2708. Maximum Strength of a Group Medium Greedy Backtracking Sorting Array 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

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.

a group of students that has the maximum possible strength. A group's strength is the product (multiplication) of the scores of all the

Intuition

following rules about multiplying numbers: Multiplying two positive numbers gives a larger number. Multiplying two negative numbers gives a positive number.

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

- 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.

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.

Multiplying a positive and a negative number results in a negative number.

Given these rules, we iterate through the sorted array:

2. Once sorted, we ignore os 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

- 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.

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

- Finally, we keep calculating the product as we go through these steps to arrive at the maximum strength.
- The implementation makes use of a few key algorithmic concepts, namely sorting and a single pass approach to maximize the

access positive numbers in the correct sequence.

If after the first positive number all other numbers are zeros, the maximum strength is zero.

a variable ans to 1 (the identity element for multiplication) to keep track of the ongoing product.

zero), it will not contribute to maximizing the product. We skip it and move on.

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.

1 n = len(nums)

return nums[0]

return 0

i += 2

increase the product.

i += 1

Example Walkthrough

1 [-3, -2, 4, 0, 5]

ans *= nums[i]

1 else:

4 if nums[1] == nums[-1] == 0:

2 if n == 1:

1 nums.sort()

Solution Approach

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.

3. Iterating and Multiplying: We then enter a loop where we iterate over the sorted numbers to calculate the product. We initialize

1 elif nums[i] <= 0: 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

The solution employs a single pass of the sorted array, multiplying pairs of negative numbers and any positive numbers while

This approach ensures that we are using each negative number (if any) in the most optimized way by pairing them and that every

skipping any non-contributing elements, to efficiently calculate the maximum strength of a possible group.

positive number, which always contributes positively to the product, is included in the final calculation.

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

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.

through each step of the solution approach with this example.

5. Skipping Non-contributing Numbers: If a number is not followed by another negative number (it's a standalone negative or

```
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.
  1 return ans
```

1 if nums[i] < 0 and i + 1 < n and nums[i + 1] < 0:

ans *= nums[i] * nums[i + 1]

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

The objective is to find the group with the maximum strength, which is the largest product of the students' scores. We will walk

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

1 Skipping 0.

Python Solution

class Solution:

18

19

30

31

32

33

34

35

36

37

9

10

11

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

41

42

43

9

10

11

12

13

14

15

16

17

18

19

20

28

29

30

31

34

35

36

37

38

40

};

Typescript Solution

if (n === 1) {

return 0;

return maxProduct;

function maxStrength(nums: number[]): number {

if (nums[1] === 0 && nums[n - 1] === 0) {

// Initialize answer as 1 for multiplication

nums.sort((a, b) => a - b);

return nums[0];

let maxProduct: number = 1;

} else if (nums[i] > 0) {

maxProduct *= nums[i];

// Return the calculated maximum strength

dominant factor, the overall time complexity is $O(n \log n)$.

// Obtain the length of the array

const n: number = nums.length;

// Sort the array in non-descending order to organize negative and positive numbers

i++; // Skip the next element as it's already included in the product

// Positive numbers always contribute to increasing the maxProduct

// Note: If it's 0 or a single negative number, it's not included

// since it wouldn't contribute to the maxProduct positively

// If there's only one element, it's the max product by default

// 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

maxProduct *= nums[i] * nums[i + 1];

nums.sort()

n = len(nums)

answer, index = 1, 0

return answer

Java Solution

/**

class Solution {

product:

consecutive negative numbers:

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

1 Product = 6 * 4 * 5 = 120

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:

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

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.

Sort the input list to group negatives, zeros and positives

If there is only one element, its own value is the max strength

def max_strength(self, nums: List[int]) -> int:

answer *= nums[index]

Return the calculated max strength

* @param numbers Array of integers.

public long maxStrength(int[] numbers) {

// Get the number of elements in the array

// Index counter to traverse the array

else if (numbers[i] <= 0) {</pre>

result *= numbers[i];

// Loop through all elements in the array

// First, sort the input array

int length = numbers.length;

return numbers[0];

Arrays.sort(numbers);

if (length == 1) {

return 0;

long result = 1;

while (i < length) {</pre>

i += 2;

i += 1;

i += 1;

else {

int i = 0;

Get the length of the list

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

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

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

If current number is positive, include it in the product by multiplying

index += 1 # Move one step forward after using the number

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

* @return The maximum strength as computed by the algorithm.

// If there is only one element, return it as the result

// skip to the number after the second negative number

if (numbers[i] < 0 && i + 1 < length && numbers[i + 1] < 0) {

// If we encounter a single negative or zero, just skip it

// If the number is positive, we multiply it to result

if (numbers[1] == 0 && numbers[length - 1] == 0) {

* 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.

// If the first two elements are zero and the last element is also zero, return 0 as the result

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

// If we encounter two consecutive negative numbers, multiply them together and

result *= (long)numbers[i] * numbers[i + 1]; // Cast to long to prevent integer overflow

if n == 1: return nums[0] 12 # If the list has only zeros after the first, max strength is 0 if nums[1] == nums[-1] == 0: return 0

```
20
           # Iterate over the list to calculate the max strength
21
           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:</pre>
24
25
                    answer *= nums[index] * nums[index + 1]
26
                    index += 2 # Move two steps forward as we used two numbers
27
                # If current number is negative or zero just skip it by incrementing the index
                elif nums[index] <= 0:</pre>
28
                    index += 1
29
```

```
52
53
54
55
56 }
```

```
// Now we return the calculated result
             return result;
 57
C++ Solution
  1 class Solution {
    public:
         long long maxStrength(vector<int>& nums) {
             // Sort the input numbers in non-decreasing order
             sort(nums.begin(), nums.end());
  6
             // Get the size of nums vector
             int numSize = nums.size();
  8
  9
 10
             // If there's only one number, return it as the answer
             if (numSize == 1) {
 11
 12
                 return nums[0];
 13
 14
 15
             // If the smallest and the largest numbers are zeros, return 0
             if (nums[1] == 0 \&\& nums[numSize - 1] == 0) {
 16
 17
                 return 0;
 18
 19
 20
             // Initialize the answer with 1 (multiplicative identity)
             long long answer = 1;
 21
 22
 23
             int index = 0;
 24
             while (index < numSize) {</pre>
 25
                 // If current and next numbers are negative, pair them and multiply
                 if (nums[index] < 0 \& index + 1 < numSize & nums[index + 1] < 0) {
 26
                     answer *= static_cast<long long>(nums[index]) * nums[index + 1];
 27
 28
                     index += 2; // Move past the paired negative numbers
 29
                 } else if (nums[index] <= 0) {</pre>
                     // Skip the number if it is zero or the last negative number without a pair
 30
 31
                     index += 1;
 32
                 } else {
 33
                     // If the number is positive, multiply it to the answer
 34
                     answer *= nums[index];
                     index += 1; // Move to the next number
 35
 36
 37
 38
 39
             // Return the computed max strength
 40
             return answer;
```

22 // Iterate over the array to calculate the maximum strength 23 for (let i = 0; i < n; ++i) { // If the current and next elements are negative, 24 // their product is positive and should be included in the maxProduct. if $(nums[i] < 0 \&\& i + 1 < n \&\& nums[i + 1] < 0) {$ 26

```
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 0(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
```

The space complexity is 0(1) because the algorithm only uses a fixed amount of additional space, like variables for indexing (1) and

 Time Complexity: 0(n log n) Space Complexity: 0(1)

Space Complexity:

In summary:

the result (ans), besides the input list itself.