundary.
    const differenceArray: number[] = new Array(maxBoundary).fill(0);

    // Iterate over the input array and update the difference array based on the allowed operations.
    for (const num of nums) {
        // For each number, increment the count at the number's index in the difference array.
        differenceArray[num]++;
        // Decrement the count at the index that is 'operationsAllowed' doubled and one past the current number.
        differenceArray[num + operationsAllowed * 2 + 1]--;
    }

    let maxBeauty = 0;  // Store the maximum beauty value found.
    let sum = 0;        // Accumulator to store the running sum while iterating over the difference array.

    // Iterate over the difference array to find the maximum accumulated frequency (beauty).
    for (const frequency of differenceArray) {
        // Update the running sum by adding the current frequency.
        sum += frequency;
        // Update the maximum beauty value if the current running sum is greater than the previous maximum.
        maxBeauty = Math.max(maxBeauty, sum);
    }

    // Return the maximum beauty value found.
    return maxBeauty;
}
```

```python
from typing import List

class Solution:
    def maximumBeauty(self, flowers: List[int], steps: int) -> int:
        # Calculate the maximum constant for adjusting the range
        max_flower = max(flowers) + steps * 2 + 2

        # Initialize a difference array with the same range
        diff_array = [0] * max_flower

        # Construct the difference array to perform range updates
        for flower_count in flowers:
            diff_array[flower_count] += 1
            diff_array[flower_count + steps * 2 + 1] -= 1

        # Initialize the variables for tracking the maximum beauty and the running sum
        max_beauty = running_sum = 0

        # Apply the difference array technique to find the total at each point
        for count_diff in diff_array:
            running_sum += count_diff
            max_beauty = max(max_beauty, running_sum)

        # Return the maximum beauty value found
        return max_beauty
```

Time and Space Complexity

The provided code implements a function to determine the maximum beauty of an array after performing certain operations. The key operations to analyze are the loops and the creation of the list `d`.

Time Complexity:

- The first part of the analysis involves the line `m = max(nums) + k * 2 + 2`. This operation is `O(n)` where `n` is the number of elements in `nums`, as it requires a pass through the entire list to determine the maximum value.
- The creation of the list `d` with a length of `m` is `O(m)`. This accounts for the space required to store the frequency differences.
- The loop that populates the list `d` iterates over each element in `nums`, so this part is `O(n)`.
- Inside this loop, there are constant-time operations `O(1)` (incrementing and decrementing the values).
- The final loop iterates through the list `d` which has a length of `m`. This loop has a time complexity of `O(m)` as it needs to visit each element in `d` to accumulate the sum and compute the maximum.

Combining these complexities, we get `O(n) + O(m) + O(n) + O(m)` which simplifies to `O(n + m)`, with `m` being proportional to `max(nums) + 2 * k`.

Space Complexity:

- The list `d` with length `m` is the most significant factor in space complexity, making it `O(m)`, as it depends on the value of `k` and the maximum value in `nums`.
- The variables `ans` and `s` are of constant space `O(1)`.

Hence, the final time complexity is `O(n + m)` and the space complexity is `O(m)`, taking into account the size of the input `nums` and the range of values derived from it combined with `k`.