2862. Maximum Element-Sum of a Complete Subset of Indices

Hard Array Math Number Theory Leetcode Link

Problem Description

You are provided with an array `nums` that is 1-indexed and contains `n` integers. A complete set of numbers is defined as a set where the product of every two elements results in a perfect square. A subset of indices from `{1, 2, ..., n}` is represented by `{i1, i2, ..., ik}`, and the element-sum of this subset is the sum of the elements at these indices (`nums[i1] + nums[i2] + ... + nums[ik]`). The objective is to determine the maximum element-sum for a complete subset of the indices set `{1, 2, ..., n}`. It's important to note that a perfect square is an integer that is the square of another integer. In other words, it has an integer square root.

Intuition

The intuition behind the solution comes from understanding what makes a set of numbers complete with respect to being a perfect square. Each number in a complete set must be paired with another number from the set such that their product is a perfect square. This suggests a relationship between the indices of the given `nums` array.

Specifically, if we look at the indices of the array that are perfect squares themselves (e.g., 1^2, 2^2, 3^2, etc.), these indices have a special property — they will always produce perfect squares when multiplied by any other index that is a perfect square. Leveraging this property, we can iterate through the array and sum up elements at indices that are perfect squares.

The algorithm makes use of two loops. The outer loop iterates over all possible starting indices `k` (from 1 to `n`). For each `k`, the inner loop checks for indices that are multiples of the square of `j` (`k * j * j`), which would be perfect squares if `k` is a perfect square. The inner loop sums up `nums[k * j * j - 1]` for all `j` producing indices within the bounds of the array; this sum is a candidate for a complete subset with maximum element-sum. The algorithm maintains the maximum such sum and returns it at the end.

Solution Approach

The solution's implementation focuses on iterating through the `nums` array while considering each element's index as a potential starting point for the set. For each starting point, we attempt to create a complete subset by including elements that correspond to indices that are products of the current index and perfect squares (`j * j`).

Here's a step-by-step walkthrough of the solution implementation:

- Initialize `n` to be the length of `nums`. This will be used to determine the bounds of our search for complete subsets.
- Initialize `ans` to 0. This variable will keep track of the maximum element-sum of any complete subset found so far.
- Iterate `k` from 1 to `n`. `k` represents the starting index of our set (1-indexed, as the problem states).
  - Within this loop, initialize `t` to 0. `t` will accumulate the sum of elements of a potential complete subset starting at index `k`.
  - Initialize `j` to 1. `j` will be used to generate perfect square multipliers as we look for other indices to include in the subset.
  - While the product of `k` and the square of `j` (`k * j * j`) is less than or equal to `n` (to stay within the bounds of the array):
    - Add `nums[k * j * j - 1]` to `t`. Since the array is 1-indexed, we subtract 1 to get the 0-indexed position used in most programming languages, including Python.
    - Increment `j` by 1 to check the next possible perfect square.
  - After considering all j's that fall within the bounds of the array, compare the sum `t` with the current maximum `ans` to see if we found a new maximum element-sum for a complete subset.
- Return `ans`, which now holds the maximum element-sum of a complete subset of the indices set `{1, 2, ..., n}`.

The choice of data structures is minimal; we only use simple variables for tracking purposes. The key pattern used is a nested loop structure where the outer loop establishes a starting point and the inner loop checks for eligible indices based on the perfect square condition.

One can note that the algorithm is brute-force in nature and may not be the most efficient for larger input sizes. However, for the problem's constraints, it is sufficient to derive the correct answer.

Example Walkthrough

Let's apply the solution approach to a small example to illustrate how it works.

Suppose we have the following array `nums` which is 1-indexed:

```
nums = [1, 2, 3, 4, 5, 7, 8]
```

In this example, `n` = 7 (the length of the array).

Now, let's walk through the algorithm:

1. We initialize `ans` to 0.
2. We start iterating `k` from 1 to `n`. In each iteration, `k` represents the starting index of a potential complete subset.

For `k = 1`:

- We initialize `t` to 0 and `j` to 1.
- When `j = 1`, `k * j * j` = `1 * 1 * 1` = 1, which is less than or equal to `n`. So we add `nums[1 - 1]` to `t`. `t` becomes `t + 1` = 1.
- Increment `j` to 2. Now, `k * j * j` = `1 * 2 * 2` = 4, which is also within the bounds. Add `nums[4 - 1]` to `t`. Now `t` becomes `1 + 4` = 5.
- Increment `j` to 3. Now, `k * j * j` = `1 * 3 * 3` = 9, which is greater than `n`. We do not add anything to `t` and break the loop.
- Compare `t` and `ans`. Here, `t` = 5, which is greater than `ans` = 0, so update `ans` to 5.

For `k = 2`:

- Again, initialize `t` to 0 and `j` to 1.
- When `j = 1`, `k * j * j` = `2 * 1 * 1` = 2, within bounds. Add `nums[2 - 1]` = 2 to `t`, so `t` is now 2.
- Increment `j` to 2. Now, `k * j * j` = `2 * 2 * 2` = 8, also within bounds. Add `nums[8 - 1]` = 8 to `t`, so `t` becomes `2 + 8` = 10.
- Incrementing `j` to 3 gives `k * j * j` = `2 * 3 * 3` = 18, outside the bounds. Break the loop.
- `t` is now 10, which is greater than `ans` = 5, so we update `ans` to 10.

This process continues for `k = 3` to `k = 7`.

