3069. Distribute Elements Into Two Arrays I

Easy Array Simulation

Problem Description

You have an array of distinct integers, numbered from 1 and following the sequence of natural numbers. Your goal is to separate all the elements of this array into two new arrays while following a specific set of rules during distribution.

In the beginning, you'll place the first element of the given array into the first new array (let's call it `arr1`), and the second element into the second new array (`arr2`). For each subsequent element, you'll make a decision based on the comparison between the last elements of `arr1` and `arr2`. If the last element of `arr1` is greater than the last element of `arr2`, you'll add the next element into `arr1`. Otherwise, you put it into `arr2`.

After you've gone through all the elements, you'll combine both `arr1` and `arr2` into a single array called `result`, where `arr1` is followed by `arr2`. The task is to return this concatenated `result` array. Notably, since integers are distinct and the distribution is influenced by the comparison between the latest numbers in each array, the sequence of elements in the final array will reflect the sequence of decision-making during the distribution process.

Intuition

The fundamental insight for approaching this problem is that the distribution process is sequential and decision-based, depending entirely on the comparison between the trailing elements of `arr1` and `arr2`. This suggests a straightforward, step-by-step simulation where you iterate through the list and distribute elements based on the stated rules.

By starting with the initial conditions (first element in `arr1` and second in `arr2`), you merely need to keep checking the last elements of both arrays to decide the placement of the next item. Since the elements are distinct, there will be no ties, and the decision will always be clear-cut.

Solution Approach

The implementation of the solution is a direct simulation of the process described in the problem definition. We translate the distinct steps of decision-making into code structure. The Python programming language offers a simple and expressive way to achieve this in just a few lines.

Initially, we create two arrays (`arr1` and `arr2`) to represent the two groups where elements of the input array (`nums`) will be distributed. The first element (`nums[0]`) is put in `arr1`, and the second (`nums[1]`) goes into `arr2`. This sets up the initial state of the arrays as per the problem's rules.

We then go into a loop starting at the third element (index `2`, since arrays are 1-indexed in the description) of `nums`. For each element `x` in the array from index `2` onwards, we must decide whether `x` will be added to `arr1` or `arr2`. The decision is based on a simple comparison:

- If the last element of `arr1` (retrieved with `arr1[-1]`) is greater than the last element of `arr2` (`arr2[-1]`), `x` is appended to `arr1`.
- Otherwise, `x` goes into `arr2`.

This process relies on two key aspects:

- **Array indexing and manipulation**: By using negative indexes (like -1), we can easily access the last elements of the arrays in Python, which is a convenient feature of the language's data structure functionality.
- **Conditional logic**: The decision-making process used for distribution is implemented through a simple `if`-`else` statement, a fundamental control structure in programming that allows us to execute different actions based on certain conditions.

After all elements have been placed into `arr1` or `arr2`, the two arrays are concatenated using the `+` operator, which in Python can merge lists. This last step creates the final `result` array as specified.

The overall design pattern follows a linear, iterative approach, meaning the elements of `nums` are processed in order. No complex data structures, backtracking, or optimization techniques are needed because the problem statement guarantees that there will be no ambiguity in the decision-making process; thus, each step depends only on the state from the previous step.

Example Walkthrough

Let's consider an example with an array of distinct integers `nums = [1, 2, 3, 4, 5]`. We will walk through the application of the solution approach to this array.

1. We start by creating two new arrays, `arr1` and `arr2`.
2. According to the rules, we must place the first element of `nums` in `arr1`, and the second in `arr2`. Thus, after this step, `arr1 = [1]` and `arr2 = [2]`.
3. Next, we begin iterating from the third element of `nums`. The third element is `3`.
   - We compare the last elements of both arrays: `arr1[-1] = 1` and `arr2[-1] = 2`. Since `1` is not greater than `2`, we add `3` to `arr2`.
   - Now, `arr1 = [1]` and `arr2 = [2, 3]`.
4. Move to the fourth element, which is `4`.
   - We compare again: `arr1[-1] = 1` and `arr2[-1] = 3`. `1` is less than `3`, so we add `4` to `arr2`.
   - `arr1 = [1]` and `arr2 = [2, 3, 4]`.
5. Finally, we take the fifth element, `5`.
   - Comparing `arr1[-1] = 1` with `arr2[-1] = 4`, `1` is less than `4`, so we add `5` to `arr2`.
   - `arr1 = [1]` and `arr2 = [2, 3, 4, 5]`.
6. After the iteration is done, we concatenate `arr1` and `arr2` to form the `result` array.
   - `result = arr1 + arr2 = [1] + [2, 3, 4, 5] = [1, 2, 3, 4, 5]`.

