1470. Shuffle the Array

Easy Array

Leetcode Link

Problem Description

The problem provides an array `nums` that has `2n` elements in it. The elements are structured in a way that the first `n` elements are grouped together (let's call them `x` group) and the next `n` elements are grouped together (let's call them `y` group). The objective is to rearrange this array by combining the elements from `x` group and `y` group alternatively, starting with the first element of the `x` group. In the end, the array should look like `[x1, y1, x2, y2, ..., xn, yn]`, which means we pick one element from the `x` group and then pick the corresponding element from the `y` group and continue this pattern until the array is fully rearranged.

Intuition

To arrive at the solution, we can realize that we need to build a new array by taking one element from the `x` group (first `n` elements of `nums`) and then taking the corresponding element from the `y` group (last `n` elements of `nums`). The approach is quite straightforward:

1. Initialize an empty list `ans` where we will store our answer.
2. Iterate over the range from 0 to `n-1` (since we have `2n` elements, and `n` denotes the number of pairs we need to create).
3. In each iteration, append the element from the `x` group (`nums[i]`) to `ans`.
4. Then, append the corresponding element from the `y` group (`nums[i + n]`) to `ans`.
5. After the loop ends, `ans` will have `2n` elements in the desired order `[x1, y1, x2, y2, ..., xn, yn]`.
6. Return the `ans` array which is now correctly shuffled.

This solution is easy to understand and implement. It uses extra space for the new array and takes linear time relative to the size of the original array, which is an efficient way to achieve the task.

Solution Approach

The implemented solution makes use of a simple algorithm and a basic array data structure to achieve the desired result. The algorithm can be considered as a single-pass approach, which can be broken down into these main steps:

1. **Initialization**: An empty list called `ans` is initialized. This is where the shuffled array will be stored.
2. **Single-Pass Loop**:
   - A `for` loop is initiated to iterate `n` times, where `n` is half the size of the input array `nums`. This is because of the input array's structure, housing two groups of `n` elements each (`x` group and `y` group).
   - Inside the loop, a simple pattern is followed:
     - The element `nums[i]` from the `x` group (the first half of the array) is appended to the `ans` list.
     - Following that, the element `nums[i + n]` from the `y` group (the second half of the array) is appended to the `ans` list.
     - This pattern is based on an observation that for any index `i` in the first half of the array, its corresponding pair from the second half can be accessed at index `i + n`.
3. **Building the Result**: The steps inside the loop ensures that `ans` grows by two elements after each iteration, maintaining the desired alternate sequence of elements from both `x` and `y` groups.
4. **Returning the Result**: Once the loop finishes, the list `ans` now contains all elements required for the output array in the correct order. It is then returned as the final shuffled array.

In terms of data structures, this approach only relies on the list data structure, which is intrinsic to Python. No additional or specialized data structures are used. The algorithm's simplicity and absence of nested loops or recursive calls mean that the time complexity is linear, `O(n)`, where `n` is the number of elements in half of the input list. The space complexity is also `O(n)` to account for the `ans` list which stores the reshuffled elements.

This approach effectively demonstrates how algorithmic patterns can emerge from the inherent structure of a problem, enabling a straightforward implementation that efficiently yields the desired result.

Example Walkthrough

Let's take an example to illustrate the solution approach where the `nums` array is `[1, 3, 2, 4]`, which represents `2n` elements, with `n` being 2. We have the first `n` elements forming group `x` (which are `[1, 3]`) and the second `n` elements forming group `y` (`[2, 4]`).

