2681. Power of Heroes

Hard Array Math Prefix Sum Sorting Leetcode Link

Problem Description

The given problem involves calculating and summing the power of all non-empty groups of heroes based on their strength values provided in an integer array `nums`. The power of a group is determined by the following:

- Identify the heroes in the group by their indices in the array, say $i_0, i_1, ..., i_k$.
- The power of this group is the square of the maximum strength value among these heroes multiplied by the minimum strength value of the same heroes, i.e., $\max(nums[i_0], nums[i_1], ..., nums[i_k])^2 * \min(nums[i_0], nums[i_1], ..., nums[i_k])$.

The goal is to add up the power of all possible non-empty groups of heroes. Since the result could be quite large, it should be returned modulo $10^9 + 7$.

Intuition

To solve the problem, we need to figure out how to efficiently calculate the sum of the powers of all groups. Brute force enumeration of all groups would be too time-consuming due to the potentially large number of hero combinations. Instead, we can analyze the pattern of how the power of groups contributes to the total sum.

The solution revolves around observing that sorting the array `nums` will make it easier to identify the maximums and minimums for different groups. Once sorted, any group with the last element as its maximum will have this last element as the maximum for all subsequent smaller subgroups ending with this element.

- Reverse iterate over the sorted `nums`. We count the contributions to the power from each element when it acts as both a maximum and minimum within different subgroups.
- A running prefix sum `p` is kept. When a new element `x` is considered, it has an additional contribution when acting as the maximum—due to all possible subgroups ending with `x`. Its contribution is `x^3` added to all contributions from previous elements.
- When `x` is a minimum, it adds `x * p` to the result, where `p` includes the contributions from elements to the right of `x` in the sorted array when they act as maximums.
- The prefix sum `p` is updated for the contribution of `x` being the maximum so far. It's the sum of all previous `p` values doubled (since every subgroup comes in two variants—with and without the new element `x`) plus the square of the new element `x` (which is the new maximum for subgroups ending with `x`).

The answer is computed by summing contributions while iterating through the array and using modulo arithmetic to handle large numbers.

Solution Approach

The solution uses a simple array and basic arithmetic operations to calculate the required sum. The steps in the provided solution approach can be broken down as follows:

1. Sort the array – Before iterating over `nums`, it is sorted in ascending order. This sorting enables us to find the maximums and minimums for subgroups based on their position in the sorted array.

2. Initialize variables – The solution sets an initial value for `ans`, which will hold the sum of powers, and `p`, which is our running prefix sum representing the contributions of maximum values so far.

3. Iterate in reverse – The solution then iterates the sorted array in reverse (`nums[::-1]`), considering each number from largest to smallest, which aligns with how we consider maximums for subgroups.

   After entering the loop, it performs the following operations for every element `x`:

   - Add to `ans` the value of `x` cubed (`x * x % mod) * x` modulo `10**9 + 7`. This covers the new subgroups where `x` is the maximum.

   - Add to `ans` the value of `x` multiplied by the current prefix sum `p`, effectively capturing the contribution of `x` when it is the minimum value in a subgroup with any of the previous elements as the maximum.

   - Update the prefix sum `p` by doubling it and adding the square of the current element `x`. Doubling accounts for the fact that each existing subgroup can now be extended by either including or excluding `x`. Adding `x*x` accounts for the new subgroups formed with `x` as the maximum.

4. Modulo operations – To handle potentially large numbers and overflow issues, modulo arithmetic is consistently applied (`% mod`) to ensure that all intermediate values and the resulting sum remain within integer limits.

By considering each element as the potential minimum and maximum of various subgroups, the solution efficiently accumulates the power of all possible combinations. The space complexity is optimal as it only uses a fixed number of variables and the original input array, while the time complexity is driven by the sorting operation and the subsequent linear traversal of the array.

Example Walkthrough

Let's use a small example to illustrate the solution approach with the given problem:

Suppose we have an integer array of hero strengths `nums = [2, 1, 3]`.

