2113. Elements in Array After Removing and Replacing Elements

Medium Array Leetcode Link

Problem Description

The problem presents an array `nums` representing a sequence of integers and a dynamic process that alternately removes elements from the beginning of the array and restores them after the array has been emptied. This process repeats indefinitely. Explicitly, each minute, one element from the beginning of the array is removed until the array is empty. Subsequently, each minute, an element is appended to the end of the array in the same order they were removed until `nums` is restored to its original state.

The problem also provides a set of queries where each query is a pair consisting of a minute `time_j` and an index `index_j`. The task is to find out what element, if any, exists at the given index at the specific time. If the index at a given time exceeds the current length of the array (since elements may have been removed or not yet restored), the answer to that query is `-1`.

Intuition

The key to solving this problem lies in recognizing the cyclic nature of changes to the array. The array completes one cycle of changes every `2 * length(nums)` minutes: half of the cycle for removing elements and the other half for restoring them. Thus, we need to determine the state of the array at a given point in this cycle to answer the queries.

Firstly, since the array repeats the same pattern every `2 * n` minutes (`n` being the initial length of `nums`), we can simplify each query time `t` by using the modulus operator `t % (2 * n)` to find the equivalent time within the first cycle.

Next, we identify two distinct phases within a cycle:

- The **removal phase**: this lasts for the first `n` minutes of the cycle. In this phase, we can simply check whether the index `i` specified in the query is still within the bounds of the array after `t` elements are removed from the start. The current element can be directly accessed as `nums[i + t]`.
- The **restoration phase**: this starts after the removal phase and lasts until the end of the cycle (from minute `n` to minute `2 * n`). During this phase, we can find out if the element at `i` has been restored by checking if `i < t - n` (which means that the element at index `i` has already been appended back). In this case, the element at index `i` is simply `nums[i]`.

Combining these phases, we iterate over each query, compute the effective time and determine the answer accordingly. The insight to solve this problem efficiently is to understand the cyclical process and calculate the element position relative to the current phase of the cycle.

Solution Approach

To implement the solution, we must navigate through the queries to determine the values at specific times. Here's a step-by-step approach that is followed by the given solution:

1. **Initialization**: We initialize an array `ans` with the same number of elements as queries filled with `-1`. This array will store the answers to each query. The variable `n` holds the length of `nums`, and `m` holds the number of queries.
2. **Query Simplification**: For each query consisting of time `t` and index `i`, we first reduce `t` to its equivalent time within the first cycle by calculating `t %= 2 * n`. This is crucial since the array's behavior repeats every `2 * n` minutes. It avoids unnecessary repetition by focusing on a single cycle.
3. **Handling Queries**:
   - We loop through each query using `j` as the index and unpack each query into `t` and `i`.
   - We check if the current minute `t` falls within the removal phase (`t < n`). If true, we see if the queried index `i` is valid after removing `t` elements from the beginning. The check `i < n - t` assures there is still an element at the index after the removal. If this is the case, we retrieve the value `nums[i + t]`.
   - If the current minute `t` falls within the restoration phase (`t > n`), we perform a different check: `i < t - n`. This checks if the element at index `i` has been restored by this minute. If it has, we know that the element at index `i` is the same as `nums[i]` since we restore the elements in the same order they were removed.

These steps use simple iterations and conditions to determine each query's answer effectively. The algorithm's time complexity is O(m), as each query is handled in constant time, given that we have precomputed `n` (the length of `nums`). There are no complex data structures used apart from an array to store the results.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we have an array `nums = [1,2,3,4]` and the queries `[(3, 1), (0, 2), (5, 0), (10, 1)]`.

1. **Initialization**: Initialize an array to store answers `ans = [-1, -1, -1, -1]` (since we have four queries). Also, we have `n = 4` (the length of `nums`) and `m = 4` (the number of queries).
2. **Query Simplification**:
   - For the first query `(3, 1)`:
     - We simplify `t = 3 % (2 * 4) = 3`. The new query is `(3, 1)`.
   - For the second query `(0, 2)`:
     - Simplification is not needed since `t = 0`. The query remains `(0, 2)`.
   - For the third query `(5, 0)`:
     - `t = 5 % (2 * 4) = 5`. The new query is `(5, 0)`.
   - For the fourth query `(10, 1)`:
     - `t = 10 % (2 * 4) = 2`. The new query is `(2, 1)`.
3. **Handling Queries**:
   - For `(3, 1)`: We are in the removal phase because `3 < 4`. Is index `1` valid after removing 3 elements? `1 < 4 - 3`, yes. So `ans[0] = nums[1 + 3] = nums[4]`, but `4` is outside the index range for `nums`, hence `ans[0]` remains `-1`.
   - For `(0, 2)`: It's the beginning of the cycle. No elements have been removed yet, so `ans[1] = nums[2] = 3`.
   - For `(5, 0)`: We are in the restoration phase because `5 > 4`. Is index `0` restored after 5 minutes when 1 minute of restoration has passed? `0 < 5 - 4`, yes, so `ans[2] = nums[0] = 1`.
   - For `(2, 1)`: Still in the removal phase as `2 < 4`. Is index `1` valid after 2 have been removed? `1 < 4 - 2`, yes, so `ans[3] = nums[1 + 2] = nums[3] = 4`.

Final `ans` array after evaluating all queries is `[-1, 3, 1, 4]`. Each element of `ans` correlates to the results of the respective queries, suggesting that at those specific times and indices, these were the correct values in the array.