Ultimately, after considering all k's (from 1 to 7), we end up with the maximum `ans` that represents the maximum element-sum for a complete subset. Suppose the final `ans` after considering all possible `k` values was 10, then 10 would be the maximum element-sum of a complete subset of indices `{1, 2, ..., n}` for the provided example array `nums`.

The example shows that by iterating over each index and calculating the sum of elements corresponding to the indices that are products of the current index and a perfect square (`j * j`), we can find the subset of indices that yields the highest sum under the given constraints. This sum is the answer to the problem.

Python Solution

```python
from typing import List

class Solution:
    def maximum_sum(self, nums: List[int]) -> int:
        # Get the total number of elements in the list
        num_elements = len(nums)
        # Initialize the maximum sum as zero
        max_sum = 0

        # Iterate through each element in the list
        for i in range(1, num_elements + 1):
            # Initialize the temporary sum for each 'i' as zero
            temp_sum = 0
            # Initialize the multiplier as one
            multiplier = 1

            # While the square of the multiplier times 'i' is within the range of the list indices
            while i * multiplier * multiplier <= num_elements:
                # Add the corresponding element to the temporary sum
                # The '-1' accounts for zero-based indexing in Python lists
                temp_sum += nums[i * multiplier * multiplier - 1]
                # Increment the multiplier
                multiplier += 1

            # Update the maximum sum if the current temporary sum is greater than the current maximum
            max_sum = max(max_sum, temp_sum)

        # Return the maximum sum after considering all elements
        return max_sum

# Example of using this solution class to find maximum sum.
# sol = Solution()
# result = sol.maximum_sum([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
# print(result) # This will print the result of calling maximum_sum with the provided list.

```

Java Solution

```java
class Solution {
    public long maximumSum(List<Integer> nums) {
        // Initialize the variable to store the maximum sum found so far.
        long maxSum = 0;
        // Store the total number of elements in the nums List.
        int listSize = nums.size();
        // Iterate over all possible values of k, up to the size of the list.
        for (int k = 1; k <= listSize; ++k) {
            // Temporary variable to store the sum for the current value of k.
            long currentSum = 0;
            // Iterate to sum the elements at index k * j^2 - 1, if within the bounds of the list.
            for (int j = 1; k * j * j <= listSize; ++j) {
                // Accumulate the sum for indices that are k times the square of j (adjusted for zero-based index).
                currentSum += nums.get(k * j * j - 1);
            }
            // Update the maxSum if the current sum is greater than the previously recorded maximum.
            maxSum = Math.max(maxSum, currentSum);
        }
        // Return the maximum sum found.
        return maxSum;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm to use the max() function

class Solution {
public:
    // This function calculates the maximum sum of a subsequence
    // from the given vector of integers, where the indices of the
    // subsequence elements are determined by the formula k * j * j.
    long long maximumSum(vector<int>& nums) {
        long long max_sum = 0; // Initialize the answer with 0
        int nums_size = nums.size(); // Get the number of elements in the vector

        // Iterate through the vector, considering each element as a starting point
        for (int k = 1; k <= nums_size; ++k) {
            long long current_sum = 0; // Initialize the sum of the current subsequence as 0

            // Iterate through multiples of k to find elements at indices k * j * j
            for (int j = 1; k * j * j <= nums_size; ++j) {
                // Add the element at index k * j * j - 1 to current_sum
                // (subtract 1 because vector indices are 0-based)
                current_sum += nums[k * j * j - 1];
            }

            // Update max_sum if the sum of the current subsequence is larger
            max_sum = std::max(max_sum, current_sum);
        }
        // Return the maximum sum found
        return max_sum;
    }
};

```

Typescript Solution

```typescript
function maximumSum(nums: number[]): number {
    // Initialize maximum sum.
    let maxSum = 0;
    // Get the number of elements in the array.
    const numElements = nums.length;

    // Iterate over each number from 1 through the number of elements.
    for (let i = 1; i <= numElements; ++i) {
        // Temporary sum for the current iteration.
        let tempSum = 0;

        // Iterate over each multiplier to find indices of the form (i * j * j).
        for (let j = 1; i * j * j <= numElements; ++j) {
            // Accumulate elements in temporary sum where the index meets the criteria.
            tempSum += nums[i * j * j - 1];
        }

        // Update maxSum with the maximum of itself and the tempSum.
        maxSum = Math.max(maxSum, tempSum);
    }

    // Return the maximum sum found.
    return maxSum;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be observed based on the two nested loops. The outer loop runs `k` from 1 to `n` (inclusive), which gives us `O(n)` for the outer loop. The inner while loop runs as long as `k * j * j <= n`. For each fixed `k`, the maximum `j` will be roughly the square root of `n/k`. Thus, the time spent on the inner loop is `O(sqrt(n/k))` for a fixed `k`. To find the total complexity, we need to sum this over all `k` from 1 to `n`. This results in `O(sum_{k=1}^{n}{sqrt(n/k)})`.

The sum can be approximated using integral bounds, which gives us `O(integral from 1 to n of sqrt(n/x) dx)`, and the integral of `sqrt(n/x)` is `2*sqrt(n*x)`, evaluating this from 1 to `n` gives us roughly `O(2*n)`, as the lower bound contributes negligibly. Hence, the overall time complexity simplifies to `O(n)`.

Space Complexity

The space complexity of the code is `O(1)`, since aside from the input list, only a constant amount of extra space is used for variables like `n`, `ans`, `t`, and `j`.