2936. Number of Equal Numbers Blocks

Medium Array Binary Search Divide and Conquer

Problem Description

In this problem, you are presented with an array of integers, `nums`, where all occurrences of any given value are situated next to each other. It means that the array is segmented into blocks of equal values. The objective is to count how many such blocks exist in the array. Since `nums` can be very large, you are provided with a `BigArray` class with two functions to interact with the array: `at(index)` to get the value at a certain index, and `size()` to get the total size of the array.

You are required to determine the number of maximal blocks of equal values within this array. A maximal block is a continuous subarray where all elements are the same and cannot be extended further while maintaining the same value.

Intuition

To solve this problem, we need to find a way to count contiguous blocks of equal values within a potentially very large array, without scanning every single element. The key observation is that, due to the property of the array (all occurrences of a value being adjacent), once we find a new value different from the current one, we know a new block has started.

Starting from the first element, we can search for the boundary of each block. Since performing a linear search may not be efficient for a large dataset, we can take advantage of the block property and use binary search. Binary search will efficiently find the next index where a different value occurs, which also marks the end of the current block.

Here's how we can arrive at the solution: We iterate through the array elements using a loop. At each step, we look for the first index on the right side of the current position that has a different value than the value at the current position. This marks the end of a block. We can speed up this search using binary search as follows:

1. From the current index, we check the next value. If it's different, this is already the end of a block.
2. If it's not different, we perform a binary search between the current index and the end of the array to find the first position where the value differs from the current block's value. This is done with the help of a key function provided to `bisect_left`, which uses the `at` method of the `BigArray`.
3. We update the current index to the boundary found (the index with a different value) to continue searching for the next block.

We repeat this process until we reach the end of the array and in the process, maintain a count of blocks found.

Solution Approach

The solution employs a binary search strategy to efficiently find the right boundary of each block in the array. Let's break down the implementation steps of the solution using algorithms and data structures:

1. Initialize an index `i` at 0 and a counter `ans` to 0 to keep the tally of blocks encountered.
2. Loop until `i` is less than the size `n` of the `BigArray`.
3. For each iteration of the loop:
   - Increment the block counter `ans` since the element at index `i` starts a new block.
   - Check the value at the current index `x = nums.at(i)`.
   - Then there are two possible scenarios: a. If the element immediately next to the current one (`i + 1`) is different from `x`, it implies the current block is of size 1. So, increment `i` by 1. b. If the next element is not different, then perform a binary search to find the left boundary of the next block where the value differs from `x`. Use the `bisect_left` method combined with a key function that leverages `nums.at(j)` to perform the binary search efficiently. c. Update the index `i` to the boundary index found using the binary search. This skips the entire block of identical elements in one step.
4. Return the counter `ans` as the final result, which represents the total number of blocks in the array.

The use of binary search significantly optimizes the performance of the solution, especially when dealing with large data sets where linear search would be too slow. It reduces the potential searches from linear time complexity (`O(n)`) to logarithmic time complexity (`O(log n)`), for each block search.

When using the `bisect_left` function, the `key` function provided is essential. It acts as a predicate to tell `bisect_left` when to stop and find an index `j` where the value at `j` no longer equals `x` (`nums.at(j) != x`). This binary search is within the range from the current index `i` to the end of the array `n`.

Thus, the algorithm's efficiency stems from its ability to jump over contiguous segments of repeated values and count blocks without individually checking each element within a block.

Example Walkthrough

Let's consider a `BigArray` represented by `nums` which contains the following array of integers:

```
nums: [2, 2, 3, 3, 3, 6, 6, 6, 6, 7, 7, 10]
```

We want to count the number of maximal blocks of equal values. Here's a step-by-step walkthrough of the solution:

1. Initialize `i` to index 0 and `ans` (the block counter) to 0. The array size `n` is 12.
2. Since `i < n`, we enter the loop for the first time.
3. We increment `ans` to 1 because `nums.at(i)` (which is 2 now) starts a new block.
4. The next element `nums.at(i+1)` is also 2, so we perform a binary search to find the left boundary of the next block. In this case, binary search finds that the value changes at index 2. We update `i` to 2.
5. We're now looking at the second block starting with a value of 3. Increment `ans` to 2.
6. The next element `nums.at(i+1)` is 3, so once again, we perform a binary search to find the next value change. This time it happens at index 5, and `i` is updated to 5.
7. Repeat these steps for the values of 6, 7, and finally 10.
   - For the block of 6's, `ans` becomes 3, and `i` jumps to 9.
   - Then for 7's, `ans` increases to 4, and `i` goes to 11.
   - Finally, for the solo value of 10, `ans` becomes 5, and since there are no more elements, the loop ends.