Python Solution

```python
class Solution:
    def elementInNums(self, nums: List[int], queries: List[List[int]]) -> List[int]:
        # Determine the length of the 'nums' list and the length of 'queries'.
        num_length, query_count = len(nums), len(queries)

        # Initialize the answer list with -1 for each query.
        answers = [-1] * query_count

        # Process each query and determine the element at the given index if it exists.
        for index, (shift, target_index) in enumerate(queries):
            # The actual shift is the remainder when 'shift' is divided by twice the 'num_length',
            # since a full rotation would bring the array back to the original state.
            actual_shift = shift % (2 * num_length)

            # Case 1: If the shift is less than the array length and the target index is within the range
            # after the shift, update the answer.
            if actual_shift < num_length and target_index < num_length - actual_shift:
                answers[index] = nums[target_index + actual_shift]

            # Case 2: If the shift is greater than the array length and the target index is within
            # the range before the shift, update the answer.
            elif actual_shift > num_length and target_index < actual_shift - num_length:
                answers[index] = nums[target_index]

        # After processing all queries, return the answers list.
        return answers

```

Java Solution

```java
class Solution {

    /**
     * Finds elements in the array after performing certain query operations.
     *
     * @param nums    The input array
     * @param queries Instructions for each query, containing two values [t, i]
     * @return Array of elements according to each query
     */
    public int[] elementInNums(int[] nums, int[][] queries) {
        int numLength = nums.length; // the length of the input array
        int queryCount = queries.length; // the number of queries
        int[] result = new int[queryCount]; // array to store the result elements

        // Loop over each query
        for (int j = 0; j < queryCount; ++j) {
            result[j] = -1; // initialize the result as -1 (not found)
            int period = queries[j][0], index = queries[j][1]; // extract period (t) and index (i) from the query
            period %= (2 * numLength); // reduce the period within the range of 0 to 2*n

            // Check if the required element falls within the unshifted part of the array
            if (period < numLength && index < numLength - period) {
                result[j] = nums[index + period]; // set the element after 'period' shifts
            }
            // Check if the required element falls within the shifted part of the array
            else if (period > numLength && index < period - numLength) {
                result[j] = nums[index]; // set the element without shift as it has cycled back
            }
        }

        return result; // return the result array with elements according to queries
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Function to get the elements in nums after applying rotations from queries
    std::vector<int> elementInNums(std::vector<int>& nums, std::vector<std::vector<int>>& queries) {

        // n is the size of the nums vector; m is the size of the queries vector
        int n = nums.size(), m = queries.size();

        // Initialize an answer vector with size m and fill it with -1
        std::vector<int> ans(m, -1);

        // Iterate over each query
        for (int j = 0; j < m; ++j) {

            // t is the number of rotations and i is the index for the current query
            int t = queries[j][0], i = queries[j][1];

            // Since the array rotates back to the original position after n*2 rotations,
            // we use modulo to simplify the number of rotations
            t %= (n * 2);

            // If the number of rotations is less than the size of nums
            // and the index is within the bounds after the rotation, get the element
            if (t < n && i < n - t) {
                ans[j] = nums[i + t];
            }
            // If the number of rotations is more than size but the index is within bounds,
            // that means the array has been rotated back to its original position
            // and possibly further, we get the element from the original position
            else if (t >= n && i < t - n) {
                ans[j] = nums[i];
            }
            // If none of the above conditions meet, the answer remains -1
            // which indicates that the element is not accessible after the rotation
        }

        // Return the answer vector
        return ans;
    }
};

```

Typescript Solution

```typescript
// Importing array type from Typescript for type annotations
import { Array } from "typescript";

// Function to get the elements in nums after applying rotations from queries
function elementInNums(nums: Array<number>, queries: Array<Array<number>>): Array<number> {

    // n is the size of the nums array; m is the size of the queries array
    const n: number = nums.length;
    const m: number = queries.length;

    // Initialize an answer array with size m and fill it with -1
    let ans: Array<number> = new Array(m).fill(-1);

    // Iterate over each query
    for (let j = 0; j < m; ++j) {

        // t is the number of rotations and i is the index for the current query
        let t: number = queries[j][0];
        const i: number = queries[j][1];

        // Since the array rotates back to the original position after n * 2 rotations,
        // we use modulo to simplify the number of rotations
        t %= (n * 2);

        // If the number of rotations is less than the size of nums
        // and the index is within the bounds after the rotation, get the element
        if (t < n && i < n - t) {
            ans[j] = nums[i + t];
        }
        // If the number of rotations is more than the size but the index is within bounds,
        // that means the array has been rotated back to its original position
        // and possibly further, we get the element from the original position
        else if (t >= n && i < t - n) {
            ans[j] = nums[i];
        }
        // If none of the above conditions meet, the answer remains -1
        // which indicates that the element is not accessible after the rotation
    }

    // Return the answer array
    return ans;
}

```

Time and Space Complexity

The code provided involves iterating over the list of queries and computing the answer for each query based on given conditions. We'll analyze both the time complexity and space complexity.

Time Complexity

The time complexity of the algorithm depends on the number of queries we need to process, as the iterations inside the loop do not depend on the size of the `nums` list. Since we are iterating over every single query once, the time complexity is `O(m)`, where `m` is the number of queries.

We do not have any nested loops that iterate over the length of the `nums` list, and the modulo and if-else operations inside the for loop are all constant time operations.

Hence, the overall time complexity is `O(m)`.

Space Complexity

The space complexity is determined by the additional space required by the program which is not part of the input. Here, the main additional space used is the `ans` list, which contains one element for each query.

Since we create an `ans` list of size `m`, the space complexity is `O(m)`, where `m` is the number of queries.

We do not use any additional data structures that grow with the size of the input, so there is no further space complexity to consider.

Therefore, the overall space complexity is also `O(m)`.