2149. Rearrange Array Elements by Sign

Medium Array Two Pointers Simulation

Problem Description

The task involves an integer array `nums` which has an even number of elements, split equally between positive and negative integers. The goal is to rearrange `nums` so that:

1. Each consecutive pair of integers have opposite signs, meaning a positive integer is always followed by a negative integer and vice versa.
2. The order of integers with the same sign should remain the same as in the original array.
3. The first integer in the rearranged array should be positive.

Our objective is to generate this modified array that satisfies all of the conditions stated above.

Intuition

To solve this problem, we can approach it by separating the positive and negative numbers while maintaining their original order. Since the array is guaranteed to have an equal number of positive and negative numbers and the array starts with a positive number, we can alternate placing positive and negative numbers to satisfy the condition that pairs must have opposite signs.

The process can be visualized more easily by thinking of two queues: one for positive numbers and one for negative numbers. We pick the numbers from each queue alternately and place them sequentially in the result array, ensuring that positive numbers are placed at even indices starting from `0`, and negative numbers are placed at odd indices starting from `1`.

By doing this, we leverage the fact that the indices used will ensure that positive and negative numbers are always paired (as they occupy adjacent slots in the array) while their relative order among positives and negatives is preserved.

The solution code realizes this conceptual process by using two pointers `i` and `j` to represent the positions in the result array where the next positive or negative number, respectively, will be placed. The pointers start from `i` for `i` (positive) and `1` for `j` (negative) and are incremented by `2` after each number is placed to maintain the required conditions. By iterating through the input array once and distributing numbers according to their sign, we can complete the rearrangement in one pass, resulting in an efficient and straightforward solution.

Solution Approach

The solution provided follows a simple yet effective approach that takes advantage of the array's specific constraints and properties. Here is a detailed explanation of how the solution is implemented:

1. We initialize a new array `ans` with the same length as the input array `nums`. This array will hold the rearranged elements.
2. We create two pointers `i` and `j`. The pointer `i` starts at `0` and will be used to place positive integers at even indices of `ans`. The pointer `j` starts at `1` and will be used for negative integers at odd indices of `ans`.
3. We then iterate over the original array `nums`. For each number, we check if it is positive or negative.
4. If the number `num` is positive, we place it at `ans[i]`, and then increase `i` by `2`. This ensures that the next positive number will be placed two indices later, thereby maintaining the alternating positive-negative pattern.
5. If the number `num` is negative, we follow a similar process by placing it at `ans[j]` and incrementing `j` by `2`.
6. The loop continues until all elements from `nums` have been placed into `ans`. Thanks to the dual-pointer approach, there is no need for extra checks since the conditions stated in the problem guarantee that the final array will start with a positive number and exhibit the alternating sign pattern.
7. Once the loop is complete, the `ans` array, which is now a correctly rearranged version of `nums`, is returned.

In terms of algorithms and data structures, this solution primarily relies on array manipulation with pointers. We use a deterministic pattern to distribute elements and do not need additional data structures, like stacks or queues, because we have the guarantee of an equal number of positive and negative numbers.

In summary, this solution takes a linear-time algorithm (`O(n)`) as we pass through the array exactly once. Its space complexity is also linear (`O(n)`) due to the additional `ans` array used to store the rearranged elements.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following `nums` array:

```
nums = [3, -1, 2, -2]
```

Following the steps of our solution approach:

1. We initialize our answer array `ans` to be of the same size as `nums`. Thus `ans` starts as `[0, 0, 0, 0]`.
2. We set up two pointers, `i` starting at `0` for positive numbers, and `j` starting at `1` for negative numbers.
3. We iterate over `nums`. The first number is `3`, which is positive. We place it at `ans[i]` (where `i` is `0`), so `ans` becomes `[3, 0, 0, 0]`, and then we increase `i` by `2`. Now `i` is `2`.
4. Next number is `-1`, which is negative. We place it at `ans[j]` (where `j` is `1`), so `ans` becomes `[3, -1, 0, 0]`, and then we increase `j` by `2`. Now `j` is `3`.
5. We continue and encounter the positive number `2`. We place it at `ans[i]` (where `i` is `2`), making `ans` `[3, -1, 2, 0]`, and increase `i` by `2` again. However, `i` is now `4`, which is out of bounds for this array so we won't use `i` anymore.
6. Finally, `-2` is negative and goes to `ans[j]` (where `j` is `3`), giving us `ans` `[3, -1, 2, -2]`. Incrementing `j` by `2` doesn't matter anymore since we've finished processing the array.

The loop finishes with all elements placed correctly, resulting in an array where each positive number is followed by a negative one. The processed `ans` array `[3, -1, 2, -2]` is now returned. This array is the rearranged version of `nums` with alternating signs, starting with a positive integer.

