922. Sort Array By Parity II

Easy Array Two Pointers Sorting

Problem Description

The problem presents us with an array named `nums` containing an equal number of odd and even integers. The task is to arrange this array in such a way that the parity of the elements matches the parity of their indices. In essence, all the even-indexed positions must contain even numbers, and all the odd-indexed positions must be filled with odd numbers. The challenge is to figure out a way to reorder the array to meet this requirement, knowing that there is at least one way to achieve it because of the even distribution of odd and even numbers.

Intuition

The approach to solving this problem involves in-place reordering of the elements to meet our even-odd position requirements. Since we know half the numbers are even and half are odd, we understand that there must be a correct place for each number in the 'reordered' array. The intuition is to iterate over the even indices and whenever we encounter an odd number at an even index, we search for an even number at an odd index to swap with. This is why we have a secondary pointer `j` starting at index 1, the first odd index, and it moves in steps of 2 (to remain at odd indices). Every time we find an even number at an odd index, we swap it with the mispositioned odd number we found at the even index.

This swapping continues until all even indices hold even numbers (and thus, all odd indices hold odd numbers) since we are swapping odd for even, maintaining the parity balance. Once we're done with the loop, every even index `i` will have an even number and ever odd index will have an odd number, resulting in an array that satisfies the conditions of the problem.

Solution Approach

The solution provided uses a two-pointer approach to rearrange the array such that even-indexed elements are even, and odd-indexed elements are odd. The algorithm iterates through the array with these key steps:

1. We initialize two pointers, `i` and `j`. Here `i` starts at index 0 (the first even index) and will be incremented by 2 in each iteration to ensure it only points to even indices. Pointer `j` starts at index 1 (the first odd index) and will be moved in increments of 2 to jump to the next odd index when needed.
2. We loop over the array with `i` up to `n` (`n` being the length of the array), making steps of size 2.
3. At each iteration, we check if the number at index `i` is odd by observing the last bit `nums[i] & 1`. If it is equal to 1, it indicates that the number is odd and is in the wrong position since `i` is even.
4. When an odd number is observed at an even index `i`, we look for an even number at an odd index `j`. We perform an inner while loop to move `j` forward until we find an even number (also checking the last bit `nums[j] & 1`).
5. As soon as an even number is found at an odd index `j`, we swap the elements at indexes `i` and `j`. This is done using Python's tuple unpacking feature: `nums[i], nums[j] = nums[j], nums[i]`.
6. The loop continues until `i` reaches the end of the array. At this point, all the even indices have even numbers due to our swapping mechanism, which ensures that every time we encounter a number out of place, we switch it with its corresponding pair.

No additional data structures are needed, and the in-place algorithm performs swaps only when necessary. This leads to a space complexity of O(1) and a time complexity that is O(n) since each element is looked at once and each odd index is evaluated at most once.

The elegance of this solution comes from recognizing that, given the constraints, for every mispositioned odd number at an even index, there must exist a mispositioned even number at an odd index that we can swap it with. Through careful stepping of the `i` and `j` pointers and efficient swapping, we're able to rearrange the array as needed without the use of any extra space.

Example Walkthrough

Let's use the following small example to illustrate the solution approach:

Consider the array `nums = [3, 6, 1, 4, 7, 2]`. Here, we have an equal number of odd and even integers. The even indices are 0, 2, 4, and they should have even numbers, while the odd indices 1, 3, 5 should have odd numbers.

Following the steps from the solution approach:

1. Initialize two pointers: `i` (starting at 0) and `j` (starting at 1).
2. Begin looping over the array with `i` going from 0 to the length of the array, incremented by 2.
3. On the first iteration: `i` is 0 and `j` is 1.
   - `nums[i]` is 3, which is odd (`3 & 1` equals 1), so it's out of place.
4. Now, we need to find an even number at an odd index `j` to swap with `nums[i]`. Since `j` is already at an odd index and `nums[j]` is 6, which is even (`6 & 1` equals 0), we don't need to move `j`.
5. Swap the elements at indices `i` and `j`. After swapping, `nums` becomes `[6, 3, 1, 4, 7, 2]`.
6. Increment `i` by 2. Now, `i` is 2.
   - `nums[i]` is 1, which is odd and out of place at an even index.
7. Increment `j` by 2 to find the next even number. Now `j` is 3.
   - `nums[j]` is 4, which is even.
8. Swap the elements at indices `i` and `j`. The array `nums` is now `[6, 3, 4, 1, 7, 2]`.
9. Increment `i` by 2. Now, `i` is 4.
   - `nums[i]` is 7, which is odd and incorrectly placed at an even index.
10. Increment `j` by 2 to move to the next odd index. Now `j` is 5.
    - `nums[j]` is 2, which is even.
11. Swap the elements at indices `i` and `j`. The array `nums` is now `[6, 3, 4, 1, 2, 7]`.

Now each even index (0, 2, 4) has even numbers and each odd index (1, 3, 5) has odd numbers. The process is complete and the array `nums` is successfully reordered. The result is an array where the parity of the elements matches the parity of their indices, thus `[6, 3, 4, 1, 2, 7]`.

