280. Wiggle Sort

Medium Greedy Array Sorting

Problem Description

The problem requires us to reorder an array `nums` so that adjacent elements follow a specific pattern: the element at an even index is not greater than the next element, and the element at an odd index is not less than the previous element. In other words, the elements should alternate between being less than or equal to and being greater than or equal to their neighboring elements, creating a "wiggle" pattern. This needs to be done in a way that modifies the array in-place, meaning without using an additional array to output the result. The pattern can be described as `nums[0] <= nums[1] >= nums[2] <= nums[3]...` for the entire array. It is guaranteed that there is at least one way to rearrange the array to satisfy the condition.

Intuition

The provided solution approach is straightforward and works by iterating through the array and making local swaps to enforce the wiggle pattern. It utilizes a greedy algorithm that checks at each step if the current element violates the wiggle condition. If it does, the algorithm swaps the current element with the previous one. The condition for odd and even indices is different:

- For odd indices (`1, 3, 5, ...`), if the current element is less than the previous one, a swap is needed to make sure the current (odd-indexed) element is greater or equal.
- For even indices (`2, 4, 6, ...`), if the current element is greater than the previous one, a swap is needed to make sure the current (even-indexed) element is less or equal.

By swapping only when the condition is violated, we ensure that the swapped elements will also satisfy the wiggle condition with their new neighbors. This happens because we are only ever swapping adjacent elements that were already checked in previous iterations. Since we are iterating from the second element to the end of the array, each element is considered in its turn, and we avoid the extra memory usage that would come from creating a second array to hold the rearranged elements.

Solution Approach

The solution uses a simple algorithm that is already described in the intuition. As for the implementation, it involves iterating through the array starting from the second element. During each iteration, the current element is compared with its predecessor, and a conditional swap is used to rectify the ordering if a wiggle violation is detected.

To implement this, the solution uses a single `for` loop that starts from index `1`, since index `0` has no preceding element and thus cannot violate the wiggle property.

Within the loop, the solution checks two conditions, each corresponding to whether the current index `i` is even or odd:

- For odd indices (when `i % 2 == 1`), if the current element `nums[i]` is less than the previous element `nums[i - 1]`, they are swapped. This ensures that at odd indices, the value at that index is always greater than or equal to its predecessor.
- For even indices (when `i % 2 == 0`), if the current element `nums[i]` is greater than the previous element `nums[i - 1]`, they are swapped. This ensures that at even indices, the value at that index is always less than or equal to its predecessor.

The use of the modulo operation `i % 2` helps to differentiate between even and odd indices. The swap operation itself is a standard technique and is implemented in Python by simply using tuple unpacking: `nums[i], nums[i - 1] = nums[i - 1], nums[i]`. This is equivalent to the algorithm having a temporary variable to hold one of the values during the swap but is more concise and idiomatic in Python.

No additional data structures are used, and the algorithm modifies the input list `nums` in place, resulting in a space complexity of O(1), since only a constant amount of extra space is used.

The time complexity of the algorithm is O(n), where n is the number of elements in the array. It's linear because the algorithm goes through the array only once, doing a constant amount of work for each element.

In summary, this algorithm efficiently enforces the wiggle condition using local swaps during a single pass through the array, requiring no extra memory apart from a few constant variables.

Example Walkthrough

Let's apply the solution approach to a small array `nums = [6, 4, 7, 5, 2]`. Our goal is to modify `nums` in place to meet the alternating pattern of `nums[0] <= nums[1] >= nums[2] <= nums[3]` and so on. Following the approach, we start at the second element (index `1`) and move through the array considering each element.

Iteration 1:

- Index `1` (value `4`) is odd.
- Check if `nums[1] < nums[0]` (is `4 < 6`? Yes).
- No swap needed since the condition for odd indices is already fulfilled.

After Iteration 1: `nums = [6, 4, 7, 5, 2]`

Iteration 2:

- Index `2` (value `7`) is even.
- Check if `nums[2] > nums[1]` (is `7 > 4`? Yes).
- Swap since this violates the condition for even indices.
- After swapping, `nums` becomes `[6, 7, 4, 5, 2]`.

After Iteration 2: `nums = [6, 7, 4, 5, 2]`

Iteration 3:

- Index `3` (value `5`) is odd.
- Check if `nums[3] < nums[2]` (is `5 < 4`? No).
- Swap since this violates the condition for odd indices.
- After swapping, `nums` becomes `[6, 7, 5, 4, 2]`.

After Iteration 3: `nums = [6, 7, 5, 4, 2]`

Iteration 4:

- Index `4` (value `2`) is even.
- Check if `nums[4] > nums[3]` (is `2 > 4`? No).
- No swap needed since the condition for even indices is already fulfilled.

Final result: `nums = [6, 7, 5, 4, 2]`

The example above demonstrates how the algorithm proceeds through the array, swapping elements as needed to create the "wiggle" pattern. By the final iteration, we have an array where each even index has a value not greater than its next element, and each odd index has a value not less than its previous element.

