2778. Sum of Squares of Special Elements

Easy Array Simulation

Leetcode Link

Problem Description

In this problem, we are presented with an array of integers `nums` which uses 1-indexing, meaning the first element is considered at position 1, the second at position 2, and so on. The array's length is denoted as `n`. A special element within this array is defined as an element `nums[i]` where the index `i` is a divisor of `n` (the array's length), i.e., `n % i == 0`.

Our task is to find the sum of the squares of all the special elements in the `nums` array. This sum is the outcome we are required to return.

Intuition

The solution hinges on the simple observation that we only need to consider elements at indices which are divisors of the array length `n`. To find these indices, we iterate over the array, starting at the 1st element (index 1), and continue to the last element (index `n`). For each element `nums[i]`, we check if `i` is a divisor of `n` by using the modulus operation `n % i`. If the result of this operation is 0, it means that `i` is a divisor of `n`, and we consider `nums[i]` a special element.

Once we identify a special element, we take its square and add it to an accumulator that is tracking the sum of squares. The concise Python code uses list comprehension, which is a compact way to process lists and apply operations to each element. By iterating over `nums` with `enumerate`, we get both the element and its 1-based index in each iteration. We filter for the indices that are divisors of `n` and calculate the square of their corresponding values, summing them all up in one line.

The elegance of the solution comes from its efficiency—there is no need to check every index against every other number to find divisors and only a single pass through the array is required. This makes the solution very efficient in terms of time complexity, having an O(n) runtime where n is the number of elements in the `nums` array.

Solution Approach

The implementation of the given solution approach uses a few key Python concepts: list comprehension, the `enumerate` function, and the modulus operator. Since no specific algorithms or complex data structures are involved in this solution, it is straightforward and elegant due to its direct logic.

A breakdown of the steps in the solution implemented in the Python code is as follows:

1. **List Comprehension**: A powerful feature in Python that allows us to create a list based on an existing list. It is often used to apply an operation to each element in the list.
2. **enumerate Function**: This is a built-in Python function that adds a counter to an iterable. In this case, it is used to get both the element and its index from the `nums` array. The `enumerate(nums, 1)` call starts the counting with 1, which aligns with the 1-indexed array described in the problem.
3. **Modulus Operator (`%`)**: This operator is used to find the remainder of the division of two numbers. In our case, it is used to check if the index `i` is a divisor of `n`.
4. **Conditional Filter**: The `if n % i == 0` part right after the `for` loop in the list comprehension acts as a filter. It includes only those elements in the final list whose indices are divisors of `n`.
5. **Square of Elements**: For each element in the list that passed the filter condition, the square is calculated using `x * x`, where `x` is the value of the element in `nums`.
6. **Sum Function**: To combine all the squared values into a single sum, the built-in `sum()` function is used.

The combination of these elements results in the following single line of Python code, which constitutes the core of the solution:

```
return sum(x * x for i, x in enumerate(nums, 1) if n % i == 0)
```

Here's the explanation of how the code executes:

- `for i, x in enumerate(nums, 1)` iterates through the `nums` array, with `i` capturing the index (starting at 1) and `x` capturing the value at each index.
- `if n % i == 0` is the condition that checks if the index `i` is a divisor of `n`.
- `x * x` computes the square of the element if the condition is true.
- `sum()` adds up all the squared values that meet the condition, resulting in the sum of the squares of all special elements.

In essence, the solution loops through the array once, checking the divisibility of each index and squaring and summing the values in one seamless operation, which makes it both efficient and clean.

Example Walkthrough

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

Assume we are given the following array `nums` with 1-indexing where the array length `n` is 6:

```
nums = [3, 1, 4, 1, 5, 9]
```

Remember that a "special element" in this context means `nums[i]` where `i` is a divisor of `n` (which is 6 in this case).

We want to find the sum of the squares of all such special elements.

1. We first determine the divisors of `n`. The divisors of 6 are 1, 2, 3, and 6.
2. We then find the elements at these indices in the `nums` array:
   - `nums[1]` is 3 (since nums is 1-indexed)
   - `nums[2]` is 1
   - `nums[3]` is 4
   - `nums[6]` is 9
3. Now, we calculate the square of each of these special elements:
   - The square of `nums[1]` (which is 3) is (3^2 = 9)
   - The square of `nums[2]` (which is 1) is (1^2 = 1)
   - The square of `nums[3]` (which is 4) is (4^2 = 16)
   - The square of `nums[6]` (which is 9) is (9^2 = 81)
4. We add up these squared values to get the sum:

   (9 (from\ nums[1]) + 1 (from\ nums[2]) + 16 (from\ nums[3]) + 81 (from\ nums[6]) = 107)

Therefore, the result for the example problem, which is the sum of the squares of the special elements, is 107.