Thus, the `result` array is `[1, 2, 3, 4, 5]`, which reflects the distribution decision made at each step. However, in this particular example, after placing the first two initial elements, all elements went into the second array due to the initial placement and specific order of `nums`. Different input arrays might yield a more varied distribution between `arr1` and `arr2`.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def result_array(self, nums: List[int]) -> List[int]:
        # Initialize the first array with the first element of nums
        first_array = [nums[0]]

        # Initialize the second array with the second element of nums
        second_array = [nums[1]]

        # Process the remaining elements, starting from the third
        for number in nums[2:]:
            # Compare the last elements of first_array and second_array
            # Place the current number in the array with the smaller last element
            if first_array[-1] > second_array[-1]:
                first_array.append(number)
            else:
                second_array.append(number)

        # Combine the two arrays and return the result
        return first_array + second_array

# Example usage:
# solution = Solution()
# result = solution.result_array([1, 2, 3, 4, 5])
# print(result) # This will print: [1, 2, 3, 4, 5]
```

Java

```java
class Solution {
    // method to construct a result array based on certain rules
    public int[] resultArray(int[] nums) {
        int n = nums.length; // length of the input array
        int[] sortedHalf1 = new int[n]; // first half of the sorted array
        int[] sortedHalf2 = new int[n]; // second half of the sorted array

        // Initialize the first element of each sorted half with the first two numbers
        sortedHalf1[0] = nums[0];
        sortedHalf2[0] = nums[1];

        // Index to track the last filled positions in sortedHalf1 and sortedHalf2
        int i = 0, j = 0;

        // Iterate over the rest of nums to separate into two sorted halves
        for (int k = 2; k < n; ++k) {
            // If the current last number in sortedHalf1 is greater than that of sortedHalf2
            if (sortedHalf1[i] > sortedHalf2[j]) {
                // Place the current number in the next spot in sortedHalf1
                sortedHalf1[++i] = nums[k];
            } else {
                // Otherwise, place it in the next spot in sortedHalf2
                sortedHalf2[++j] = nums[k];
            }
        }

        // Append the contents of sortedHalf2 to the end of sortedHalf1
        for (int k = 0; k <= j; ++k) {
            sortedHalf1[++i] = sortedHalf2[k];
        }

        // Return the combined array
        return sortedHalf1;
    }
}
```

C++

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to create a result array from the input array 'nums'
    vector<int> resultArray(vector<int>& nums) {
        int n = nums.size();  // Size of the input array

        // Initial array containing the first element
        vector<int> array1 = {nums[0]};

        // Check for edge cases where nums might contain less than 2 elements
        if (n < 2) {
            return array1;  // If less than 2, return array1 as the result
        }

        // Second array containing the second element
        vector<int> array2 = {nums[1]};

        // Loop through the rest of elements starting from the third element
        for (int k = 2; k < n; ++k) {
            // Decide which array to append the current element (nums[k]) based on the last elements of array1 and array2
            if (array1.back() > array2.back()) {
                array1.push_back(nums[k]);
            } else {
                array2.push_back(nums[k]);
            }
        }

        // Merge array2 into array1, thus array1 will contain elements of both arrays
        array1.insert(array1.end(), array2.begin(), array2.end());

        return array1;  // Return the merged array as the result
    }
};
```

TypeScript

```typescript
/**
 * Takes an array of numbers and returns a new array.
 * The new array is a concatenation of two subarrays:
 * - The first subarray contains elements from the original array at even indices.
 * - The second subarray contains elements from the original array at odd indices.
 * Each subarray builds from the second element by comparing the last elements
 * and pushing the next value to the subarray with the smaller last value.
 * @param {number[]} nums - The input array of numbers.
 * @returns {number[]} The resulting concatenated array from two subarrays.
 */
function resultArray(nums: number[]): number[] {
    // Initialize first subarray with the first element of the input array
    const subArrayEven: number[] = [nums[0]];

    // Initialize second subarray with the second element of the input array
    const subArrayOdd: number[] = [nums[1]];

    // Iterate over the rest of the elements starting from the third element
    for (const num of nums.slice(2)) {
        // Compare the last elements of both subarrays and push the current
        // number (num) to the subarray with the smaller last item
        if (subArrayEven[subArrayEven.length - 1] > subArrayOdd[subArrayOdd.length - 1]) {
            // If the last item of subArrayEven is greater, push to subArrayOdd
            subArrayOdd.push(num);
        } else {
            // Otherwise, push to subArrayEven
            subArrayEven.push(num);
        }
    }

    // Concatenate the two subarrays and return the result
    return subArrayEven.concat(subArrayOdd);
}
```

```python
from typing import List

class Solution:
    def result_array(self, nums: List[int]) -> List[int]:
        # Initialize the first array with the first element of nums
        first_array = [nums[0]]

        # Initialize the second array with the second element of nums
        second_array = [nums[1]]

        # Process the remaining elements, starting from the third
        for number in nums[2:]:
            # Compare the last elements of first_array and second_array
            # Place the current number in the array with the smaller last element
            if first_array[-1] > second_array[-1]:
                first_array.append(number)
            else:
                second_array.append(number)

        # Combine the two arrays and return the result
        return first_array + second_array

# Example usage:
# solution = Solution()
# result = solution.result_array([1, 2, 3, 4, 5])
# print(result) # This will print: [1, 2, 3, 4, 5]
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`. This is because there is a single loop that iterates through the `nums` array starting from the third element, which runs in `O(n-2)` time. However, since constant factors are disregarded in Big O notation, it simplifies to `O(n)`.

The space complexity is also `O(n)`. Two new arrays, `arr1` and `arr2`, are created and potentially, all elements of `nums` could be added to these arrays. In the worst case, both `arr1` and `arr2` together would contain all elements of the input `nums`, making the space complexity linear with respect to the size of the input.