1538. Guess the Majority in a Hidden Array

Medium Array Math Interactive

Problem Description

In this problem, we're given an array of integers called `nums`, where every integer is either 0 or 1. Direct access to this array is not permitted; instead, we have access to an API `ArrayReader` that provides two functions. The `query(a, b, c, d)` function can be used to query the array with four different indices (with the condition `0 <= a < b < c < d < ArrayReader.length()`) and returns the distribution of the four selected elements as follows:

- Returns `4` if all four elements are 0's or all are 1's.
- Returns `2` if three elements are the same (either three 0's and one 1, or three 1's and one 0).
- Returns `0` if there's an even split (two 0's and two 1's).

The `length()` function returns the size of the array.

Our goal is to find the index of any element with the most frequent value, which could be either 0 or 1. In case of a tie, where 0 and 1 are equally frequent, we return -1.

Intuition

To solve this problem, we start with the understanding that the `query()` function can give us an insight into the majority element by comparing the distribution of 0s and 1s among different subsets of four elements. Since we're allowed to call `query()` `2 * n` times where `n` is the array length, we have enough queries at our disposal to deduce the majority element's index.

The solution includes querying the first four elements to establish the initial distribution (let's call this the baseline distribution). Afterwards, we loop through the rest of the elements, using `query()` with the first three indices fixed and the fourth one changing to include each new index from the rest of the array. If the distribution matches the baseline, the new element shares the majority value with the first three; if it doesn't match, it's different.

The first element not included in the initial query (`index 4`) is specially treated by checking its distribution with the next 3 consecutive elements. This helps us in confirming whether the initially queried elements were a majority or a split.

After looping through all elements, we compare the count of elements that matched the baseline distribution (`i`) with those that didn't (`j`). The index that occurs more is the majority. If both counts are equal, we have a tie, and we return `-1`.

The intuition is based on iteratively narrowing down the possible indices for the majority element by comparing the distribution of subsets of the array using the `query()` function.

Solution Approach

The implementation of the solution approach involves using a simple iteration to leverage the `ArrayReader` API effectively. Here's a breakdown of each part of the code and the algorithm's data structures, or patterns used:

1. **Initialization:** The `n` variable is initialized with the size of the array using `reader.length()`. Two counters, `i` and `j`, are initialized to keep track of the counts of indices that match or do not match the initial baseline distribution. Variable `k` is initialized to store an index where the value is different from the baseline if such an index is found.
2. **Initial Query:** We perform the first `query(0, 1, 2, 3)` to establish a baseline distribution stored in the variable `x`. This tells us the distribution pattern of the first four elements.
3. **Iterative Comparison:** A `for` loop is used to iterate through the elements starting from index `4` to the last index `n-1`. In each iteration, we query a subset that includes the first three fixed indices (`0, 1, 2`) and the current index `i`.
   - If the return value of this query matches `x`, we increment counter `i`.
   - If the return value doesn't match `x`, we increment counter `j` and update `k` with the current index `i`.
4. **Special Cases:** After the loop, we need to verify the initial three elements (which are fixed in the previous `query()` calls) to ensure their distribution. This requires three additional queries, each omitting one of the first three indices alongside the fourth index `4`. This helps in confirming whether the initially queried indices were a majority.
   - The results of these queries are compared against a different baseline distribution stored in `y` which is obtained by `query(0, 1, 2, 4)`.
   - If the result of these comparisons is equal to `y`, increment `s` since the omitted element matches the initial query (suggesting it's a majority element); otherwise, increment `s` and update `k` respectively.
5. **Determine Majority Index:** Finally, we determine whether there is a majority by comparing the counts `s` and `k`.
   - If `s` equals `k`, we conclude there is a tie and return `-1`;
   - If `k` is greater, we return `k` which was part of the original subset and, therefore, in the majority.
   - If `s` is greater, the index `k` is returned, indicating that `k` is part of the frequent value.

Data structures used in this solution are primarily basic integer variables for counting. The pattern used is iterative comparisons with early stopping as soon as the majority element is confirmed. The algorithm doesn't require complex data structures as it makes effective use of the provided API to solve the problem.

