2164. Sort Even and Odd Indices Independently

Easy Array Sorting

Problem Description

In this problem, we're given an array `nums` that we need to rearrange based on certain rules. Specifically, the indices of the array are divided into two groups – odd indices and even indices. Values at odd indices must be sorted in non-increasing order, meaning that each value should be less than or equal to the one before it. Conversely, the values at even indices must be sorted in non-decreasing order, meaning each value should be greater than or equal to the one before it. After applying these sorting rules, the modified array should be returned.

For example, let's say `nums` is `[6, 3, 5, 2, 8, 1]`. After sorting, the values at odd indices `[3, 2, 1]` should be sorted in non-increasing order, and the values at even indices `[6, 5, 8]` should be sorted in non-decreasing order. The rearranged array will be `[5, 3, 6, 2, 8, 1]`.

Intuition

The solution leverages the fact that we can treat the odd and even indexed elements of the array separately. We can "slice" the original array into two - one containing the even-indexed elements and the other containing the odd-indexed elements. Once we separate the two, we can sort them individually according to the rules: non-decreasing order for even-indexed elements and non-increasing order for odd-indexed elements.

When we sort the even-indexed elements, we extract elements starting at index 0 and then every second element thereafter (using Python's slicing syntax `[::2]`). For the odd-indexed elements, we start at index 1 and again take every second element (using `[1::2]`). Once sorted, we can then interleave these two lists back into `nums` in the original order of odd and even indices to get our final rearranged array.

The implementation of this intuition is straightforward and involves the following steps:

1. Slice out and sort the even-indexed elements.
2. Slice out, sort (in reverse order), and reverse the odd-indexed elements, so they become non-increasing.
3. Merge the sorted even-indexed elements with the sorted odd-indexed ones by interleaving them back into the `nums` array.
4. Return the rearranged `nums` array.

By following this approach, we can arrive at the solution in an efficient and straightforward manner.

Solution Approach

The implementation of the solution uses a simple and efficient approach:

1. **Slicing**: To separate the even and odd indexed elements, Python's slicing feature is used. The slice `nums[::2]` takes every element starting from index `0`, and continuing in steps of `2` (this gives us all even-indexed elements since Python is 0-indexed). Similarly, `nums[1::2]` gives us all odd-indexed elements starting at index `1`.

2. **Sorting**: Python's built-in `sorted()` function is then used to sort these two subsets. For the even-indexed elements, the function call `sorted(nums[::2])` returns a new list where the elements are in ascending (non-decreasing) order. For the odd-indexed elements, the call `sorted(nums[1::2], reverse=True)` returns them sorted in descending (non-increasing) order due to the `reverse=True` parameter. It is important to understand that sorting in reverse is the same as sorting normally and then reversing the list, which is exactly what we want for the non-increasing sorting condition.

3. **Merging back into the original array**: After sorting, these sorted subsets are interleaved back into `nums`. The even-indexed elements replace the original even-indexed elements (`nums[::2] = a`), and the odd-indexed elements replace the original odd-indexed elements (`nums[1::2] = b`). This step overwrites `nums` with the newly sorted values while maintaining the original structure of even and odd indices.

4. **Returning the result**: Finally, the `nums` list, now rearranged according to the specified rules, is returned.

The Python slicing feature is particularly useful in this solution because it allows us to easily extract and manipulate elements based on their indices. The approach is efficient since we're operating directly on the slices of the input list, and Python's built-in sorting function is optimized for performance.

Overall, the data structure used is the input list `nums` itself, which is modified in place to avoid using extra space. The only algorithms used are the slicing and sorting operations provided by Python. This is a great example of a problem that can be solved elegantly with the right choice of built-in functions and language features.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following array `nums`:

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

We need to sort the even-indexed elements (indices 0, 2, 4: values 4, 2, 6) in non-decreasing order, and the odd-indexed elements (indices 1, 3, 5: values 1, 3, 5) in non-increasing order.

According to the solution approach:

1. **Slicing the even-indexed elements**: We extract the even-indexed elements (initially at indices 0, 2, 4), which gives us `[4, 2, 6]`. Then we sort this slice in non-decreasing order using `sorted(nums[::2])`.

   After sorting, we get:

   ```
   even_sorted = [2, 4, 6]
   ```

2. **Slicing the odd-indexed elements**: We do the same for the odd-indexed elements (initially at indices 1, 3, 5), resulting in `[1, 3, 5]`. We sort this slice in non-increasing order with `sorted(nums[1::2], reverse=True)`.

