Documentation for Wiggle Sort II Solution

Problem Overview

Wiggle Sort II is a problem where the goal is to rearrange an input array such that the elements alternate in a specific pattern: nums[0] < nums[1] > nums[2] < nums[3]... This ensures the values at odd indices are strictly greater than their neighboring even indices, creating a "wiggle" effect. The challenge lies in efficiently performing this reordering while ensuring all constraints are met, including cases with duplicate values.

Constraints and Challenges

The key constraints for this problem include:

1. The array can have up to 10^4 elements, making scalability a concern.
2. All values in the array are non-negative integers in the range [0,5000][0, 5000].
3. There is a guaranteed solution for every valid input array.
4. The problem requires an efficient approach, emphasizing O(n) time complexity or O(1) extra space.

The main challenge arises in ensuring duplicate values do not disrupt the wiggle pattern. For instance, merely sorting and rearranging the array without considering the alternation of smaller and larger values can lead to incorrect results.

Solution Approach

The solution is based on sorting the array to easily partition it into "smaller" and "larger" groups relative to the median. By dividing the sorted array into two halves, we ensure:

1. The "smaller" half contains elements less than or equal to the median.
2. The "larger" half contains elements greater than the median.

After partitioning, the elements are rearranged using a **virtual index mapping** where smaller values are placed at even indices (0, 2, 4,...) and larger values at odd indices (1, 3, 5,...). This strategy ensures the wiggle pattern is maintained.

Virtual Index Mapping

Virtual indexing is critical for achieving the desired rearrangement efficiently. Instead of directly modifying the array sequentially, we map indices such that the values alternate between smaller and larger groups:

- The smaller group fills the even indices, ensuring nums[i] < nums[i+1] for these indices.
- The larger group fills the odd indices, ensuring nums[i] > nums[i-1] for these indices.

This mapping avoids overwriting elements or violating the wiggle conditions, even when duplicates are present. Reversing the halves before filling the indices ensures a correct interleaving of values.

Handling Edge Cases

The solution accounts for edge cases such as:

1. **Duplicate Values**: By dividing and reversing the partitions, the solution ensures duplicates are placed in a way that maintains the wiggle condition.
2. **Small Arrays**: For arrays with fewer than three elements, the problem is trivial, and the sorted array already meets the wiggle conditions.
3. **Large Arrays:** The virtual indexing approach ensures scalability by minimizing operations and preserving in-place reordering.

Efficiency Considerations

The sorting step dominates the time complexity of this solution, resulting in an overall complexity of O(n log n). The rearrangement and virtual indexing are linear operations, contributing O(n) to the overall complexity. While this solution is not O(n), it is practical and straightforward to implement with reliable performance for large inputs. The space complexity is O(n) due to the need for temporary partitions, though an O(1) space solution is possible using advanced techniques like in-place partitioning.

Applications and Insights

The Wiggle Sort II problem highlights the importance of efficient data rearrangement in real-world scenarios. This concept is applicable in areas like data visualization, where alternating patterns improve interpretability, or in scheduling tasks that require balanced prioritization. The solution also demonstrates how sorting and indexing techniques can be combined to achieve specific data arrangements.