1. **Sort the array**: The first step is to sort `nums` to `[1, 2, 3]`.

2. **Initialize variables**: We initialize `ans` to 0, which will store the final answer, and `p` to 0, which is the prefix sum of contributions when numbers act as maximums.

3. **Iterate in reverse**: Now, we iterate from the end of the sorted array.

   At the start, `ans = 0` and `p = 0`. We go through each element `x` in `nums[::-1]` (which gives us `[3, 2, 1]`) and perform the next steps.

   - For `x = 3`:
     - Update `ans` by adding `3^3 % (10^9 + 7) = 27`.
     - `p` gets updated to `2 * p + x^2 = 0 + 3^2 = 9`.
   - For `x = 2`:
     - Update `ans` by adding both `2^3 % (10^9 + 7) = 8` and `2 * p = 2 * 9 = 18`. Now, `ans = ans + 8 + 18 = 27 + 26 = 53`.
     - Update `p` to `2 * p + x^2 = 2 * 9 + 2^2 = 18 + 4 = 22`.
   - For `x = 1`:
     - Update `ans` by adding both `1^3 % (10^9 + 7) = 1` and `1 * p = 1 * 22 = 22`. Now, `ans = 53 + 1 + 22 = 76`.
     - Update `p` to `2 * p + x^2 = 2 * 22 + 1^2 = 44 + 1 = 45`.

4. **Modulo operations**: Always use modulo `10^9 + 7` during the addition to keep the numbers within bounds. However, in our small example, the answers are still small, so we don't need to apply modulo.

After iterating through all elements, we have the final `ans = 76`. This is the sum of the power of all possible non-empty groups of heroes calculated with the given formula, and since the value is already less than `10^9 + 7`, we don't need to apply the modulo operation in this example.

In a larger example, if at any step the value of `ans` or `p` exceeds `10^9 + 7`, we would take `ans % (10^9 + 7)` or `p % (10^9 + 7)` respectively to keep the values within the limit.

Python Solution

```python
from typing import List

class Solution:
    def sumOfPower(self, nums: List[int]) -> int:
        # Define the modulo to perform operations under
        modulo = 10**9 + 7
        # Sort the list of numbers in non-descending order
        nums.sort()
        # Initialize the answer variable to accumulate the sum of powers
        answer = 0
        # Initialize a variable 'power_sum' to store the accumulated powers of elements
        power_sum = 0
        # Iterate over the numbers in reverse (largest to smallest)
        for num in reversed(nums):
            # For each number, add the cubed value (mod modulo) multiplied by the number to the answer
            answer = (answer + (num * num % modulo) * num) % modulo
            # Add the current number times the accumulated power_sum to the answer
            answer = (answer + num * power_sum) % modulo
            # Update the power_sum with the current num squared plus twice the previous power_sum
            power_sum = (power_sum * 2 + num * num) % modulo
        # Return the computed answer
        return answer

