798. Smallest Rotation with Highest Score

Hard Array Prefix Sum

Problem Description

The problem provides an array `nums` and asks to find the best rotation index `k` that yields the highest score after rotating the array. When an array is rotated `k` times, the elements are shifted in such a way that the element at index `k` moves to index `0`, the element at index `k+1` moves to index `1`, and so on until the element at index `0` moves to index `n-k` where `n` is the length of the array.

The score is calculated based on the condition that each element that is less than or equal to its new index after the rotation gets one point. The goal is to determine the rotation index `k` that maximizes this score. If there are multiple rotation indices yielding the same maximum score, the smallest index `k` should be returned.

For example, take `nums = [2,4,1,3,0]`. When we rotate `nums` by `k = 2`, the result is `[1,3,0,2,4]`. In this rotated array, the elements at indices `2`, `3`, and `4` (which are `0`, `2`, and `4` respectively) are less than or equal to their indices, thereby each contributing one point to the score. Therefore, the total score is `3`.

Intuition

The intuition behind the given solution is to leverage differences in the array to efficiently calculate scores for all possible rotations without performing each rotation explicitly. A key observation is that when rotating the array, certain elements will start to contribute to the score when they are moved to particular positions. Similarly, they'll stop contributing to the score when rotated past a certain point.

The solution uses a difference array `d`, which tracks changes in the score potential as the array is virtually rotated. The elements at `d[k]` represents the net change in score when rotating from `k-1` to `k`. This allows us to compute the score for each rotation using a running total rather than recalculating the entire score each time.

For each value in the `nums` array, we identify the range of rotation indices for which that value will contribute to the score. This is done by using modular arithmetic to ensure indices wrap around properly:

- `i - (i + 1) % n` is the first index that the number at the current index `i` will not contribute to the score after rotation, so we increment the score change at index `i`.
- `r = (n + i + 1 - v) % n` is the first index where the number will start to contribute to the score, so we decrement the score change at index `r`.

After populating the difference array, we can iterate through it, keeping a running total (`s`). Whenever we get a running total that is higher than the current maximum (`mx`), we update the maximum and store the index `k`, which would be our answer.

Thus, the solution efficiently finds the best rotation that yields the highest score by intelligently tracking when each number starts and stops contributing to the score, rather than performing and checking each rotation one by one.

Solution Approach

The solution provided uses a smart approach with help of a difference array to track the change in score with each possible rotation. Here's a breakdown of the implementation:

1. **Initialize variables**: We start by initializing a few variables. `n` stores the length of the array, `mx` is used to keep track of the maximum score encountered so far (initialized to -1 since the score can't be negative), and `ans` holds the rotation index that yields the maximum score (initialized to `n` to ensure we return the smallest index in case of a tie). We also initialize a difference array `d` of size `n`, with all elements set to 0.
2. **Populate the difference array**: Going through each element in `nums` with their index `i` and value `v`, we calculate two values `l` and `r` using modular arithmetic to wrap around the array.
   - `l = (i - (i + 1) % n)`: This value represents the rotation index right after the element `v` at index `i` would stop contributing a point. Therefore, we increment the score change at this index with `d[l] += 1`.
   - `r = (n + i + 1 - v) % n`: This value represents the rotation index when the element `v` starts contributing a point. We then decrement the score change at this index with `d[r] -= 1`.
3. **Find the best rotation**: To find the rotation yielding the highest score, we track a running total `s`. We iterate through the difference array `d`, adding `d[k]` to `s` for each index `k`. Whenever the running total exceeds the current maximum score `mx`, we update `mx` with this new score and set `ans` to the current index `k`. This process takes advantage of the fact that `d[k]` represents the net change in score when "rotating" from `k-1` to `k`, allowing us to efficiently calculate the score for each potential rotation.
4. **Return the optimal rotation index**: After iterating through the entire difference array and updating `mx` and `ans` accordingly, we finally return `ans`. This index corresponds to the smallest rotation index `k` that achieves the maximum score.

This algorithm leverages the properties of *difference arrays* to calculate the cumulative effect of each rotation on the score without exhaustive computation, resulting in an efficient solution. The use of modular arithmetic ensures that index values are correctly wrapped around the boundaries of the array, which is necessary for the rotations.

Example Walkthrough

Let's go through a small example to illustrate the solution approach using the array `nums = [2, 3, 1, 4, 0]`.

