2708. Maximum Strength of a Group

Medium Greedy Array Backtracking Sorting Leetcode Link

Problem Description

In this problem, we are provided with an array `nums` of integers that represents the scores of students in an exam. The goal is to form a group of students that has the maximum possible strength. A group's strength is the product (multiplication) of the scores of all the students included in the group. The challenge lies in selecting which students should be in the group to maximize this product.

To clarify, the array is 0-indexed, meaning the indexing starts at `0`, and the group must be non-empty, so at least one student must be included. The task is to find the maximum strength that can be achieved and return it.

Intuition

For the solution, we need to find a way to combine the scores in `nums` to get the highest possible product. A key insight here is the following rules about multiplying numbers:

- Multiplying two positive numbers gives a larger number.
- Multiplying two negative numbers gives a positive number.
- Multiplying a positive and a negative number results in a negative number.

Based on these, we can devise the following approach to maximize the product:

1. First, sorting the `nums` array helps arrange the numbers so that we can process them in order.
2. Once sorted, we ignore `0`s unless all the other numbers are zeroes too, because they do not have any impact on the product.
3. For negative numbers, we prefer to multiply them in pairs (as multiplying two negatives gives a positive), which means we look for pairs of negative numbers to maximize the positive outcome.
4. For positive numbers, we simply take them as they are, since multiplying them will always increase the product.

Given these rules, we iterate through the sorted array:

- When we find consecutive negative numbers, we multiply them together and continue.
- If we encounter a single negative number or zero, we skip it if there's no pair (this will be true if there's an odd number of negatives in total).
- For positive numbers, we include each one in the product.
- We need to be cautious of an edge case where there is only one negative number and the rest are zeros. The result in this case should be zero since we cannot form a negative pair and we don't want to include a standalone negative.

Finally, we keep calculating the product as we go through these steps to arrive at the maximum strength.

Solution Approach

The implementation makes use of a few key algorithmic concepts, namely sorting and a single pass approach to maximize the product of a subset of the array. Let's examine the process step-by-step with regards to the Python solution code provided:

1. **Sorting**: We start off by sorting the array `nums` in ascending order. This action arranges all the negative numbers (if any) at the beginning, followed by zeros and positive numbers. Sorting is crucial because it lets us efficiently pair negative numbers, and access positive numbers in the correct sequence.

   ```
   nums.sort()
   ```

2. **Handling Special Cases**: Before the main logic, we have a few checks for edge cases:

   - If the array contains only one number, that number is the maximal strength by default.
   - If after the first positive number all other numbers are zeros, the maximum strength is zero.

   ```
   n = len(nums)
   if n == 1:
       return nums[0]
   if nums[1] == nums[-1] == 0:
       return 0
   ```

3. **Iterating and Multiplying**: We then enter a loop where we iterate over the sorted numbers to calculate the product. We initialize a variable `ans` to 1 (the identity element for multiplication) to keep track of the ongoing product.

4. **Utilizing Negative Numbers in Pairs**: As we loop through the array, we check if there are at least two consecutive negative numbers. If so, their product is positive and will contribute to maximizing the strength, so we multiply them together and add to our total product `ans`. We increment the loop counter by 2 since we've processed two elements.

   ```
   if nums[i] < 0 and i + 1 < n and nums[i + 1] < 0:
       ans *= nums[i] * nums[i + 1]
       i += 2
   ```

5. **Skipping Non-contributing Numbers**: If a number is not followed by another negative number (it's a standalone negative or zero), it will not contribute to maximizing the product. We skip it and move on.

   ```
   elif nums[i] <= 0:
       i += 1
   ```

6. **Including Positive Numbers in the Product**: Finally, when we encounter positive numbers, we multiply them with our product `ans`. We know they will always contribute to a maximal product as any positive number times another positive number will increase the product.

   ```
   else:
       ans *= nums[i]
       i += 1
   ```

7. **Return Result**: After looping through all elements of the sorted array, the variable `ans` now holds the maximum strength, and we return this value.