```

Java Solution

```java
class Solution {
    public int sumOfPower(int[] nums) {
        // Define the modulus value for large numbers to stay within the integer bounds
        final int MOD = (int) 1e9 + 7;

        // Sort the input array in ascending order
        Arrays.sort(nums);

        // Initialize the answer as 0 and a helper variable p to compute powers
        long answer = 0;
        long powers = 0;

        // Iterate over the array in reverse to compute the sum of powers
        for (int i = nums.length - 1; i >= 0; --i) {
            long currentNum = nums[i];

            // Compute the contribution of the current number raised to the power of 3
            // Make sure to take the modulo to prevent integer overflow
            answer = (answer + (currentNum * currentNum % MOD) * currentNum) % MOD;

            // Add the contribution of the current number times the partial sum of powers (p)
            // Again, use modulo to avoid overflow
            answer = (answer + currentNum * powers % MOD) % MOD;

            // Update p to be 2 times itself plus the current number squared (mod MOD)
            powers = (powers * 2 + currentNum * currentNum % MOD) % MOD;
        }

        // Cast the long result back to int before returning, as per the method signature expectation
        return (int) answer;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int sumOfPower(vector<int>& nums) {
        const int MOD = 1e9 + 7; // define modulo value to prevent integer overflow
        sort(nums.rbegin(), nums.rend()); // sort the numbers in descending order

        long long totalSum = 0; // holds the total sum result
        long long powerSum = 0; // holds the sum of squares, multiplied by 2 each iteration

        // Iterate through all the numbers in the vector
        for (long long num : nums) {
            // Add to totalSum the current number cubed, modulo MOD
            totalSum = (totalSum + (num * num % MOD) * num) % MOD;

            // Add to totalSum the current number multiplied by powerSum, modulo MOD
            totalSum = (totalSum + num * powerSum % MOD) % MOD;

            // Update powerSum by multiplying by 2 and adding the current number squared, modulo MOD
            powerSum = (powerSum * 2 + num * num % MOD) % MOD;
        }

        return static_cast<int>(totalSum); // cast totalSum to int and return the final answer
    }
};

```

Typescript Solution

```typescript
function sumOfCubedPowers(nums: number[]): number {
    // The modulo to use for preventing overflow
    const MODULO = 10 ** 9 + 7;

    // Sort the numbers array in ascending order
    nums.sort((a, b) => a - b);

    // Initialize the answer to 0
    let answer = 0;

    // Initialize the prefix sum, representing sum of powers
    let prefixSum = 0;

    // Iterate over the array in reverse
    for (let i = nums.length - 1; i >= 0; --i) {
        // Get the current number as a BigInt for precision in calculations
        const currentNum = BigInt(nums[i]);

        // Add the cube of the current number modulo MODULO to the answer
        answer = (answer + Number((currentNum * currentNum * currentNum) % BigInt(MODULO))) % MODULO;

        // Add the product of the currentNum and the prefixSum modulo MODULO to the answer
        answer = (answer + Number((currentNum * BigInt(prefixSum)) % BigInt(MODULO))) % MODULO;

        // Update the prefixSum: multiply by 2 and add the square of currentNum
        prefixSum = Number((BigInt(prefixSum) * 2n + currentNum * currentNum) % BigInt(MODULO));
    }

    // Return the final answer
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided function is determined by a few key operations:

1. Sorting the `nums` list: The `sort()` function has a time complexity of `O(n log n)` where `n` is the number of elements in the list.

2. The for-loop that iterates over the sorted list in reverse: The loop runs `n` times, where `n` is the length of `nums`.

   Inside the loop, all operations are constant time (`O(1)`), as they involve basic arithmetic operations and modulo `%` operation, which do not depend on the size of `nums`.

   Combining these constants with the iteration gives us `O(n)` for the loop.

Combining both operations, the total time complexity of the code is `O(n log n)` for sorting the list plus `O(n)` for the loop, which simplifies to `O(n log n)` overall, as `O(n log n)` is the dominating term.

Space Complexity

The space complexity of the provided function is determined by:

1. The additional space used by the sorting algorithm. Most Python implementations use an in-place sorting algorithm, like Timsort, which would make the space complexity of sorting `O(n)` in the worst case. However, the sorted list uses the same space that was allocated for `nums`, so this may not count as additional space.

2. Variables `ans`, `p`, and `mod` occupy `O(1)` space as they store integers that don't depend on the size of the input list.

3. The loop itself does not use any additional space that depends on `n` (no new list or data structure is being created that grows with the input).

Considering the above points, the space complexity of the function is `O(1)` for the variables defined outside the loop since they do not use additional space dependent on the input list size. However, taking into account the potential additional space required for sorting (if interpreted as additional space rather than in-place), it could be argued to be `O(n)` in a pessimistic assessment.

Overall, if the sort is considered in-place, the space complexity is `O(1)`; otherwise, it may be `O(n)` in the worst case.