2161. Partition Array According to Given Pivot

Medium Array Two Pointers Simulation

Problem Description

In this problem, we are given an array `nums` and an integer `pivot`. The goal is to rearrange the array such that:

1. All elements less than `pivot` are positioned before all elements greater than `pivot`.
2. All elements equal to `pivot` are placed in the middle, between the elements less than and those greater than `pivot`.
3. The relative order of elements less than and those greater than `pivot` must be the same as in the original array.

In summary, the task is to partition the array into three parts: the first containing elements less than `pivot`, the second containing elements equal to `pivot`, and the third containing elements greater than `pivot`, while preserving the original relative order of the elements in the first and third parts.

Intuition

To solve this problem, we can approach it by separating the elements into three distinct lists based on their comparison to the `pivot`:

1. A list to hold elements less than `pivot`.
2. A list for elements equal to `pivot`.
3. A list for elements greater than `pivot`.

After segregating the elements into the respective lists, we can then concatenate these lists in the order of less than `pivot`, equal to `pivot`, and greater than `pivot`. This concatenation will result in the desired array that fulfills all the problem conditions.

The reason we use separate lists instead of in-place swaps is that in-place operations might make it complex to preserve the original relative order. Simple list operations like appending and concatenation keep the original order intact and make the implementation straightforward and efficient.

This approach ensures that we only pass through the array once, making the algorithm linear in time because each element is considered exactly once and placed into one of the three lists.

Solution Approach

The solution is implemented in Python and uses a simple and effective algorithm involving basic list operations. Here's the walk-through of the implementation:

1. Three separate lists are initialized: `a` to hold elements less than `pivot`, `b` for elements equal to `pivot`, and `c` for elements greater than `pivot`.

```
a, b, c = [], [], []
```

2. The algorithm proceeds by iterating through each element `x` in the given array `nums`.

```
for x in nums:
```

3. For each element `x`, a comparison is made to classify it into one of the three lists:
   - If `x` is less than `pivot`, it is appended to the list `a`.
   - If `x` is equal to `pivot`, it is appended to list `b`.
   - If `x` is greater than `pivot`, it is appended to list `c`.

```
if x < pivot:
    a.append(x)
elif x == pivot:
    b.append(x)
else:
    c.append(x)
```

4. By the end of the loop, all the elements are distributed among the three lists, preserving their original relative order within each category (less than, equal to, and greater than `pivot`).
5. The final step is to concatenate the three lists: `a` (elements less than `pivot`), `b` (elements equal to `pivot`), and `c` (elements greater than `pivot`). This results in the rearranged array that meets all the required conditions.

```
return a + b + c
```

Through the use of lists and the built-in list method `append`, the solution takes advantage of Python's dynamic array capabilities. This eliminates the need for complex index management or in-place replacements that might compromise the relative order of the elements.

The solution relies on the efficiency of Python's underlying memory management for dynamic arrays, and it works within the confines of O(n) space complexity (where `n` is the number of elements in `nums`) because it creates separate lists for partitioning the data, which are later merged. The time complexity is also O(n), as each element is looked at exactly once during the for-loop iteration.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Suppose we have the following array and pivot:

```
nums = [9, 12, 3, 5, 14, 10, 10]
pivot = 10
```

Now, we will apply the solution algorithm step by step:

1. Initialize three empty lists `a`, `b`, and `c` to categorize the elements as less than, equal to, and greater than the pivot, respectively:

```
a = []
b = []
c = []
```

2. Iterate through each element `x` in the array `nums`:
   - We start with `x = 9` which is less than the pivot, so we append it to the list `a`: `a = [9]`.
   - Next, `x = 12` is greater than the pivot, appended to `c`: `c = [12]`.
   - Then, `x = 3` is less than the pivot, appended to `a`: `a = [9, 3]`.
   - Followed by `x = 5` which is again less than the pivot, so `a` becomes `a = [9, 3, 5]`.
   - We proceed to `x = 14` which is greater than the pivot and append it to `c`: `c = [12, 14]`.
   - Next, we have two elements equal to the pivot, `x = 10`, so we append both to `b`: `b = [10, 10]`.

   At the end of the iteration, the lists are as follows:

```
a = [9, 3, 5]
b = [10, 10]
c = [12, 14]
```

3. All the elements have been classified into three separate lists while preserving their original order within each list category.
4. We concatenate the three lists in the order of `a`, `b`, and `c` to get the final result:

```
result = a + b + c
# result = [9, 3, 5, 10, 10, 12, 14]
```

The final array is `[9, 3, 5, 10, 10, 12, 14]` which satisfies the condition of keeping all elements less than 10 before all elements equal to 10 and those greater than 10, while preserving the original relative order within each category.

By using this approach, we've maintained a simple, understandable, and efficient solution that neatly classifies and recombines the elements as desired.

