905. Sort Array By Parity

Easy Array Two Pointers Sorting

Problem Description

The LeetCode problem at hand requires us to rearrange the elements of a given integer array `nums`. The task is to move all even numbers to the front (beginning) of the array and place all odd numbers towards the rear (end) of the array. It's important to note that the problem does not require the numbers to be sorted within the even or odd groups, which means that the relative order between the even or odd elements does not matter. We just need to ensure that all the even integers appear before any odd integer in the resulting array. There are no constraints for the order of elements within their respective groups (even or odd), and thus any configuration that satisfies the condition will count as a correct answer.

Intuition

The solution to this problem is built on the two-pointer technique. We strategically place two pointers (`i` and `j`) at the start and end of the array respectively. The idea is to increment `i` (the left pointer) until we find an odd number and decrement `j` (the right pointer) until we find an even number. When both conditions are met, we have an odd number at the left pointer and an even number at the right pointer, and we swap them. This is based on the intuition that even numbers should be at the front and odd numbers at the back.

By incrementing and decrementing the pointers only when the even/odd conditions are not met, we ensure the pointers will eventually scan through the whole array. The process continues until the two pointers meet or cross each other, which indicates all even elements have been moved ahead of the odd elements. We use bitwise AND, `nums[i] & 1`, to quickly determine if a number is odd (if the operation results in `1`) or even (if it results in `0`). After finishing the algorithm, we return the modified array which now has all even integers at the beginning followed by odd integers.

Solution Approach

The implementation of the solution leverages the two-pointer technique, which is a common pattern used for array manipulation problems. This technique helps us to work with two different positions of the array simultaneously. Here's the breakdown of the steps involved, using the algorithm shown in the Reference Solution Approach:

1. Initialize two pointers `i` at `0` (start of the array) and `j` at `len(nums) - 1` (end of the array). This is where we will begin our processing from both ends of the array.
2. Loop through the array using a `while` loop that runs as long as `i < j`. This condition ensures that we keep processing until both pointers meet or cross, which means we have compared all elements.
3. Inside the loop:
   - We check if `nums[i]` is odd by using `nums[i] & 1`. Using the bitwise AND operation with `1` gives a result of `1` if the last bit is `1` (which means the number is odd), and `0` otherwise (which means the number is even).
   - If `nums[i]` is odd, we need to move this number towards the end of the array. We swap `nums[i]` with `nums[j]` (the current element at the pointer from the end of the array) using the tuple unpacking syntax `nums[i], nums[j] = nums[j], nums[i]`.
   - After swapping, we decrement `j` by `1` (`j -= 1`) to move the end pointer one step to the left, as we've just placed an odd number in its correct position towards the end.
   - If `nums[i]` is not odd (i.e., it's even), we simply increment `i` by `1` (`i += 1`) to continue the loop and look for the next odd number, as even numbers are already in their correct position at the start of the array.
4. The loop continues until `i` is no longer less than `j`, at which point we've achieved our goal of moving all even numbers to the front and all odd numbers to the end of the array.
5. Finally, we return the modified array `nums`.

This approach efficiently uses the two-pointer technique without the need for extra space (except for a few temporary variables), and it operates directly on the input array, resulting in an in-place algorithm. Since each element is looked at most once, the time complexity of this solution is O(n), where n is the number of elements in the array.

Example Walkthrough

Let's consider the array `nums = [3, 8, 5, 13, 6, 12, 4, 1]` and walk through the solution approach detailed above using this example.

1. We initialize two pointers `i` at `0` (start of the array) and `j` at `len(nums) - 1`, which is `7` in this case.
2. We start our `while` loop with the condition `i < j`. Initially, `i` is `0` and `j` is `7`.
3. Inside the `while` loop:
   - We check `nums[i] & 1` to determine if `nums[i]` is odd.
   - For `nums[0]`, which is `3`, we find it to be odd (`3 & 1` equals `1`).
   - We swap `nums[i]` with `nums[j]`, so now our array looks like this: `[1, 8, 5, 13, 6, 12, 4, 3]`.
   - We then decrement `j` by `1`, so `j` now equals `6`.
4. The next iteration starts with `i = 0` and `j = 6`:
   - `nums[i]` is now `1`, which is odd (`1 & 1` equals `1`).
   - We swap `nums[i]` with `nums[j]`, leading to: `[4, 8, 5, 13, 6, 12, 1, 3]`.
   - We again decrement `j` by `1`, and `j` is now `5`.
5. We continue the loop:
   - Now, `nums[i]` is `4`. It's even (`4 & 1` equals `0`) so we don't swap. We just increment `i` by `1`, resulting in `i = 1`.
   - During the same iteration, `nums[1]` is `8`, which is also even. So `i` is incremented again to `i = 2`.
6. With `i = 2` and `j = 5`, the loop continues:
   - `nums[2]` is `5`, which is odd.
   - We swap `nums[2]` with `nums[5]`, making the array `[4, 8, 12, 13, 6, 5, 1, 3]`.
   - We decrement `j` to `4`.
7. We iterate further:
   - `i` is `2`, and `j` is `4`.
   - For `i = 2`, `nums[i]` is `12` (even), so we increment `i` to `3`.
   - No swap needed as `nums[i]` is now `13` (odd) and `nums[j]` is `6` (even).
8. With `i = 3` and `j = 4`, we perform our last iteration:
   - We find `nums[3]` is odd, so we swap `nums[3]` and `nums[4]`, resulting in `[4, 8, 12, 6, 13, 5, 1, 3]`.
   - We decrement `j` to `3`.
9. At this point, `i` equals `j`. According to our loop condition `i < j`, the loop terminates.
10. We have successfully moved all even numbers to the front and all odd numbers to the end of the array.

