985. Sum of Even Numbers After Queries

Medium Array Simulation

Leetcode Link

Problem Description

In this problem, you are provided with two pieces of information: 1) an integer array `nums`, which contains some integers, and 2) an array of arrays `queries`, where each subarray contains exactly two integers representing a value (`val_i`) and an index (`index_i`). The goal is to process each array in `queries` sequentially. The processing steps for each query are as follows: a) Add `val_i` to `nums[index_i]` (modify the element in `nums` at the specified index by adding the value). b) Calculate the sum of all even numbers currently in the `nums` array.

You need to perform this two-step process for each query and record the result of the second step. The final output should be an array where each element corresponds to the sum of even numbers after the corresponding query has been processed.

Intuition

The naive approach to solving this problem would be to compute the sum of all even numbers in the `nums` array after applying each query. However, this would be inefficient because it would require iterating through the entire `nums` array to recalculate the sum after each query, resulting in a time complexity that would be burdensome for large arrays.

The solution instead maintains a running total sum `s` of all even numbers in the `nums` array. This sum is updated incrementally. Here's the thought process for this approach:

1. Calculate and store the sum of all even numbers in the `nums` array before processing any queries.
2. For each query `[val_i, index_i]`:
   - If the number at `nums[index_i]` is even before the query, subtract it from `s` to temporarily remove it from the even sum. This is done because the addition might make it odd.
   - Apply the update `nums[index_i] = nums[index_i] + val_i`.
   - After the update, check if `nums[index_i]` is now even. If it is, add it back to `s`, as the change either kept it even or turned it from odd to even.
3. Record the current value of `s` after each query to build the answer array, which represents the sum of even numbers after each modification.

This way, we avoid recalculating the sum of even numbers from scratch after each query, effectively reducing complexity and speeding up the algorithm.

Solution Approach

The implementation of the solution uses an approach that efficiently updates and calculates the sum of even numbers after each query applied to the `nums` array. The initial setup and steps followed in the code are as follows:

1. **Initialize the Even Sum**: Before processing any queries, the solution calculates the initial sum `s` of all the even numbers in `nums`. This is done using a list comprehension and the built-in `sum` function:

   ```
   1 s = sum(x for x in nums if x % 2 == 0)
   ```

2. **Process Queries**: The implementation then iterates over each query in `queries`. For each query `[v, i]`, where `v` is the value to be added and `i` is the index, the solution:

   - Checks if `nums[i]` is even before the update. If it is, the value of `nums[i]` is subtracted from the running even sum `s`, as the update might turn it into an odd number:

     ```
     1 if nums[i] % 2 == 0:
     2     s -= nums[i]
     ```

   - Applies the update to `nums[i]` by adding the value `v` to it:

     ```
     1 nums[i] += v
     ```

   - Checks if after the update, `nums[i]` is even. If it has become or remained even, it is added to the running even sum `s`:

     ```
     1 if nums[i] % 2 == 0:
     2     s += nums[i]
     ```

   - Append the current running sum `s` (which now reflects the sum of even values after the update) to an answer list `ans`:

     ```
     1 ans.append(s)
     ```

3. **Return Results**: After processing all the queries, the list `ans`, which has recorded the sum of even numbers after each query, is returned as the final output.

By maintaining and updating the running even sum `s` with each query, the algorithm achieves an efficient and dynamic way to keep track of changes without recomputing the entire sum after each update. This greatly improves the time efficiency, especially when there are many queries or the `nums` array is large.

Example Walkthrough

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

Suppose we are given the integer array `nums = [1, 2, 3, 4]` and queries `[[1, 0], [-3, 2], [5, 3], [4, 1]]`. We need to perform the queries on `nums` and track the sum of even numbers in `nums` after each query.

1. **Initialize the Even Sum**:

   Before any queries, we find the sum of all even numbers in `nums`:

   ```
   1 nums = [1, 2, 3, 4]
   2 s = sum(x for x in nums if x % 2 == 0) # s = 2 + 4 = 6
   ```

   The initial sum `s` of even numbers in `nums` is 6.