Example Walkthrough

Let's assume we have a `nums` array of size 8, and we perform the following steps:

1. **Initialization:**
   - The array size `n` is 8 (determined using `reader.length()`).
   - We initialize two counters, `i` and `k`, to zero. Counter `i` will track indices matching the baseline distribution, while `k` tracks the others.
   - The `k` variable is initialized but not yet set.
2. **Initial Query:**
   - We run `query(0, 1, 2, 3)` and let's say it returns `4`. This suggests all elements are the same (all 0s or all 1s), so our baseline distribution `x` is `4`.
3. **Iterative Comparison:**
   - We iterate with a loop for indices `4` to `7` (the rest of the array).
   - We perform `query(0, 1, 2, 4)`. Assume it returns `4`, identical to our baseline, so we increment `i`.
   - Next, `query(0, 1, 2, 5)` might return `2`, unlike our baseline. We increment `j` and set `k` to `5`.
   - We continue with `query(0, 1, 2, 6)`, returning `4`, so we increment `i`.
   - Lastly, `query(0, 1, 2, 7)` returns `4`, incrementing `i` again.
4. **Special Cases:**
   - We then create a separate baseline `y` with `query(0, 1, 2, 4)` (we have already done this and know it's `4`).
   - We run `query(1, 2, 3, 4)` for special case handling, and if it returns `4`, we increment `s`—let's assume it does;
   - Next, `query(0, 2, 3, 4)` and if it returns `4`, we increment `s`—let's say it does;
   - Finally, `query(0, 1, 3, 4)` and if it returns `4`, we increment `s`—let's assume it also does.

At the end of this process, let's assume `s` is `4`, and `k` is `5`. We now decide the majority element:

5. **Determine Majority Index:**
   - Since `s` is greater than `k`, we return index `5` because it belongs to the subset that matched our initial baseline the most.

This example confirmed that the majority values were present in the initial subset, and indexes that matched this distribution were in the majority. If `k` had been greater, we would return `k`, which in this walkthrough was `5`. If `s` and `i` were equal, indicating an even split between 0s and 1s, we would return `-1`.

Python Solution

```python
class Solution:
    def guessMajority(self, reader: "ArrayReader") -> int:
        # Get the length of the array
        n = reader.length()

        # Initialize the counter for the two possible majorities and an index tracker
        majority_count, minority_count = 1, 0
        minority_index = 0

        # Perform the initial query to establish a baseline comparison
        baseline_query = reader.query(0, 1, 2, 3)

        # Iterate through the rest of the array starting from index 4
        for i in range(4, n):
            # Query with the new index and compare to the baseline
            if reader.query(0, 1, 2, i) == baseline_query:
                majority_count += 1
            else:
                minority_count += 1
                minority_index = i  # Store the index of the first occurrence in minority

        # Perform additional queries to determine the first three elements' majority/minority status
        # These queries involve the element at index 4 to compare with the baseline value
        initial_queries_comparison = reader.query(1, 2, 3, 4)
        for i in range(3):
            if reader.query(i, (i + 1) % 3, (i + 2) % 3, 4) == initial_queries_comparison:
                majority_count += 1
            else:
                minority_count += 1
                minority_index = i  # Update the index of the first occurrence in minority

        # If the counts are the same, we return -1, as there is no majority
        if majority_count == minority_count:
            return -1

        # The element at index 3 has a different status compared to the first three elements due to the setup
        # Hence, if the majority count is greater, the index of the minority is returned; otherwise, 3 is returned
        return minority_index if majority_count > minority_count else 3
```

Java Solution

```java
class Solution {
    public int guessMajority(ArrayReader reader) {
        // Get the length of the array
        int arrayLength = reader.length();

        // Initial query to determine the pattern to compare against
        int initialQueryResult = reader.query(0, 1, 2, 3);

        // 'sameMajorityCount' is the count of indices following the initial pattern
        // 'differentMajorityCount' is the count of indices that differ from the initial pattern
        int sameMajorityCount = 1, differentMajorityCount = 0;

        // 'differentIndex' keeps track of the index where a different pattern was first noticed
        int differentIndex = 0;

        // Check rest of the elements with the initial ones except the last three
        for (int i = 4; i < arrayLength; ++i) {
            if (reader.query(0, 1, 2, i) == initialQueryResult) {
                sameMajorityCount++; // Same pattern detected, increment count
            } else {
                differentMajorityCount++; // Different pattern detected, increment count and update the index
                differentIndex = i;
            }
        }

        // Perform queries swapping out one of the first three elements with the fourth one
        // This checks the pattern of the last three indices
        int subsequentQueryResult = reader.query(1, 2, 3, 4);
        for (int i = 0; i < 3; ++i) {
            if (reader.query(i, (i + 1) % 3, (i + 2) % 3, 4) == subsequentQueryResult) {
                if (queryResult == subsequentQueryResult) {
                    sameMajorityCount++; // Increment count if the same as the subsequent pattern
                } else {
                    differentMajorityCount++; // Increment different count and update index with the swapped element
                    differentIndex = i;
                }
            }
        }

        // If counts are equal, no unique majority index exists
        if (sameMajorityCount == differentMajorityCount) {
            return -1;
        }

        // Return the index which has the majority pattern (most occurrences)
        // If the counts of the same pattern is greater, return the constant 3 as per initial result
        // If counts of the different pattern is greater, return the index where the difference was first noted
        return sameMajorityCount > differentMajorityCount ? 3 : differentIndex;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int guessMajority(ArrayReader& reader) {
        // Get the length of the array
        int n = reader.length();

        // Query the first four elements to get a baseline comparison
        int firstSubsetResult = reader.query(0, 1, 2, 3);

        // Initialize the count for the majority and minority elements
        int majorityCount = 1; // We start with 1 since we know the first four elements are the baseline
        int minorityCount = 0;

        // Keep track of the latest minority element index
        int minorityIndex = 0;

        // Compare the rest of the array elements starting from the fifth element
        for (int i = 4; i < n; ++i) {
            // If the next element belongs to the majority, increase the majority count
            if (reader.query(0, 1, 2, i) == firstSubsetResult) {
                ++majorityCount;
            } else {
                // Otherwise, increase the minority count and update the minorityIndex
                ++minorityCount;
                minorityIndex = i;
            }
        }

        // Perform another query with 0, 1, 2 and a known different element (4 in this case)
        int secondSubsetResult = reader.query(0, 1, 2, 4);

        // Check the remaining three elements against the new query to determine if they belong to the majority or minority
        if (reader.query(0, 1, 2, 4) == secondSubsetResult) {
            ++majorityCount;
        } else {
            ++minorityCount;
            minorityIndex = 3; // Element 3 is different
        }
        if (reader.query(0, 2, 3, 4) == secondSubsetResult) {
            ++majorityCount;
        } else {
            ++minorityCount;
            minorityIndex = 1; // Element 1 is different
        }
        if (reader.query(1, 3, 4) == secondSubsetResult) {
            ++majorityCount;
        } else {
            ++minorityCount;
            minorityIndex = 2; // Element 2 is different
        }

        // If the majority and minority counts are the same, the result is ambiguous
        if (majorityCount == minorityCount) {
            return -1;
        }

        // If the majority count is greater, the answer is the initial subset index (3)
        // Otherwise, return the index of the minority element
        return majorityCount > minorityCount ? 3 : minorityIndex;
    }
};
```