The final count `ans` is 5, which means there are 5 blocks of equal values in the provided array `nums`.

Solution Implementation

Python

```python
# Definition for BigArray remains unchanged.
# class BigArray:
#     def at(self, index: int) -> int:
#         pass
#     def size(self) -> int:
#         pass

from bisect import bisect_left
from typing import Optional

class Solution:
    def count_blocks(self, nums: Optional["BigArray"]) -> int:
        """Counts the number of contiguous blocks of identical elements in the nums BigArray."""
        index, size = 0, nums.size()  # Initialize the index and get the size of BigArray.
        block_count = 0  # Initialize the count of contiguous blocks.

        # Iterate through the array.
        while index < size:
            block_count += 1  # Increment block count for each new block found.
            element = nums.at(index)  # Get the element at the current index.

            # Condition to check if we are within bounds and if the next element is different.
            if index + 1 < size and nums.at(index + 1) != element:
                index += 1
            else:
                # Use binary search to find the index of the next different element.
                # bisect_left will find the leftmost value exactly matching the key.
                # The range object is used here as a sequence input for the bisect_left function.
                index += bisect_left(range(index, size), True, key=lambda j: nums.at(j) != element)

        return block_count  # Return the total number of blocks.

# Additional notes:
# The `bisect_left` function is used to find the first index where `nums.at(j) != element` is True.
# The `range(index, size)` is given to the `bisect_left` as a sequence over which it performs binary search.
# The lambda function is used as a key to the bisect_left function to decide if the condition is met.
```

Java

```java
class Solution {

    /**
     * This method counts the number of blocks with the same value in a BigArray.
     *
     * @param nums A BigArray object which contains the elements in which we need to find blocks.
     * @return The number of blocks with the same value.
     */
    public int countBlocks(BigArray nums) {
        int count = 0; // Initialize block count to zero

        // Iterate over the elements of the BigArray starting from the first index
        for (long i = 0, totalSize = nums.size(); i < totalSize; ++count) {
            // Find the next starting index of a 'new' value block, increasing block count
            i = findNextBlockStart(nums, i, totalSize);
        }

        // Return the total number of blocks found
        return count;
    }

    /**
     * Finds the starting index of the next distinct value block.
     *
     * @param nums A BigArray object to search within.
     * @param start The index to start searching from.
     * @param totalSize The total size of the BigArray.
     * @return The starting index of the next block.
     */
    private long findNextBlockStart(BigArray nums, long start, long totalSize) {
        long end = totalSize;  // Set the end index to the total size of the BigArray
        int currentValue = nums.at(start);  // Get the value of the start index

        // Perform binary search to find the end index of the current value block
        while (start < end) {
            long mid = (start + end) >> 1; // Find the mid index by right shifting the sum of start and end

            // If the mid index has a different value, that's a potential new block, so move end to mid
            if (nums.at(mid) != currentValue) {
                end = mid;
            } else {
                // Otherwise, the current block continues, so move start one index past mid
                start = mid + 1;
            }
        }

        // Return the starting index of the next block (after the end of the current block)
        return start;
    }
}
```

C++

```cpp
/**
 * Definition for BigArray.
 * The class provides an interface for arrays with a potentially very large number of elements.
 * It includes methods to construct a BigArray, retrieve a value at a given index, and get the size of the array.
 */
class BigArray {
public:
    // Constructor that initializes the BigArray with the provided elements.
    BigArray(vector<int> elements);

    // Returns the element at the specified index.
    int at(long long index);

    // Returns the total size (number of elements) of the BigArray.
    long long size();
};


class Solution {
public:
    // Counts the number of contiguous blocks of identical elements within the BigArray.
    int countBlocks(BigArray* nums) {
        int blockCount = 0; // Initialize the count of blocks
        using ll = long long; // Alias for long long to simplify typing
        ll n = nums->size(); // Get the total size of the BigArray

        // Lambda function to find the next boundary where the value changes in the BigArray.
        auto findNextBoundary = [&](ll leftIndex) {
            ll rightIndex = n; // Set the search's right boundary to the end of the array
            int currentValue = nums->at(leftIndex); // Get the value at the current leftIndex
            while (leftIndex < rightIndex) {
                ll midpoint = (leftIndex + rightIndex) >> 1; // Compute the midpoint for binary search
                if (nums->at(midpoint) != currentValue) {
                    // If the value at the midpoint is different, search to the left
                    rightIndex = midpoint;
                } else {
                    // If the value is the same, search to the right
                    leftIndex = midpoint + 1;
                }
            }
            return leftIndex; // Return the boundary where the value changes
        };

        // Iterate through the BigArray and count the contiguous blocks
        for (ll i = 0; i < n; ++blockCount) {
            // Each search returns the start of the next block, which becomes the new i
            i = findNextBoundary(i);
        }
        return blockCount; // Return the total number of contiguous blocks found
    }
};
```