1. **Initialize variables**: The length of the array `n` is 5. We initialize `mx` as -1, `ans` as 5, and a difference array `d` of size 5 with all elements as 0.
2. **Populate the difference array**: We process each element to determine when it starts and stops contributing to the score.
   - For `nums[0] = 2`, `l = (0 + 1) % 5 = 1` and `r = (5 + 0 + 1 - 2) % 5 = 4`. So `d[1]` gets incremented by 1 and `d[4]` gets decremented by 1.
   - Repeat the process for all other elements in `nums`:
     - `nums[1] = 3`: `l = 2`, `r = 0` (since after 3 rotations, index 1 will be at position 3) → `d[2]++, d[0]--`
     - `nums[2] = 1`: `l = 3`, `r = 3` → `d[3]++, d[3]--`
     - `nums[3] = 4`: `l = 4`, `r = 1` → `d[4]++, d[1]--`
     - `nums[4] = 0`: `l = 0`, `r = 4` → `d[0]++`, no change at `r` since it wraps around to the same position.
3. **Find the best rotation**: We now have the difference array `d = [1, 0, 0, 1, -1]` after processing all elements. Starting with a score `s = 0`, we iterate through `d` and track the running total.
   - At `k = 0`, the score `s` becomes `s = d[0] = 0 + 1 = 1`. Since `s` is greater than `mx` (-1), we update `mx = 1` and `ans = 0`.
   - At `k = 1`, the score does not change (`s = d[1] = 0`), and since `ans` is not exceeded, we do not update `mx` or `ans`.
   - At `k = 2`, again, the score does not change (`s = d[2] = 0`).
   - At `k = 3`, the score increases to `s = d[3] = 1 + 1 = 2`. We update `mx = 2`, and `ans = 3`.
   - Finally, at `k = 4`, the score decreases to `s = d[4] = 2 - 1 = 1`. Since `mx` is not exceeded, we do not update it.
4. **Return the optimal rotation index**: After processing the entire difference array, we find that the maximum score is `mx = 2` achieved at `ans = 3`. Therefore, we return `ans = 3` as the optimal rotation index for the maximum score.

Using this example, we've shown how the difference array helps avoid calculating each possible rotation's score from scratch, instead using differences to achieve an efficient computation of the best rotation index.

Solution Implementation

Python

```python
class Solution:
    def bestRotation(self, nums: List[int]) -> int:
        n = len(nums)
        max_score, best_k = -1, 0  # Initialize max score and best k to hold the highest score and best k value
        delta = [0] * n  # Create a delta array to keep track of score changes

        # Walk through each number in the array to calculate score changes
        for index, value in enumerate(nums):
            left = (index + 1) % n
            right = (n + index + 1 - value) % n

            # Increment score for this position and decrement at the position where it starts losing scores
            delta[left] += 1
            delta[right] -= 1
            if left > right:  # If the range wraps around, we increment the first element too
                delta[0] += 1

        score = 0  # Keep track of the aggregate score so far
        # Iterate through delta array to find the k with the maximum score
        for k, change in enumerate(delta):
            score += change  # Update the score by the current change
            # If the current score is greater than max_score, update it and record the current k
            if score > max_score:
                max_score = score
                best_k = k

        return best_k  # Return the k index that gives the maximum score
```

Java

```java
class Solution {
    public int bestRotation(int[] nums) {
        int n = nums.length; // Get the length of the array 'nums'
        int[] delta = new int[n]; // Initialize an array to track the change in scores for each rotation 'k'

        for (int i = 0; i < n; ++i) {
            // Calculate the lower bound (inclusive) of the range for which the number nums[i] gains a score if 'k' is the rotation.
            int lower = (i + 1) % n;
            // Calculate the upper bound (exclusive) of the range for which the number nums[i] gains a score if 'k' is the rotation.
            int upper = (n + i + 1 - nums[i]) % n;
            // Increment the score change at the lower bound
            ++delta[lower];
            // Decrement the score change at the upper bound as it is exclusive
            --delta[upper];

            // If after rotating, the number wraps around the array, decrease score change at index 0
            if (lower > upper) {
                --delta[0];
            }
        }

        int maxScore = -1; // Initialize maxScore to track the maximum score
        int score = 0; // Score for the current rotation 'k'
        int bestRotationIndex = 0; // Initialize bestRotationIndex to track the best rotation which gives maxScore

        for (int k = 0; k < n; ++k) {
            // Accumulate score changes for rotation 'k'
            score += delta[k];
            // If the current rotation score is greater than maxScore, update maxScore and bestRotationIndex
            if (score > maxScore) {
                maxScore = score;
                bestRotationIndex = k;
            }
        }

        // Return the index 'k' corresponding to the best rotation of the array that maximizes the score
        return bestRotationIndex;
    }
}
```