The Python code implementing this using the given solution approach would look like this:

```
nums = [3, 1, 4, 1, 5, 9]
n = len(nums)
result = sum(x * x for i, x in enumerate(nums, 1) if n % i == 0)
```

After executing this code, the variable `result` would hold the value 107, which is the correct answer to our example problem.

Python Solution

```
class Solution:
    def sumOfSquares(self, nums: List[int]) -> int:
        # Calculate the length of the input list
        length = len(nums)

        # Use a list comprehension to find the sum of squares of the elements
        # that correspond to indices which are divisors of the list's length
        sum_of_squares = sum(x * x for index, x in enumerate(nums, start=1) if length % index == 0)

        # Return the calculated sum of squares
        return sum_of_squares

# The enumerate function in the list comprehension is used to iterate over
# the nums list along with the indices, starting from 1. The condition inside
# the list comprehension checks if the current index is a divisor of the
# length of the list by checking if the remainder of the division is 0.
# If the condition is True, the square of the number at the current index (x * x)
# is included in the sum_of_squares. The sum function then adds up all
# the squared values to give the final result.
```

Java Solution

```
class Solution {
    // This method calculates the sum of squares of specific elements in the array.
    // It adds the square of the number at the index that is a divisor of the length of the array.
    public int sumOfSquares(int[] nums) {
        // 'n' represents the length of the input array 'nums'.
        int n = nums.length;
        // 'sum' will hold the cumulative sum of squares of selected elements.
        int sum = 0;

        // We iterate over all possible divisors of 'n', starting from 1 to 'n' inclusive.
        for (int i = 1; i <= n; ++i) {
            // We check if 'i' is a divisor of 'n'.
            if (n % i == 0) {
                // If 'i' is a divisor, add the square of the element at the '(i - 1)th index to 'sum'.
                // Since array indices in Java are 0-based, we access the element using 'i - 1'.
                sum += nums[i - 1] * nums[i - 1];
            }
        }
        // The method returns the calculated sum.
        return sum;
    }
}
```

C++ Solution

```
#include <vector> // Include the header for std::vector

class Solution {
public:
    // Function to calculate the sum of squares of elements at indices which are divisors of the vector's size.
    int sumOfSquares(std::vector<int>& nums) {
        int size = nums.size(); // Store the size of the vector
        int totalSum = 0; // Initialize the sum accumulator

        // Loop over the elements of the vector starting at index 1 (not 0)
        for (int index = 1; index <= size; ++index) {
            // If the current index is a divisor of the vector's size
            if (size % index == 0) {
                // Add the square of the corresponding element to the total sum.
                // Note that we subtract 1 from the index since C++ arrays are 0-based.
                totalSum += nums[index - 1] * nums[index - 1];
            }
        }

        // Return the total sum of the squares.
        return totalSum;
    }
};
```

Typescript Solution

```
/**
 * Calculates the sum of the squares of elements in the given array
 * `nums` where the element's index plus one is a divisor of the array's length.
 * @param nums An array of numbers.
 * @return The sum of the squares of selected elements.
 */
function sumOfSquares(nums: number[]): number {
  // Get the number of elements in the array `nums`.
  const arrayLength = nums.length;
  // Initialize the variable to hold the sum of the squares.
  let sum = 0;
  // Iterate over the array `nums`.
  for (let index = 0; index < arrayLength; ++index) {
    // Check if index plus one is a divisor of the array's length.
    if (arrayLength % (index + 1) === 0) {
      // If so, add the square of the current element to the sum.
      sum += nums[index] * nums[index];
    }
  }
  // Return the computed sum of squares.
  return sum;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` list. This is because the function iterates over all elements in the list exactly once. The condition `n % i == 0` can be checked in constant time for each iteration, so it does not add to the complexity order.

Space Complexity

The space complexity of the code is `O(1)`. No additional space is required that is dependent on the input list size, as the summation is done on-the-fly and only the sum variable (`x * x for i, x in enumerate(nums, 1) if n % i == 0`) is being maintained throughout the loop.