Solution Implementation

Python

```
class Solution:
    def pivotArray(self, nums: List[int], pivot: int) -> List[int]:
        # Initialize lists to hold numbers smaller than,
        # equal to and greater than the pivot
        smaller_than_pivot = []
        equal_to_pivot = []
        greater_than_pivot = []

        # Iterate through each number in the input list
        for number in nums:
            # If the number is less than the pivot,
            # add it to the smaller_than_pivot list
            if number < pivot:
                smaller_than_pivot.append(number)
            # If the number is equal to the pivot,
            # add it to the equal_to_pivot list
            elif number == pivot:
                equal_to_pivot.append(number)
            # If the number is greater than the pivot,
            # add it to the greater_than_pivot list
            else:
                greater_than_pivot.append(number)

        # Combine the lists and return the result.
        # with all numbers less than the pivot first,
        # followed by numbers equal to the pivot,
        # and finally numbers greater than the pivot
        return smaller_than_pivot + equal_to_pivot + greater_than_pivot
```

Java

```
class Solution {
    // This method takes an array 'nums' and an integer 'pivot', then reorders the array such that
    // all elements less than 'pivot' come before elements equal to 'pivot', and those come before elements greater than 'pivot'.
    public int[] pivotArray(int[] nums, int pivot) {
        int n = nums.length; // Get the length of the array.
        int[] ans = new int[n]; // Create a new array 'ans' to store the reordered elements.
        int index = 0; // Initialize an index variable to keep track of the position in 'ans' array.

        // First pass: Place all elements less than 'pivot' into the 'ans' array.
        for (int num : nums) {
            if (num < pivot) {
                ans[index++] = num;
            }
        }

        // Second pass: Place all elements equal to 'pivot' into the 'ans' array.
        for (int num : nums) {
            if (num == pivot) {
                ans[index++] = num;
            }
        }

        // Third pass: Place all elements greater than 'pivot' into the 'ans' array.
        for (int num : nums) {
            if (num > pivot) {
                ans[index++] = num;
            }
        }

        return ans; // Return the reordered array.
    }
}
```

C++

```
#include <vector>

class Solution {
public:
    // Function to rearrange elements in an array with respect to a pivot element.
    // All elements less than pivot come first, followed by elements equal to pivot,
    // and then elements greater than pivot.
    std::vector<int> pivotArray(std::vector<int>& nums, int pivot) {
        // Vector to store the rearranged elements.
        std::vector<int> rearranged;

        // First pass: add elements less than pivot to rearranged vector.
        for (int num : nums) {
            if (num < pivot) {
                rearranged.push_back(num);
            }
        }

        // Second pass: add elements equal to pivot to rearranged vector.
        for (int num : nums) {
            if (num == pivot) {
                rearranged.push_back(num);
            }
        }

        // Third pass: add elements greater than pivot to rearranged vector.
        for (int num : nums) {
            if (num > pivot) {
                rearranged.push_back(num);
            }
        }

        // Return the vector containing elements in the desired order.
        return rearranged;
    }
};
```

TypeScript

```
// Import the array class from TypeScript default library
import { number } from "yoga-layout";

// Function to rearrange elements in an array with respect to a pivot element.
// All elements less than pivot come first, followed by elements equal to pivot,
// and then elements greater than pivot.
function pivotArray(nums: number[], pivot: number): number[] {
    // Array to store the rearranged elements.
    let rearranged: number[] = [];

    // First pass: add elements less than pivot to rearranged array.
    for (let num of nums) {
        if (num < pivot) {
            rearranged.push(num);
        }
    }

    // Second pass: add elements equal to pivot to rearranged array.
    for (let num of nums) {
        if (num === pivot) {
            rearranged.push(num);
        }
    }

    // Third pass: add elements greater than pivot to rearranged array.
    for (let num of nums) {
        if (num > pivot) {
            rearranged.push(num);
        }
    }

    // Return the array containing elements in the desired order.
    return rearranged;
}
```

```
class Solution:
    def pivotArray(self, nums: List[int], pivot: int) -> List[int]:
        # Initialize lists to hold numbers smaller than,
        # equal to and greater than the pivot
        smaller_than_pivot = []
        equal_to_pivot = []
        greater_than_pivot = []

        # Iterate through each number in the input list
        for number in nums:
            # If the number is less than the pivot,
            # add it to the smaller_than_pivot list
            if number < pivot:
                smaller_than_pivot.append(number)
            # If the number is equal to the pivot,
            # add it to the equal_to_pivot list
            elif number == pivot:
                equal_to_pivot.append(number)
            # If the number is greater than the pivot,
            # add it to the greater_than_pivot list
            else:
                greater_than_pivot.append(number)

        # Combine the lists and return the result.
        # with all numbers less than the pivot first,
        # followed by numbers equal to the pivot,
        # and finally numbers greater than the pivot
        return smaller_than_pivot + equal_to_pivot + greater_than_pivot
```

Time and Space Complexity

Time Complexity:

The time complexity of the given code relies primarily on the for loop that iterates over all `n` elements in the input list `nums`. Inside the loop, the code compares each element with the `pivot` and then adds it to one of the three lists (`a`, `b`, or `c`). Each of these operations—comparison and append—is performed in constant time, `O(1)`. However, combining the lists at the end `a + b + c` takes `O(n)` time since it creates a new list containing all `n` elements. Therefore, the overall time complexity of the code is `O(n)`.

Space Complexity:

The space complexity refers to the amount of extra space or temporary space used by the algorithm. In this case, we're creating three separate lists (`a`, `b`, and `c`) to hold elements less than, equal to, and greater than the pivot, respectively. In the worst-case scenario, all elements could be less than, equal to, or greater than the pivot, leading to each list potentially containing `n` elements. Therefore, the additional space used by the lists is directly proportional to the number of elements in the input, `n`. Thus, the space complexity is `O(n)`.