   After sorting and reversing for non-increasing order, we get:

   ```
   odd_sorted = [5, 3, 1]
   ```

3. **Merging back into the original array**: We now interleave these sorted slices back into the array `nums`. The even-indexed elements get replaced with `even_sorted`:

   ```
   nums[::2] = even_sorted   // nums becomes [2, _, 4, _, 6, _]
   ```

   And the odd-indexed elements get replaced with `odd_sorted`:

   ```
   nums[1::2] = odd_sorted   // nums becomes [2, 5, 4, 3, 6, 1]
   ```

4. **Returning the result**: The array `nums` is now properly rearranged:

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

The resulting array has the even-indexed elements sorted in non-decreasing order and the odd-indexed elements sorted in non-increasing order, which matches the problem's requirements. This represents our final solution, the rearranged `nums` array, which can now be returned.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def sortEvenOdd(self, nums: List[int]) -> List[int]:
        # Sort the elements at even indices in non-decreasing order
        even_index_sorted = sorted(nums[::2])

        # Sort the elements at odd indices in non-increasing order (or decreasing order)
        odd_index_sorted = sorted(nums[1::2], reverse=True)

        # Updating the original list, place the sorted even indices back in their original places
        nums[::2] = even_index_sorted

        # Update the original list, place the sorted odd indices back in their positions
        nums[1::2] = odd_index_sorted

        return nums  # Return the newly sorted list according to the rules

# The code above defines a method `sortEvenOdd` within a class `Solution`.
# The method takes a list `nums` as input and returns a new list in which
# the elements at even indices are sorted in non-decreasing order, and the
# elements at odd indices are sorted in non-increasing order.
```

Java

```java
import java.util.Arrays;

class Solution {

    // The sortEvenOdd method takes an array of integers and sorts the indices of the array such that
    // all even indices are sorted in ascending order and all odd indices are sorted in descending order.
    public int[] sortEvenOdd(int[] nums) {
        int n = nums.length; // The length of the provided array.

        // Arrays to separately store elements at even and odd indices.
        int[] evenIndexedElements = new int[n / 2 + n % 2]; // n/2 is equivalent to dividing by 2.
        int[] oddIndexedElements = new int[n >> 1]; // These will be sorted separately.

        // Split the original array elements into two separate arrays.
        for (int i = 0, j = 0; i < n / 2; i += 2, ++j) {
            evenIndexedElements[j] = nums[i];
            oddIndexedElements[j] = nums[i + 1];
        }

        // If the number of elements is odd, the last element belongs to the even index array.
        if (n % 2 == 1) {
            evenIndexedElements[evenIndexedElements.length - 1] = nums[n - 1];
        }

        // Sort the even and odd indexed elements independently.
        Arrays.sort(evenIndexedElements); // Sorts in ascending order.
        Arrays.sort(oddIndexedElements); // To be reversed later.

        // Array to store the final sorted elements.
        int[] sortedArray = new int[n];

        // Merge the even indexed elements back into the final array.
        for (int i = 0, j = 0; j < evenIndexedElements.length; i += 2, ++j) {
            sortedArray[i] = evenIndexedElements[j];
        }

        // Merge and reverse the odd indexed elements into the final array.
        for (int i = 1, j = oddIndexedElements.length - 1; j >= 0; i += 2, --j) {
            sortedArray[i] = oddIndexedElements[j]; // Inserting in descending order.
        }

        return sortedArray; // Return the final sorted array.
    }
}
```

C++

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

class Solution {
public:
    // Function to sort even and odd indexed elements in separate order
    vector<int> sortEvenOdd(vector<int>& nums) {
        // Retrieve the size of the input vector
        int size = nums.size();

        // Vectors to hold even and odd indexed elements
        vector<int> evenIndexedElements;
        vector<int> oddIndexedElements;

        // Iterate over the input vector and distribute elements to even or odd vectors
        for (int i = 0; i < size; ++i) {
            if (i % 2 == 0) {
                // Even indexed elements (0-indexed)
                evenIndexedElements.push_back(nums[i]);
            } else {
                // Odd indexed elements (0-indexed)
                oddIndexedElements.push_back(nums[i]);
            }
        }

        // Sort even indexed elements in ascending order
        sort(evenIndexedElements.begin(), evenIndexedElements.end());
        // Sort odd indexed elements in descending order
        sort(oddIndexedElements.begin(), oddIndexedElements.end(), greater<int>());

        // Vector to store the final sorted numbers
        vector<int> sortedNums(size);

        // Merge even indexed elements back into `sortedNums`
        for (int i = 0, j = 0; j < evenIndexedElements.size(); i += 2, ++j) {
            sortedNums[i] = evenIndexedElements[j];
        }

        // Merge odd indexed elements back into `sortedNums`
        for (int i = 1, j = 0; j < oddIndexedElements.size(); i += 2, ++j) {
            sortedNums[i] = oddIndexedElements[j];
        }

        // Return the final sorted vector
        return sortedNums;
    }
};
```