2. **Process Queries**:

   First query `[1, 0]`: We add 1 to `nums[0]`, which results in `nums` becoming `[2, 2, 3, 4]`.

   Since `nums[0]` was not even before the query (`1` is odd), we don't subtract anything from `s`. After the update, `nums[0]` becomes even (`2`), so we add it to `s`.

   ```
   1 s += nums[0] # s = 6 + 2 = 8
   2 ans.append(s) # ans = [8]
   ```

   Second query `[-3, 2]`: We subtract 3 from `nums[2]` (`nums[2] - 3`), and `nums` becomes `[2, 2, 0, 4]`.

   Before the query, `nums[2]` was odd, so the sum `s` remains unchanged. After the update, `nums[2]` is even (`0`), and it's added to the sum `s`.

   ```
   1 s += nums[2] # s = 8 + 0 = 8 (unchanged)
   2 ans.append(s) # ans = [8, 8]
   ```

   Third query `[5, 3]`: We now add 5 to `nums[3]`, resulting in `nums` becoming `[2, 2, 0, 9]`.

   `nums[3]` was even before (`4`), so we subtract it from the sum `s` first. After the update (`4 + 5 = 9`), `nums[3]` is odd, so we don't add it back to `s`.

   ```
   1 s -= nums[3] # Subtracting the old value of `nums[3]`, s = 8 - 4 = 4
   2 nums[3] += 5 # nums[3] becomes odd after the addition
   3 ans.append(s) # ans = [8, 8, 4]
   ```

   Fourth query `[4, 1]`: The last query adds 4 to `nums[1]`, changing `nums` to `[2, 6, 0, 9]`.

   Since `nums[1]` was even before (`2`), we remove it from the sum `s`. After the update (`2 + 4 = 6`), `nums[1]` is still even, so we add the new value back to `s`.

   ```
   1 s -= nums[1] # Subtracting the initial value of `nums[1]`, s = 4 - 2 = 2
   2 nums[1] += 4 # Updating `nums[1]`
   3 s += nums[1] # Adding the updated value to `s`, s = 2 + 6 = 8
   4 ans.append(s) # ans = [8, 8, 4, 8]
   ```

3. **Return Results**:

   After processing all the queries, the `ans` list, which contains the sum of even numbers after each query, is `[8, 8, 4, 8]`. This list is the final output.

Thus, by following this approach, we efficiently kept track of the sum of even numbers in `nums` after each query without recalculating the entire sum each time.

Python Solution

```python
from typing import List

class Solution:
    def sumEvenAfterQueries(self, nums: List[int], queries: List[List[int]]) -> List[int]:
        # Calculate the initial sum of even numbers in the array
        sum_even = sum(value for value in nums if value % 2 == 0)
        result = []  # Initialize the result list

        # Iterate over each query in the queries list
        for value_to_add, index in queries:
            # Check if the number at the index is even before the operation
            if nums[index] % 2 == 0:
                # If it's even, subtract it from sum_even because it might change
                sum_even -= nums[index]

            # Add the value to the number at the given index
            nums[index] += value_to_add

            # Check if the number at the index is even after the operation
            if nums[index] % 2 == 0:
                # If it's even now, add it to sum_even
                sum_even += nums[index]

            # Add the current sum of even numbers to the result list
            result.append(sum_even)

        # Return the result list which contains the sum of even numbers after each query
        return result
```

