324. Wiggle Sort II

Medium | Array | Divide and Conquer | Quickselect | Sorting | Leetcode Link

Problem Description

The given problem asks us to take an integer array `nums` and reorder it so that it alternates between numbers that are less than and greater than their adjacent numbers. In other words, after reordering, each number at an even index (0, 2, 4, ...) should be less than its right neighbor, and every number at an odd index (1, 3, 5, ...) should be greater than its left neighbor. Formally, we need to rearrange `nums` such that `nums[0] < nums[1] > nums[2] < nums[3]...` and so on. We're also told that there is always a valid way to reorder the input array to meet these conditions.

Intuition

To arrive at a solution, we need to consider a way to make sure that for each pair of neighboring numbers, the even-indexed number is less than its neighbor, and the odd-indexed number is greater than its neighbor. One approach to achieve this is to sort the array first, which gives us a sequence of numbers in non-decreasing order. Once we have this sorted array, we can construct the result by taking elements from the sorted array and placing them into the `nums` array at the correct positions to create the alternating sequence (wiggle pattern).

The intuition behind using a sorted array is that, if we take the middle of the sorted array as the pivot, the smaller half will have the "less than" elements, and the larger half will have the "greater than" elements. By alternating placing elements from these two halves into the `nums` array, we enforce the wiggle property.

Here's the approach step by step:

1. Create a copy of the `nums` array and sort it.
2. Find the indices `i` and `j` which point to the middle element of the smaller half (if the length is odd, it's exactly the middle; if it's even, it's the last element of the "less than" side) and the last element of the sorted array, respectively.
3. Loop over the `nums` array, filling even indices with elements from the "less than" side (pointed by `i`) and odd indices with elements from the "greater than" side (pointed by `j`).
4. For each step, decrement the indices `i` and `j` to move towards the beginning of their respective halves.

Using this strategy ensures that each even-indexed element comes from the smaller half, guaranteeing it is smaller than the following element (which will come from the greater half). Consequently, every odd-indexed element comes from the larger half, ensuring it is greater than the preceding element (from the smaller half).

Solution Approach

The solution to the problem leverages a simple sorting algorithm and then applies an insightful pattern to reorder the `nums` array into a wiggle sequence. Here's how it works, with a detailed explanation of the code provided earlier.

1. **Sorting**: First, a sorted copy of the `nums` array is created, denoted in the code as `arr`. This step is accomplished using Python's built-in sorting function, which is typically implemented as Timsort and runs in O(N log N) time complexity. This sorted array is used as a reference to build the final wiggle sorted array.

2. **Index Calculation**: Before we start reordering, we calculate two indices, `i` and `j`. Index `i` is initialized to the mid-point of the first half of the sorted array. If the length of the array `n` is even, `i` would be `(n / 2) - 1`, and if `n` is odd, it would be `(n - 1) / 2`. This is done with the expression `(n - 1) >> 1`, which is a bit-shift operation equivalent to dividing by 2 and floor rounding the result. Index `j` is initialized to the last element of the sorted array, `n - 1`.

3. **Reordering into a Wiggle Sequence**: The code uses a loop to iterate through each position `k` in the original `nums` array. For even values of `k` (where `k % 2 == 0`), the element at the position `i` is placed into `nums[k]`. This ensures that every even-indexed position in `nums` will receive elements from the smaller half of the `arr`. After placing the element, `i` is decremented to move backwards through the first half of `arr`.

   On the other hand, for odd values of `k`, the element at position `j` is placed into `nums[k]`, which ensures that every odd-indexed position in `nums` will receive elements from the larger half of the `arr`. Similarly, `j` is decremented after placing the element to move backwards through the second half.

4. **In-Place Update**: The reordering of elements is done in place within the input array `nums`. This means the algorithm does not require additional space proportional to the input size, and the space complexity for this reordering is O(1), not counting the sorted copy of the array.

In summary, the solution algorithm uses a sort operation followed by a smart iterative reordering approach. By placing the smaller numbers at even indices and the larger numbers at odd indices, it achieves the desired wiggle sort pattern that alternates between "<" and ">" relations between adjacent elements.

Example Walkthrough

Let's consider the array `nums = [3, 5, 2, 1, 6, 4]` and walk through the solution approach to reorder it into a wiggle pattern.

1. **Sorting**: First, we create a sorted copy of the `nums` array.

```
arr = sorted(nums)  // arr becomes [1, 2, 3, 4, 5, 6]
```

2. **Index Calculation**: We calculate the two indices `i` and `j`. `nums` has 6 elements, so `n` is even.

```
i = (n / 2) - 1 = (6 / 2) - 1 = 2  // points to the third element in arr
j = n - 1 = 6 - 1 = 5              // points to the last element in arr
```

   So `i` starts at 2 (the element is 3 in the sorted `arr`), and `j` starts at 5 (the element is 6 in `arr`).

3. **Reordering into a Wiggle Sequence**: We loop through each position `k` in the original `nums` array, alternating writing values from `arr` starting with the `i` and `j` indices.

   For `k = 0` (even index):

```
nums[0] = arr[i] = 3
i = i - 1 = 1
```

   For `k = 1` (odd index):

```
nums[1] = arr[j] = 6
j = j - 1 = 4
```

   For `k = 2` (even index):

```
nums[2] = arr[i] = 2
i = i - 1 = 0
```

   For `k = 3` (odd index):

```
nums[3] = arr[j] = 5
j = j - 1 = 3
```

   For `k = 4` (even index):

```
nums[4] = arr[i] = 1
i = i - 1 = -1  // (we don't actually move to -1, as the loop ends)
```

   For `k = 5` (odd index):

```
nums[5] = arr[j] = 4
j = j - 1 = 2  // (we don't actually move to 2, as the loop ends)
```

4. **Final Output**: The result is an in-place update of `nums` array into the wiggle pattern:

```
nums = [3, 6, 2, 5, 1, 4]
```

   This final array satisfies the wiggle condition that for every even-indexed position `p`, `nums[p]` is less than `nums[p + 1]`, and for every odd-indexed position `q`, `nums[q]` is greater than `nums[q - 1]`.