Solution Implementation

Python

```python
class Solution:
    def sortArrayByParityII(self, nums: List[int]) -> List[int]:
        # Get the length of the input list
        length = len(nums)
        # Initialize 'odd_index' to the first odd position
        odd_index = 1

        # Iterate through all even indices of the list
        for even_index in range(0, length, 2):
            # Check if the current even position contains an odd number
            if nums[even_index] % 2 == 1:
                # Look for an even number at odd positions
                while nums[odd_index] % 2 == 1:
                    # Increase 'odd_index' by 2 to check the next odd position
                    odd_index += 2

                # Swap the odd number at the even index with the even number at the odd index
                nums[even_index], nums[odd_index] = nums[odd_index], nums[even_index]

        # Return the sorted by parity list
        return nums
```

Java

```java
class Solution {
    public int[] sortArrayByParityII(int[] nums) {
        // i points to the next even index, j points to the next odd index
        int i = 0; // i should index even elements
        int j = 1; // j should index odd elements

        // Iterate over the array, considering only even indices for i
        while (i < nums.length) {
            // If the number at the current even index is odd,
            // we need to swap it with a number at an odd index.
            if (nums[i] & 1) != 0) {
                // Find the next number that's out of place at an odd index
                while ((nums[j] & 1) != 0) {
                    j += 2;
                }
                // Swap the numbers at indices i and j
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
            }
            // Move to the next even index
            i += 2;
        }
        // Return the sorted array with even indices containing even numbers
        // and odd indices containing odd numbers
        return nums;
    }
}
```

C++

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to sort the array so that at every even index we have an even number,
    // and at every odd index, we have an odd number.
    vector<int> sortArrayByParityII(vector<int>& nums) {
        // Traverse through the elements at even indices.
        for (int evenIndex = 0, oddIndex = 1; evenIndex < nums.size(); evenIndex += 2) {
            // Check if the current even index has an odd number.
            if (nums[evenIndex] & 1) == 1) {
                // Find the odd index that has an even number.
                while ((nums[oddIndex] & 1) == 1) {
                    oddIndex += 2;
                }
                // Swap the odd number at an even index with the even number at an odd index.
                swap(nums[evenIndex], nums[oddIndex]);
            }
        }
        // Return the sorted array.
        return nums;
    }
};
```

TypeScript

```typescript
/**
 * Sorts an array such that every even-indexed element is even and every odd-indexed element is odd.
 * @param {number[]} nums - The input array to be sorted in place.
 * @returns {number[]} The sorted array.
 */
function sortArrayByParityII(nums: number[]): number[] {
    // Initialize two pointers, `evenIndex` for even indexes and `oddIndex` for odd indexes.
    let evenIndex: number = 0;
    let oddIndex: number = 1;

    // Iterate over the array through the even indexes.
    while (evenIndex < nums.length) {
        // Check if the current even index contains an odd value.
        if ((nums[evenIndex] & 1) === 1) {
            // If so, look for an even value at the odd index positions.
            while ((nums[oddIndex] & 1) === 1) {
                oddIndex += 2;
            }
            // Swap the values at `evenIndex` and `oddIndex` to satisfy the condition.
            [nums[evenIndex], nums[oddIndex]] = [nums[oddIndex], nums[evenIndex]];
        }

        // Move the `evenIndex` pointer to the next even position.
        evenIndex += 2;
    }

    // Return the sorted array.
    return nums;
}

// Usage example:
// const sorted = sortArrayByParityII([4, 2, 5, 7]);
// console.log(sorted);
```

```python
class Solution:
    def sortArrayByParityII(self, nums: List[int]) -> List[int]:
        # Get the length of the input list
        length = len(nums)
        # Initialize 'odd_index' to the first odd position
        odd_index = 1

        # Iterate through all even indices of the list
        for even_index in range(0, length, 2):
            # Check if the current even position contains an odd number
            if nums[even_index] % 2 == 1:
                # Look for an even number at odd positions
                while nums[odd_index] % 2 == 1:
                    # Increase 'odd_index' by 2 to check the next odd position
                    odd_index += 2

                # Swap the odd number at the even index with the even number at the odd index
                nums[even_index], nums[odd_index] = nums[odd_index], nums[even_index]

        # Return the sorted by parity list
        return nums
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `nums`. This is because the outer loop runs for every second element (i.e., each even-indexed position), which results in `n/2` iterations. Inside the loop, we check if the current even-indexed number is odd. If it is odd, we perform a while loop to find the next odd-indexed position that contains an odd number to swap with.

The key point to note here is that each odd number is moved at most once since after it is moved to an even index, it doesn't need to be moved again. Therefore, the while loop altogether will also run for a maximum of `n/2` odd numbers throughout the entire execution of the algorithm.

Combining the outer and inner loop we still have a linear relationship with the number of elements `n`, leading to a total time complexity of `O(n)`.

Space Complexity

The space complexity of the code is `O(1)` since only a fixed number of extra variables (`n`, `j`) are used, and these do not grow with the input size. The swaps are done in-place, which means no additional space proportional to the input size is required.