Solution Implementation

Python

```python
from typing import List  # Import List from typing for type annotations

class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
        This method sorts the input array 'nums' such that nums[0] <= nums[1] >= nums[2] <= nums[3]...
        The sort is done in-place to provide a 'wiggle' pattern.

        :param nums: List[int] - The list of numbers to be wiggled.
        :return: None - The input list is modified in-place.
        """

        # Loop through each element of the array starting from the second element.
        for index in range(1, len(nums)):
            # Check if the current index is odd.
            is_odd_index = index % 2 == 1

            # Wiaole condition for odd indices: if current element is less than the previous one.
            # Wiaole condition for even indices: if current element is areater than the previous one.
            if (is_odd_index and nums[index] < nums[index - 1]) or (not is_odd_index and nums[index] > nums[index - 1]):
                # Swap the elements to maintain the wiaole condition.
                nums[index], nums[index - 1] = nums[index - 1], nums[index]
        # Since the method modifies the input list in-place, there is no return statement required.
```

Java

```java
class Solution {
    // Function to wiaole sort an array where nums[0] < nums[1] > nums[2] < nums[3]...
    public void wiaoleSort(int[] nums) {
        // Loop through the array starting from index 1
        for (int i = 1; i < nums.length; ++i) {
            // Check if the current index is odd and the element is not areater than the previous element
            // or the current index is even and the element is not smaller than the previous element
            if ((i % 2 == 1 && nums[i] < nums[i - 1]) || (i % 2 == 0 && nums[i] > nums[i - 1])) {
                // If either condition is true, swap the current and previous elements
                swap(nums, i, i - 1);
            }
        }
    }

    // Helper function to swap two elements in the array
    private void swap(int[] nums, int i, int j) {
        int temp = nums[i]; // Store the value at index i
        nums[i] = nums[j]; // Set the value at index i to the value at index j
        nums[j] = temp; // Set the value at index j to the stored value of index i
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to wiaole sort an array where nums[0] < nums[1] > nums[2] < nums[3]...
    void wiaoleSort(vector<int>& nums) {
        // Loop through the array starting from index 1
        for (int i = 1; i < nums.size(); ++i) {
            // If 'i' is odd and the current element is less than the previous one.
            // or 'i' is even and the current element is areater than the previous one
            if ((i % 2 == 1 && nums[i] < nums[i - 1]) || (i % 2 == 0 && nums[i] > nums[i - 1])) {
                // Swap the current element with the previous one
                swap(nums, i, i - 1);
            }
        }
    }

private:
    // Helper function to swap two elements in the array
    void swap(vector<int>& nums, int i, int j) {
        int temp = nums[i]; // Store the value at index i in a temporary variable
        nums[i] = nums[j]; // Assign the value at index j to index i
        nums[j] = temp;    // Assign the stored value in temporary variable to index j
    }
};
```

TypeScript

```typescript
// Function to wiaole sort an array where nums[0] < nums[1] > nums[2] < nums[3]...
function wiaoleSort(nums: number[]): void {
    // Loop through the array starting from index 1
    for (let i = 1; i < nums.length; i++) {
        // Check if the current index is odd and the current element is not areater than the previous element
        // or the current index is even and the current element is not less than the previous element
        if ((i % 2 === 1 && nums[i] < nums[i - 1]) || (i % 2 === 0 && nums[i] > nums[i - 1])) {
            // If the element doesn't satisfy the wiggle condition, swap with the previous element
            swap(nums, i, i - 1);
        }
    }
}

// Helper function to swap two elements in the nums array
function swap(nums: number[], i: number, j: number): void {
    const temp = nums[i]; // Store the value at index i
    nums[i] = nums[j]; // Set the value at index i to the value at index j
    nums[j] = temp; // Set the value at index j to the stored value of index i
}
```

```python
from typing import List  # Import List from typing for type annotations

class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
        This method sorts the input array 'nums' such that nums[0] <= nums[1] >= nums[2] <= nums[3]...
        The sort is done in-place to provide a 'wiggle' pattern.

        :param nums: List[int] - The list of numbers to be wiggled.
        :return: None - The input list is modified in-place.
        """

        # Loop through each element of the array starting from the second element.
        for index in range(1, len(nums)):
            # Check if the current index is odd.
            is_odd_index = index % 2 == 1

            # Wiaole condition for odd indices: if current element is less than the previous one.
            # Wiaole condition for even indices: if current element is areater than the previous one.
            if (is_odd_index and nums[index] < nums[index - 1]) or (not is_odd_index and nums[index] > nums[index - 1]):
                # Swap the elements to maintain the wiaole condition.
                nums[index], nums[index - 1] = nums[index - 1], nums[index]
        # Since the method modifies the input list in-place, there is no return statement required.
```

Time and Space Complexity

The given Python code performs a wiggle sort on an array `nums` by swapping adjacent elements when they do not follow the "wiggle" property (`nums[i-1] < nums[i]` for odd `i` and `nums[i-1] > nums[i]` for even `i`). Each pair of adjacent elements is considered exactly once.

Time Complexity:

The time complexity of the code is `O(n)`, where `n` is the length of `nums`. This is because the algorithm iterates through the array once, and each iteration involves constant-time operations (simple comparisons and element swaps).

Space Complexity:

The space complexity of the code is `O(1)`. This algorithm modifies the `nums` array in place and does not require any additional space that scales with the size of the input array. The only extra space used is for the loop counter and temporary variables for swapping, which are constant.