TypeScript

```typescript
// Define the type for a BigArray for better understanding. This type should align with
// the provided information about the BigArray class in the initial code comments.
interface BigArray {
    at(index: number): number;
    size(): number;
}

/**
 * Counts the number of distinct continuous subarrays where the value is constant,
 * within a given BigArray.
 *
 * @param {BigArray} bigArray An instance of BigArray that contains the numbers
 *                            we want to analyze.
 *
 * @return {number} Returns the number of constant-value subarrays (blocks).
 */
function countBlocks(bigArray: BigArray | null): number {
    if (bigArray === null) {
        return 0; // If the input is null, there are no blocks to count.
    }

    const totalSize = bigArray.size();

    /**
     * Searches for the right boundary of a block with the same values that starts at index `left`.
     *
     * @param {number} left Left boundary index where the same value block starts.
     *
     * @return {number} Returns the non-inclusive right boundary index of the block.
     */
    const searchRightBoundary = (left: number): number => {
        let right = totalSize;
        const valueAtLeft = bigArray.at(left);

        // Binary search to find the non-inclusive right boundary index where the value changes.
        while (left < right) {
            const mid = left + Math.floor((right - left) / 2);
            if (bigArray.at(mid) !== valueAtLeft) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    let count = 0; // Initialize the count of blocks to 0.
    let currentIndex = 0; // Start from the first index of the BigArray.

    // Iterate until the currentIndex reaches the end of the BigArray.
    while (currentIndex < totalSize) {
        // Find the next block's right boundary and assign it as the new currentIndex.
        currentIndex = searchRightBoundary(currentIndex);
        count++; // Increment the count for each block found.
    }

    return count; // Return the total count of distinct blocks.
}


# Definition for BigArray remains unchanged.
# class BigArray:
#     def at(self, index: int) -> int:
#         pass
#     def size(self) -> int:
#         pass

from bisect import bisect_left
from typing import Optional

class Solution:
    def count_blocks(self, nums: Optional["BigArray"]) -> int:
        """Counts the number of contiguous blocks of identical elements in the nums BigArray."""
        index, size = 0, nums.size()  # Initialize the index and get the size of BigArray.
        block_count = 0  # Initialize the count of contiguous blocks.

        # Iterate through the array.
        while index < size:
            block_count += 1  # Increment block count for each new block found.
            element = nums.at(index)  # Get the element at the current index.

            # Condition to check if we are within bounds and if the next element is different.
            if index + 1 < size and nums.at(index + 1) != element:
                index += 1
            else:
                # Use binary search to find the index of the next different element.
                # bisect_left will find the leftmost value exactly matching the key.
                # The range object is used here as a sequence input for the bisect_left function.
                index += bisect_left(range(index, size), True, key=lambda j: nums.at(j) != element)

        return block_count  # Return the total number of blocks.

# Additional notes:
# The `bisect_left` function is used to find the first index where `nums.at(j) != element` is True.
# The `range(index, size)` is given to the `bisect_left` as a sequence over which it performs binary search.
# The lambda function is used as a key to the bisect_left function to decide if the condition is met.
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(m * log n)`, where `m` is the number of distinct elements in the `nums` `BigArray`, and `n` is the size of `nums`. This is because for every unique value `x` in the array, the code performs a binary search using `bisect_left` to find the next index where the value of `nums.at(j)` is different from `x`. Given that binary search has a time complexity of `O(log n)`, and this is done for `m` distinct elements, the total time complexity is `O(m * log n)`.

Space Complexity

The space complexity of the code is `O(1)` because the algorithm uses a fixed number of variables `i`, `n`, `ans`, and `x`, and does not create any auxiliary data structures that grow with the input size. This means that the space used remains constant regardless of the size of the input array `nums`.