TypeScript

```typescript
function sortEvenOdd(nums: number[]): number[] {
    // Retrieve the size of the input array
    const size: number = nums.length;

    // Arrays to hold even and odd indexed elements
    let evenIndexedElements: number[] = [];
    let oddIndexedElements: number[] = [];

    // Iterate over the input array and distribute elements to even or odd arrays
    for (let i = 0; i < size; i++) {
        if (i % 2 === 0) {
            // Even indexed elements (0-indexed)
            evenIndexedElements.push(nums[i]);
        } else {
            // Odd indexed elements (0-indexed)
            oddIndexedElements.push(nums[i]);
        }
    }

    // Sort even indexed elements in ascending order
    evenIndexedElements.sort((a, b) => a - b);
    // Sort odd indexed elements in descending order
    oddIndexedElements.sort((a, b) => b - a);

    // Array to store the final sorted numbers
    let sortedNums: number[] = new Array(size);

    // Merge even indexed elements back into `sortedNums`
    for (let i = 0, j = 0; j < evenIndexedElements.length; i += 2, j++) {
        sortedNums[i] = evenIndexedElements[j];
    }

    // Merge odd indexed elements back into `sortedNums`
    for (let i = 1, j = 0; j < oddIndexedElements.length; i += 2, j++) {
        sortedNums[i] = oddIndexedElements[j];
    }

    // Return the final sorted array
    return sortedNums;
}
```

```python
from typing import List

class Solution:
    def sortEvenOdd(self, nums: List[int]) -> List[int]:
        # Sort the elements at even indices in non-decreasing order
        even_index_sorted = sorted(nums[::2])

        # Sort the elements at odd indices in non-increasing order (or decreasing order)
        odd_index_sorted = sorted(nums[1::2], reverse=True)

        # Updating the original list, place the sorted even indices back in their original places
        nums[::2] = even_index_sorted

        # Update the original list, place the sorted odd indices back in their positions
        nums[1::2] = odd_index_sorted

        return nums  # Return the newly sorted list according to the rules

# The code above defines a method `sortEvenOdd` within a class `Solution`.
# The method takes a list `nums` as input and returns a new list in which
# the elements at even indices are sorted in non-decreasing order, and the
# elements at odd indices are sorted in non-increasing order.
```

Time and Space Complexity

The given Python code takes an input list `nums` and sorts the even-indexed elements in ascending order and the odd-indexed elements in descending order.

Time Complexity

The main operations that determine the time complexity are the two `sorted` function calls and the slicing operations.

1. `nums[::2]` and `nums[1::2]` are slicing operations that take $O(n)$ time, where n is the length of the list `nums`. These operations are done twice each, once to create `a` and `b`, and once to update `nums`.
2. `sorted(nums[::2])` sorts the even-indexed elements, which is roughly $n/2$ elements, leading to a time complexity of $O(n/2 * \log(n/2))$. This simplifies to $O(n * \log(n))$ in terms of big-O notation.
3. `sorted(nums[1::2], reverse=True)` sorts the odd-indexed elements, which is also roughly $n/2$ elements, with the same complexity of $O(n * \log(n))$.

Hence, the overall time complexity of the code snippet is $O(n * \log(n))$, as the sorting operations dominate the time complexity.

Space Complexity

The space complexity considerations include the additional space required for storing the sorted sublists `a` and `b`.

1. `a` and `b` each store about $n/2$ elements, so together they require $O(n)$ space.

However, the `sorted` function creates a new list, and hence the space complexity for both `a` and `b` is $O(n)$ combined since each list gets up to $n/2$ elements, thus making the space complexity $O(n)$.

The final assignment operations where `nums[::2] = a` and `nums[1::2] = b` do not use additional space as they are happening in-place in the original list `nums`.

Therefore, the overall space complexity of the function is $O(n)$.