This example has followed the solution steps exactly and provided the desired output in accordance with the original problem's requirements.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def rearrangeArray(self, nums: List[int]) -> List[int]:
        # Initialize an array of the same length as `nums` with all elements set to 0
        rearranged = [0] * len(nums)

        # `positive_index` tracks the next position for a positive number
        # `negative_index` tracks the next position for a negative number
        positive_index, negative_index = 0, 1

        # Iterate over all numbers in the given list
        for num in nums:
            if num > 0:
                # If the current number is positive, place it in the next available positive index
                rearranged[positive_index] = num
                # Increment the positive index by 2 to point to the next position for a positive number
                positive_index += 2
            else:
                # If the current number is negative, place it in the next available negative index
                rearranged[negative_index] = num
                # Increment the negative index by 2 to point to the next position for a negative number
                negative_index += 2

        # Return the rearranged array where positive and negative numbers alternate
        return rearranged
```

Java

```java
class Solution {
    public int[] rearrangeArray(int[] nums) {
        // Initialize a new array to hold the rearranged elements
        int[] rearrangedArray = new int[nums.length];

        // Two pointers to place positive and negative numbers in the array.
        // Positives will be placed at even indices and negatives at odd indices.
        int positiveIndex = 0, negativeIndex = 1;

        // Iterate through all the numbers in the input array.
        for (int num : nums) {
            if (num > 0) {
                // When we encounter a positive number, we place it at the next even index
                rearrangedArray[positiveIndex] = num;
                positiveIndex += 2; // Move the pointer to the next position for a positive number
            } else {
                // When we encounter a negative number, we place it at the next odd index
                rearrangedArray[negativeIndex] = num;
                negativeIndex += 2; // Move the pointer to the next position for a negative number
            }
        }

        // Return the rearranged array where no two consecutive numbers have the same sign
        return rearrangedArray;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    // This method rearranges the elements of the input array such that
    // positive and negative numbers alternate, beginning with a positive number.
    // It expects a vector of integers and returns the rearranged vector.
    std::vector<int> rearrangeArray(std::vector<int>& nums) {
        std::vector<int> rearranged(nums.size()); // Create a new vector for rearranged elements
        int positiveIndex = 0; // Initialize index for placing positive numbers, starting from position 0
        int negativeIndex = 1; // Initialize index for placing negative numbers, starting from position 1

        // Iterate over each number in the input array
        for (int num : nums) {
            if (num > 0) {
                // If current number is positive, place it at the next available positive index
                rearranged[positiveIndex] = num;
                positiveIndex += 2; // Increment the position by 2 to skip the next negative place
            } else {
                // If current number is negative, place it at the next available negative index
                rearranged[negativeIndex] = num;
                negativeIndex += 2; // Increment the position by 2 to skip the next positive place
            }
        }
        // Return the rearranged vector
        return rearranged;
    }
};
```

TypeScript

```typescript
function rearrangeArray(nums: number[]): number[] {
    // Initialize an empty array to store the rearranged elements
    let rearranged = [];
    // Initialize two pointers to fill positive and negative numbers respectively
    let positiveIndex = 0,
        negativeIndex = 1;

    // Iterate through each number in the input array
    for (let num of nums) {
        if (num > 0) {
            // If the current number is positive, place it at the next available positive index
            rearranged[positiveIndex] = num;
            positiveIndex += 2; // Increment the positive index by 2 to maintain alternating positions
        } else {
            // If the current number is negative, place it at the next available negative index
            rearranged[negativeIndex] = num;
            negativeIndex += 2; // Increment the negative index by 2 to maintain alternating positions
        }
    }

    // Return the rearranged array with alternated positive and negative numbers
    return rearranged;
}
```

```python
from typing import List

class Solution:
    def rearrangeArray(self, nums: List[int]) -> List[int]:
        # Initialize an array of the same length as `nums` with all elements set to 0
        rearranged = [0] * len(nums)

        # `positive_index` tracks the next position for a positive number
        # `negative_index` tracks the next position for a negative number
        positive_index, negative_index = 0, 1

        # Iterate over all numbers in the given list
        for num in nums:
            if num > 0:
                # If the current number is positive, place it in the next available positive index
                rearranged[positive_index] = num
                # Increment the positive index by 2 to point to the next position for a positive number
                positive_index += 2
            else:
                # If the current number is negative, place it in the next available negative index
                rearranged[negative_index] = num
                # Increment the negative index by 2 to point to the next position for a negative number
                negative_index += 2

        # Return the rearranged array where positive and negative numbers alternate
        return rearranged
```

Time and Space Complexity

The provided Python code aims to rearrange an array such that positive and negative elements are placed alternatively, starting with a positive element. Here's the analysis of its time and space complexity:

Time Complexity:

The code utilizes a single `for` loop that iterates over the entire list `nums`, meaning that each element in the original list is looked at exactly once. This results in a linear time complexity relative to the size of the input list. Therefore, the time complexity is `O(n)`, where `n` is the length of the list `nums`.

Space Complexity:

The space complexity involves analyzing both the input space and the additional space used by the algorithm excluding the input and output. The space taken by the list `nums` is not considered extra space as it is the input.

The code uses `ans`, an auxiliary list of the same length as `nums`, to store the rearranged elements. No additional data structures that grow with the input size are used; therefore, the extra space used by the code is proportional to the input size. Hence, the space complexity for the auxiliary space is `O(n)`.

In some cases, the output space is not considered in space complexity analysis. If the output space is not to be considered, only a constant amount of extra space is used for the variables `i` and `j`. This would make the space complexity `O(1)`. However, if the output space is included, then the total space complexity, accounting for both the input and the created returned list `ans`, would indeed be `O(n)`.

The choice of whether to consider the output space depends on the context or the specific definitions followed.