C++

```cpp
class Solution {
public:
    int bestRotation(vector<int>& nums) {
        int n = nums.size();            // Store the size of the input vector nums
        int maxScore = -1;              // Initialize the variable to keep track of the maximum score
        int bestK = n;                  // Initialize the variable to store the best rotation index k
        vector<int> diffs(n);           // Initialize a difference array to track score changes

        for (int i = 0; i < n; ++i) {
            int left = (i + 1) % n;                         // Calculate the left index for score increment
            int right = (n + i + 1 - nums[i]) % n;          // Calculate the right index for score decrement

            diffs[left]++;                                  // Increment score at position 'left'
            diffs[right]--;                                 // Decrement score at position 'right'
            if (left > right) {                             // If the range wraps around the array
                diffs[0]++;                                 // Increment score at the beginning of the array
            }
        }

        int score = 0;                  // Initialize the score variable for the accumulative score sum
        for (int k = 0; k < n; ++k) {
            score += diffs[k];          // Update the score by adding the current difference

            if (score > maxScore) {     // Update maxScore and bestK if a higher score is found
                maxScore = score;
                bestK = k;
            }
        }

        // Return the best rotation index 'k' with the highest score
        return bestK;
    }
};
```

TypeScript

```typescript
// Global function that returns the best rotation for the maximum score
function bestRotation(nums: number[]): number {
    const n: number = nums.length; // Store the length of the input array nums
    let maxScore: number = -1;     // Initialize the variable for tracking the maximum score
    let bestK: number = n;         // Initialize the variable to store the best rotation index k
    let diffs: number[] = new Array(n).fill(0); // Initialize a difference array with zeros to track score changes

    for (let i = 0; i < n; ++i) {
        let left: number = (i + 1) % n;                 // Calculate the left index for score increment
        let right: number = (n + i + 1 - nums[i]) % n; // Calculate the right index for score decrement

        diffs[left]++;  // Increment score at position 'left'
        diffs[right]--; // Decrement score at position 'right'
        if (left > right) {  // If the range wraps around the array
            diffs[0]++;  // Increment score at the beginning of the array
        }
    }

    let score: number = 0; // Initialize the score variable for the accumulative score sum
    for (let k = 0; k < n; ++k) {
        score += diffs[k];  // Update the score by adding the difference at index k

        if (score > maxScore) {  // If a higher score is found, update maxScore and bestK
            maxScore = score;
            bestK = k;
        }
    }

    // Return the best rotation index 'k' with the highest score
    return bestK;
}

// The function can then be used as follows:
// let nums = [some array of numbers];
// let result = bestRotation(nums);


class Solution:
    def bestRotation(self, nums: List[int]) -> int:
        n = len(nums)
        max_score, best_k = -1, 0  # Initialize max score and best k to hold the highest score and best k value
        delta = [0] * n  # Create a delta array to keep track of score changes

        # Walk through each number in the array to calculate score changes
        for index, value in enumerate(nums):
            left = (index + 1) % n
            right = (n + index + 1 - value) % n

            # Increment score for this position and decrement at the position where it starts losing scores
            delta[left] += 1
            delta[right] -= 1
            if left > right:  # If the range wraps around, we increment the first element too
                delta[0] += 1

        score = 0  # Keep track of the aggregate score so far
        # Iterate through delta array to find the k with the maximum score
        for k, change in enumerate(delta):
            score += change  # Update the score by the current change
            # If the current score is greater than max_score, update it and record the current k
            if score > max_score:
                max_score = score
                best_k = k

        return best_k  # Return the k index that gives the maximum score
```

Time and Space Complexity

The given Python code implements a function to determine the best rotation for an array such that the number of elements `nums[i]` that are not greater than their index `i` is maximized when the array `nums` is rotated.

Time Complexity:

The time complexity of the function is O(`n`), where `n` is the length of the array `nums`.

1. The loop to calculate the difference array: This loop runs for `n` iterations, where `n` is the length of the array `nums`. Inside the loop, we calculate the changes for each rotation using modulo operations and update the difference array (`d`). Each update is a constant time operation. Therefore, this step takes O(`n`) time.
2. The loop to find the best rotation: This loop also runs for `n` iterations. It iterates over the difference array, cumulatively adding the differences (`k += t`) to find the score for each rotation, and keeps track of the maximum score and the corresponding index (rotation). Since each iteration of the loop does a constant amount of work, this step is also O(`n`).

Thus, the overall time complexity of the function is O(`n`).

Space Complexity:

The space complexity of the function is O(`n`).

- We maintain a difference array (`d`) of the same length as the `nums` array, which takes O(`n`) space.
- Aside from the difference array, only a constant amount of extra space is used for variables to keep track of the maximum score, current score, and the best rotation.

Therefore, the total space used by the function is determined by the size of the difference array, which is O(`n`).