Java Solution

```java
class Solution {
    // Function to calculate the sums of even valued numbers after each query
    public int[] sumEvenAfterQueries(int[] nums, int[][] queries) {
        int evenSum = 0; // Variable to keep track of the sum of even numbers

        // Initial pass to calculate sum of even numbers in the original array
        for (int num : nums) {
            if (num % 2 == 0) {
                evenSum += num;
            }
        }

        // The number of queries
        int numQueries = queries.length;
        // Array to hold results after each query
        int[] ans = new int[numQueries];
        // Index for placing results in the `ans` array
        int resultIndex = 0;

        // Iterate over each query
        for (int[] query : queries) {
            int value = query[0]; // The value to be added
            int index = query[1]; // The index of the nums array to which the value is to be added

            // If the current number at index is even, subtract it from the evenSum
            if (nums[index] % 2 == 0) {
                evenSum -= nums[index];
            }

            // Update the number by adding the value from the query
            nums[index] += value;

            // If the updated number is even, add it to the evenSum
            if (nums[index] % 2 == 0) {
                evenSum += nums[index];
            }

            // Store the current sum of even numbers in the result array
            ans[resultIndex++] = evenSum;
        }

        // Return the array with results after each query
        return ans;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<int> sumEvenAfterQueries(vector<int>& nums, vector<vector<int>>& queries) {
        int sumEven = 0; // Variable to store the sum of even numbers

        // Calculate the initial sum of even elements in nums
        for (int num : nums) {
            if (num % 2 == 0) {
                sumEven += num;
            }
        }

        vector<int> result; // Vector to store the results after each query
        result.reserve(queries.size()); // Reserve space to avoid reallocations

        // Process each query
        for (auto& query : queries) {
            int val = query[0]; // Value to add
            int index = query[1]; // Index at which the value is to be added

            // If the original number at index is even, subtract it from sumEven
            if (nums[index] % 2 == 0) {
                sumEven -= nums[index];
            }

            // Add the value to the number at index
            nums[index] += val;

            // If the new number at index is even, add it to sumEven
            if (nums[index] % 2 == 0) {
                sumEven += nums[index];
            }

            // Append the current sumEven to the result
            result.push_back(sumEven);
        }

        return result; // Return the final result vector
    }
};
```

Typescript Solution

```typescript
function sumEvenAfterQueries(nums: number[], queries: number[][]): number[] {
    // Calculate the sum of even numbers in the array.
    let sumEven = 0;
    // Calculate the initial sum of all even numbers in the 'nums' array.
    for (const num of nums) {
        if (num % 2 === 0) {
            sumEven += num;
        }
    }

    // Prepare the array to store result after each query.
    const result: number[] = [];

    // Iterate over each query in 'queries' array.
    for (const [valueToAdd, index] of queries) {
        // If the element at the current index is even, subtract it from 'sumEven'.
        if (nums[index] % 2 === 0) {
            sumEven -= nums[index];
        }

        // Add the value from the query to the element at the current index.
        nums[index] += valueToAdd;

        // If the updated element at the current index is even, add it to 'sumEven'.
        if (nums[index] % 2 === 0) {
            sumEven += nums[index];
        }

        // Append the current sum of even numbers to the result array.
        result.push(sumEven);
    }

    // Return the final result array.
    return result;
}

```

Time and Space Complexity

The provided code snippet consists of an initial computation that sums the even values of the input `nums` list, followed by an iteration through each query in `queries`. Each query modifies a single element of `nums` and conditionally updates the sum of even numbers.

Time Complexity:

The initial sum computation has a time complexity of $O(N)$, where N is the number of elements in `nums`.

The for loop runs for every query in `queries`, which we can denote as Q, where Q is the number of queries. Inside the loop, we have constant-time operations. Therefore, the loop runs in $O(Q)$ time.

The total time complexity of the function is a result of the initial sum computation and the queries loop, which we can express as $O(N) + O(Q)$. Therefore, the overall time complexity is $O(N + Q)$.

Space Complexity:

The space complexity is determined by the additional space used by the algorithm aside from the input itself. The space used includes the sum `s` and the list `ans` that accumulates the resulting sums after each query.

- `s` is a single integer, so it requires $O(1)$ space.
- `ans` will hold the result after every query, which means it will grow to the size of Q. Therefore, it requires $O(Q)$ space.

Considering the additional space used by the function, the total space complexity is $O(Q)$ since Q may vary independently of N and the size of `ans` directly depends on the number of queries.