   ```
   return ans
   ```

The solution employs a single pass of the sorted array, multiplying pairs of negative numbers and any positive numbers while skipping any non-contributing elements, to efficiently calculate the maximum strength of a possible group.

This approach ensures that we are using each negative number (if any) in the most optimized way by pairing them and that every positive number, which always contributes positively to the product, is included in the final calculation.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the array:

```
[-3, -2, 4, 0, 5]
```

The objective is to find the group with the maximum strength, which is the largest product of the students' scores. We will walk through each step of the solution approach with this example.

1. **Sorting**: First, we sort the array in ascending order, resulting in:

   ```
   [-3, -2, 0, 4, 5]
   ```

2. **Handling Special Cases**: Our array has more than one number, and there are positive numbers that are not followed by zeros. Thus, we have no special case that immediately determines the result.

3. **Iterating and Multiplying**: We start with an answer variable `ans` initialized to 1.

4. **Utilizing Negative Numbers in Pairs**: We begin iterating from the start of the sorted array. We find that `-3` and `-2` are two consecutive negative numbers:

   ```
   Product = 1 * (-3) * (-2) = 6
   ```

   We pair these together and move two positions forward in the array.

5. **Skipping Non-contributing Numbers**: The next number is `0`, which does not contribute to the product. We skip it:

   ```
   Skipping 0.
   ```

6. **Including Positive Numbers in the Product**: The next numbers are `4` and `5`, which are positive. We multiply them to our current product:

   ```
   Product = 6 * 4 * 5 = 120
   ```

   We process each number, moving one step forward in the array after each multiplication.

7. **Return Result**: After we've processed all elements, the final product `ans` equals `120`. This is the maximum strength we can obtain from any grouping of these scores.

Thus, for our example array `[-3, -2, 4, 0, 5]`, the maximum strength of a group is `120`.

Python Solution

```python
from typing import List  # Ensure List is imported from the typing module for type annotations

class Solution:
    def max_strength(self, nums: List[int]) -> int:
        # Sort the input list to group negatives, zeros and positives
        nums.sort()
        # Get the length of the list
        n = len(nums)

        # If there is only one element, its own value is the max strength
        if n == 1:
            return nums[0]

        # If the list has only zeros after the first, max strength is 0
        if nums[1] == nums[-1] == 0:
            return 0

        # Initialize the answer with 1 as a neutral multiplier and start index with 0
        answer, index = 1, 0

        # Iterate over the list to calculate the max strength
        while index < n:
            # If current and next numbers are negative, multiply them for a positive product
            if nums[index] < 0 and index + 1 < n and nums[index + 1] < 0:
                answer *= nums[index] * nums[index + 1]
                index += 2  # Move two steps forward as we used two numbers
            # If current number is negative or zero just skip it by incrementing the index
            elif nums[index] <= 0:
                index += 1
            # If current number is positive, include it in the product by multiplying
            else:
                answer *= nums[index]
                index += 1  # Move one step forward after using the number

        # Return the calculated max strength
        return answer