The resulting array is `[4, 8, 12, 6, 13, 5, 1, 3]`, which meets the requirements of the problem. This example demonstrates how the two-pointer technique applied in this algorithm effectively separates even and odd numbers within a single pass through the array without the need for any additional space, adhering to a linear time complexity O(n).

Solution Implementation

Python

```python
class Solution:
    def sortArrayByParity(self, nums: List[int]) -> List[int]:
        # Initialize two pointers, left at the start of the list and right at the end
        left, right = 0, len(nums) - 1

        # Loop through the array until the two pointers meet
        while left < right:
            # If the number at the current left pointer is odd,
            # we swap it with the number at the right pointer.
            if nums[left] % 2 == 1:
                # Swap the elements at left and right pointers
                nums[left], nums[right] = nums[right], nums[left]
                # Move the right pointer inwards, since we've placed an odd number at the end
                right -= 1
            else:
                # If the number at the left pointer is even, we move the left pointer inwards
                left += 1

        # Return the array which is now sorted by parity:
        # Even numbers at the front, odd numbers at the back
        return nums
```

Java

```java
class Solution {

    // This function takes an integer array 'nums' and returns an array
    // with all even integers followed by all odd integers.
    public int[] sortArrayByParity(int[] nums) {
        // Initialize two pointers: 'left' starting from the beginning and 'right' from the end of the array.
        int left = 0, right = nums.length - 1;

        // Continue until 'left' pointer is less than 'right' pointer.
        while (left < right) {
            // If the element at 'left' index is odd, swap it with the element at 'right' index.
            if (nums[left] % 2 == 1) {
                // Perform the swap of elements.
                int temp = nums[left];
                nums[left] = nums[right];
                nums[right] = temp;

                // Decrement 'right' pointer as we have moved an odd number to the correct side of array.
                right--;
            } else {
                // Increment 'left' pointer if the current element is even,
                // as it is already in the correct side of array.
                left++;
            }
        }
        // Return the rearranged array with all even numbers first followed by odd numbers.
        return nums;
    }
}
```

C++

```cpp
#include <vector> // Include the header for vector
using namespace std;

class Solution {
public:
    vector<int> sortArrayByParity(vector<int>& nums) {
        // Initialize two pointers.
        // 'left' for the start of the array.
        // 'right' for the end of the array.
        int left = 0, right = nums.size() - 1;

        // Loop until the two pointers meet.
        while (left < right) {
            // If the current element at 'left' is odd,
            // swap it with the element at 'right' and move 'right' backwards.
            if (nums[left] % 2 == 1) {
                swap(nums[left], nums[right]);
                --right;
            } else {
                // If the element is even, move 'left' forward.
                ++left;
            }
        }

        // Return the modified vector, which is sorted by parity.
        return nums;
    }
};
```

TypeScript

```typescript
/**
 * Sorts the array such that even numbers come before odd numbers,
 * and even and odd numbers each maintain their relative ordering.
 *
 * @param {number[]} nums - The array of numbers to be sorted.
 * @return {number[]} The sorted array.
 */
function sortArrayByParity(nums: number[]): number[] {
    let leftIndex: number = 0; // Index starting from the beginning of the array
    let rightIndex: number = nums.length - 1; // Index starting from the end of the array

    // Continue until the left index is no longer less than the right index
    while (leftIndex < rightIndex) {
        // Check if the current element at the left index is odd
        if (nums[leftIndex] % 2 !== 0) {
            // Swap the odd number from the beginning with the element at the right index
            [nums[leftIndex], nums[rightIndex]] = [nums[rightIndex], nums[leftIndex]];
            rightIndex--; // Move the right index one step to the left
        } else {
            leftIndex++; // If it's an even number, move the left index one step to the right
        }
    }

    return nums; // Return the reordered array
}

// The function can now be exported and used in other TypeScript modules
export { sortArrayByParity };
```

```python
class Solution:
    def sortArrayByParity(self, nums: List[int]) -> List[int]:
        # Initialize two pointers, left at the start of the list and right at the end
        left, right = 0, len(nums) - 1

        # Loop through the array until the two pointers meet
        while left < right:
            # If the number at the current left pointer is odd,
            # we swap it with the number at the right pointer.
            if nums[left] % 2 == 1:
                # Swap the elements at left and right pointers
                nums[left], nums[right] = nums[right], nums[left]
                # Move the right pointer inwards, since we've placed an odd number at the end
                right -= 1
            else:
                # If the number at the left pointer is even, we move the left pointer inwards
                left += 1

        # Return the array which is now sorted by parity:
        # Even numbers at the front, odd numbers at the back
        return nums
```

Time and Space Complexity

The provided code has a time complexity of `O(n)` where `n` is the length of the input list `nums`. This is because each element in the list is visited at most once by either the `i` or `j` pointers which move towards each other from opposite ends of the list.

The space complexity of the code is `O(1)` because it sorts the array in place without using any extra space proportional to the input size. Only a constant amount of additional memory space is used for the index variables `i` and `j`.