Following the steps from the solution approach:

1. **Initialization**: We initialize an empty list `ans` where the shuffled array will be stored.
2. **Single-Pass Loop**:
   - We start with a `for` loop that iterates from `0` to `n-1`, which in this case is from `0` to `1`.
   - In the first iteration (i=0):
     - We append `nums[0]` (which is `1` from the `x` group) to `ans`, making `ans` now `[1]`.
     - We then append `nums[0 + n]` (which is `nums[2]`, or `2` from the `y` group) to `ans`. Now `ans` is `[1, 2]`.
   - In the second iteration (i=1):
     - We append `nums[1]` (which is `3` from the `x` group) to `ans`, so `ans` becomes `[1, 2, 3]`.
     - Next, we append `nums[1 + n]` (which is `nums[3]`, or `4` from the `y` group) to `ans`. The `ans` list is now `[1, 2, 3, 4]`.
3. **Building the Result**: At the end of the loop, `ans` has grown to include all `2n` elements in the desired alternate `x, y` pattern, which is `[1, 2, 3, 4]` for our example.
4. **Returning the Result**: We return the `ans` list, which is `[1, 2, 3, 4]`. This is our final shuffled array, where the `x` group and the `y` group elements are alternatively arranged.

Through this example, we can observe the simplicity and effectiveness of the approach, which leads us to the correct shuffled array by performing the operation step by step.

Python Solution

```python
from typing import List

class Solution:
    def shuffle(self, nums: List[int], n: int) -> List[int]:
        # Initialize an empty list for the shuffled result
        shuffled_list = []

        # Iterate over the range from 0 to n
        for i in range(n):
            # Append the element from the first half of the list
            shuffled_list.append(nums[i])

            # Append the corresponding element from the second half of the list
            shuffled_list.append(nums[i + n])

        # Return the shuffled list
        return shuffled_list

```

Java Solution

```java
class Solution {
    public int[] shuffle(int[] nums, int n) {
        // Initialize a new array 'result' with size twice of 'n'.
        int[] result = new int[2 * n];

        // Use a single loop to iterate through the first half of the 'nums' array.
        for (int index = 0, resultIndex = 0; index < n; ++index) {
            // Place element from the first half of 'nums' into the 'result' array.
            result[resultIndex++] = nums[index];
            // Place the corresponding element from the second half of 'nums'.
            result[resultIndex++] = nums[index + n];
        }

        // Return the shuffled 'result' array.
        return result;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Shuffle the `nums` vector in a specific pattern and return the result.
    vector<int> shuffle(vector<int>& nums, int n) {
        // Initialize an empty vector to hold the shuffled result.
        vector<int> shuffledResult;

        // Loop over the elements in the first half of `nums`.
        for (int i = 0; i < n; ++i) {
            // Add the element from the first half of `nums` to the result.
            shuffledResult.push_back(nums[i]);
            // Add the corresponding element from the second half of `nums` to the result.
            shuffledResult.push_back(nums[i + n]);
        }

        // Return the shuffled vector.
        return shuffledResult;
    }
};

```

Typescript Solution

```typescript
/**
 * Shuffles an array consisting of 2n elements in the form of [x1,x2,...,xn,y1,y2,...,yn]
 * into the form [x1,y1,x2,y2,...,xn,yn].
 *
 * @param {number[]} nums - The array to shuffle with 2n elements.
 * @param {number} n - Half the length of the array, representing the split point.
 * @returns {number[]} - The shuffled array.
 */
function shuffle(nums: number[], n: number): number[] {
    // Initialize an empty array to store the shuffled elements
    let shuffledArray: number[] = [];

    // Loop through the first half of the array
    for (let i = 0; i < n; i++) {
        // Push the element from the first half
        shuffledArray.push(nums[i]);
        // Push the corresponding element from the second half
        shuffledArray.push(nums[n + i]);
    }

    // Return the shuffled array
    return shuffledArray;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code can be analyzed based on the for loop that iterates over `n` elements. Inside the loop, the code performs a constant amount of work (two append operations). Therefore, the time complexity is directly proportional to `n`.

The time complexity is `O(n)`.

Space Complexity

The space complexity consists of the additional space required by the solution in addition to the input data. Since it creates a new list `ans` that eventually holds `2 * n` elements, it uses additional space proportional to the size of the input.

Hence, the space complexity is `O(n)`.