```

Java Solution

```java
import java.util.Arrays; // Import the Arrays library to use its sort method

class Solution {
    /**
     * Calculates the maximum strength by multiplying elements in a specific way.
     * Negative pairs are multiplied together, and all non-negative numbers are multiplied by the result.
     *
     * @param numbers Array of integers.
     * @return The maximum strength as computed by the algorithm.
     */
    public long maxStrength(int[] numbers) {
        // First, sort the input array
        Arrays.sort(numbers);

        // Get the number of elements in the array
        int length = numbers.length;

        // If there is only one element, return it as the result
        if (length == 1) {
            return numbers[0];
        }

        // If the first two elements are zero and the last element is also zero, return 0 as the result
        if (numbers[1] == 0 && numbers[length - 1] == 0) {
            return 0;
        }

        // This will hold our final result, starting with 1 (the multiplicative identity)
        long result = 1;

        // Index counter to traverse the array
        int i = 0;

        // Loop through all elements in the array
        while (i < length) {
            // If we encounter two consecutive negative numbers, multiply them together and
            // skip to the number after the second negative number
            if (numbers[i] < 0 && i + 1 < length && numbers[i + 1] < 0) {
                result *= (long)numbers[i] * numbers[i + 1]; // Cast to long to prevent integer overflow
                i += 2;
            }
            // If we encounter a single negative or zero, just skip it
            else if (numbers[i] <= 0) {
                i += 1;
            }
            // If the number is positive, we multiply it to result
            else {
                result *= numbers[i];
                i += 1;
            }
        }

        // Now we return the calculated result
        return result;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    long long maxStrength(vector<int>& nums) {
        // Sort the input numbers in non-decreasing order
        sort(nums.begin(), nums.end());

        // Get the size of nums vector
        int numSize = nums.size();

        // If there's only one number, return it as the answer
        if (numSize == 1) {
            return nums[0];
        }

        // If the smallest and the largest numbers are zeros, return 0
        if (nums[1] == 0 && nums[numSize - 1] == 0) {
            return 0;
        }

        // Initialize the answer with 1 (multiplicative identity)
        long long answer = 1;

        int index = 0;
        while (index < numSize) {
            // If current and next numbers are negative, pair them and multiply
            if (nums[index] < 0 && index + 1 < numSize && nums[index + 1] < 0) {
                answer *= static_cast<long long>(nums[index]) * nums[index + 1];
                index += 2; // Move past the paired negative numbers
            } else if (nums[index] <= 0) {
                // Skip the number if it is zero or the last negative number without a pair
                index += 1;
            } else {
                // If the number is positive, multiply it to the answer
                answer *= nums[index];
                index += 1; // Move to the next number
            }
        }

        // Return the computed max strength
        return answer;
    }
};

```

Typescript Solution

```typescript
function maxStrength(nums: number[]): number {
    // Sort the array in non-descending order to organize negative and positive numbers
    nums.sort((a, b) => a - b);

    // Obtain the length of the array
    const n: number = nums.length;

    // If there's only one element, it's the max product by default
    if (n === 1) {
        return nums[0];
    }

    // If the second element is 0 and the last is 0 after sorting,
    // it means all elements are 0, so the max product is 0
    if (nums[1] === 0 && nums[n - 1] === 0) {
        return 0;
    }

    // Initialize answer as 1 for multiplication
    let maxProduct: number = 1;

    // Iterate over the array to calculate the maximum strength
    for (let i = 0; i < n; ++i) {
        // If the current and next elements are negative,
        // their product is positive and should be included in the maxProduct.
        if (nums[i] < 0 && i + 1 < n && nums[i + 1] < 0) {
            maxProduct *= nums[i] * nums[i + 1];
            i++; // Skip the next element as it's already included in the product
        } else if (nums[i] > 0) {
            // Positive numbers always contribute to increasing the maxProduct
            maxProduct *= nums[i];
        }
        // Note: If it's 0 or a single negative number, it's not included
        // since it wouldn't contribute to the maxProduct positively
    }

    // Return the calculated maximum strength
    return maxProduct;
}

```

Time and Space Complexity

The provided code snippet takes a list of integers, sorts them, and then iterates over them to calculate the product of some numbers under certain conditions. Here's the complexity analysis:

- **Time Complexity:**

The time complexity of sorting the list is `O(n log n)`, where `n` is the number of elements in the input list. The sorting is the most significant operation in terms of complexity.

After sorting, the code iterates through the sorted list once to calculate the product, which takes `O(n)` time. Since `O(n log n)` is the dominant factor, the overall time complexity is `O(n log n)`.

- **Space Complexity:**

The space complexity is `O(1)` because the algorithm only uses a fixed amount of additional space, like variables for indexing (`i`) and the result (`ans`), besides the input list itself.

In summary:

- Time Complexity: `O(n log n)`
- Space Complexity: `O(1)`