Python Solution

```python
class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
        This method takes an array 'nums' and reorders it in-place to a wiggle sort order.
        Where nums[0] < nums[1] > nums[2] < nums[3]...
        """
        # Sort the array to make it easier to find the median.
        sorted_nums = sorted(nums)
        length = len(sorted_nums)
        # Find the midpoints for the smaller and larger halves
        # If 'length' is odd, 'mid' is the exact middle, else it's just before the middle
        mid = (length - 1) // 2
        end = length - 1

        # Reorder the array by placing the largest element at the end and the next
        # largest at the beginning, then the second-largest at index 2, and so on.
        for index in range(length):
            if index % 2 == 0:
                # Even index gets the next element from the smaller half
                nums[index] = sorted_nums[mid]
                mid -= 1
            else:
                # Odd index gets the next element from the larger half
                nums[index] = sorted_nums[end]
                end -= 1
        # The array is now reordered in-place

```

Java Solution

```java
import java.util.Arrays;

class WiggleSortSolution {

    /**
     * Sorts the given array into wiggle sort order.
     * A wiggle sort order means that nums[0] < nums[1] > nums[2] < nums[3]...
     * @param nums The input array to be wiggle sorted.
     */
    public void wiggleSort(int[] nums) {

        // Clone the original array to manipulate and sort without altering the argument array.
        int[] sortedArray = nums.clone();
        // Sort the cloned array in non-decreasing order.
        Arrays.sort(sortedArray);

        // Get the size of the array.
        int n = nums.length;
        // Find the mid point of the array to split the values.
        int midIndex = (n - 1) >> 1; // Equivalent to (n-1)/2
        // Set the index for the larger half of the values.
        int highIndex = n - 1;

        // Iterate over each index of the original 'nums' array.
        for (int k = 0; k < n; ++k) {
            if (k % 2 == 0) {
                // For even index, assign the next smaller value from the first half of 'sortedArray'.
                nums[k] = sortedArray[midIndex--];
            } else {
                // For odd index, assign the next larger value from the second half of 'sortedArray'.
                nums[k] = sortedArray[highIndex--];
            }
        }

        // Now 'nums' is restructured in place to follow wiggle sort order.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    void wiggleSort(vector<int>& nums) {
        // Copy the input array
        vector<int> sortedNums = nums;

        // Sort the copied array
        sort(sortedNums.begin(), sortedNums.end());

        // Get the size of the array
        int n = nums.size();

        // Calculate indices for odd and even position elements
        int midIndex = (n - 1) >> 1; // Right shift by 1 is equivalent to divide by 2
        int lastIndex = n - 1;

        // Iterate over the numbers to interleave them
        for (int k = 0; k < n; ++k) {
            if (k % 2 == 0) {
                // Even index, assign value from the first half of the sorted array
                nums[k] = sortedNums[midIndex--];
            } else {
                // Odd index, assign value from the second half of the sorted array
                nums[k] = sortedNums[lastIndex--];
            }
        }
        // Now 'nums' is wiggle sorted
    }
};

```

Typescript Solution

```typescript
/**
 * Wiggle Sort function to reorder an array so that nums[0] < nums[1] > nums[2] < nums[3]...
 * @param {number[]} nums - The array of numbers to sort in wiggle fashion
 * @returns {void} Modifies the input array in-place to satisfy the wiggle property
 */
function wiggleSort(nums: number[]): void {
    // Create an array to act as a bucket to count occurrences of each number
    const maxNumValue = 5000;
    const frequencyBucket: number[] = new Array(maxNumValue + 1).fill(0);

    // Fill the bucket with frequency of each number in `nums`
    for (const value of nums) {
        frequencyBucket[value]++;
    }

    const totalElements = nums.length;
    let currentValue = maxNumValue;

    // Fill odd positions with the next greatest element's occurrences
    for (let i = 1; i < totalElements; i += 2) {
        while (frequencyBucket[currentValue] === 0) { // Find the next greatest element
            --currentValue;
        }
        nums[i] = currentValue; // Place the number in the current position
        --frequencyBucket[currentValue]; // Decrease the bucket count for this number
    }

    // Fill even positions with the next greatest element's occurrences
    for (let i = 0; i < totalElements; i += 2) {
        while (frequencyBucket[currentValue] === 0) { // Find the next greatest element
            --currentValue;
        }
        nums[i] = currentValue; // Place the number in the current position
        --frequencyBucket[currentValue]; // Decrease the bucket count for this number
    }
}

```

Time and Space Complexity

The time complexity of the provided code is `O(n log n)`. This is because the code includes a sorting operation on the list `nums`, which typically has a time complexity of `O(n log n)` for the average sorting algorithm like Timsort used in Python's `sorted()` method.

After sorting, the code proceeds to rearrange the elements in a single pass through the list. Each iteration of the single pass takes constant time, so the loop has a time complexity of `O(n)`. However, since the sorting step is more dominant, the overall time complexity does not change and remains `O(n log n)`.

The space complexity of the code is `O(n)` because a new list `arr` of size `n` is created when sorting the `nums` list. Aside from this, the reassignment of values in `nums` is done in place and does not require any additional space that scales with the input size.