Typescript Solution

```typescript
function guessMajority(reader: ArrayReader): number {
    // Retrieve the length of the array from the reader
    const arrayLength = reader.length();

    // Perform the initial query on the first four elements
    const firstQueryResult = reader.query(0, 1, 2, 3);

    // Initialize counter for the majority and minority elements, as well as the index of the first different element
    let majorityCount = 1;
    let minorityCount = 0;
    let firstDifferentIndex = 0;

    // Iterate over the rest of the elements, starting from the 5th element
    for (let i = 4; i < arrayLength; ++i) {
        // Compare each new element with the initial three elements to determine its type
        if (reader.query(0, 1, 2, i) === firstQueryResult) {
            ++majorityCount;
        } else {
            ++minorityCount;
            firstDifferentIndex = i;
        }
    }

    // Perform additional queries by excluding one element from the first four each time
    const secondQueryResult = reader.query(0, 1, 2, 4);
    if (reader.query(0, 1, 3, 4) === secondQueryResult) {
        ++majorityCount;
    } else {
        ++minorityCount;
        firstDifferentIndex = 0;
    }
    if (reader.query(0, 2, 3, 4) === secondQueryResult) {
        ++majorityCount;
    } else {
        ++minorityCount;
        firstDifferentIndex = 1;
    }
    if (reader.query(1, 2, 3, 4) === secondQueryResult) {
        ++majorityCount;
    } else {
        ++minorityCount;
        firstDifferentIndex = 2;
    }

    // If there is an equal number of majority and minority elements, there is no majority
    if (majorityCount === minorityCount) {
        return -1;
    }

    // The majority is determined by the element with more occurrences
    // If the majority is in the first four elements, we return 3, which points to the index 3 in the initial query
    // Otherwise, we return the index of the first element that was different from the initial majority
    return majorityCount > minorityCount ? 3 : firstDifferentIndex;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the provided `guessMajority` function primarily depends on the number of calls made to the `reader.query` method within a loop and outside of it.

- There is a fixed number of calls to `reader.query` made outside of the loop for the indices `0, 1, 2, 3`, and four additional calls to compare with index `4`.
- The loop iterates from index `4` to `n - 1` (where `n` is the length of the array), making one call to `reader.query` per iteration.
- Hence, the number of `query` calls is `4 (initial) + 4 (comparisons with index 4) + (n - 4)` for the loop.

Thus, the total number of `query` calls is `8 + (n - 4) = n + 4`. Given that each `query` call is assumed to have a constant time complexity, the time complexity of the `guessMajority` function is $O(n)$.

Space Complexity

The space complexity of the code is determined by the variables used in the `guessMajority` function.

- Constant extra space is used for the variables `a`, `b`, `x`, and `y`, independent of the input size `n`.
- No additional data structures that grow with the input size are used.

Therefore, the space complexity of the `guessMajority` function is $O(1)$, which means it requires constant space.