Documentation

Problem Understanding:

The task is to design an algorithm that performs two main operations on an integer array: **shuffle** and **reset**. The **reset** operation should return the array to its original configuration, and the **shuffle** operation should return a random array permutation. The algorithm needs to ensure that each possible permutation is equally likely, meaning no permutation is favored over another. The goal is to implement a class that supports these two operations efficiently.

Intuition:

The primary challenge in this problem is implementing a random shuffle in such a way that all permutations of the array are equally probable. The **Fisher-Yates shuffle** algorithm is the perfect solution for this. It works by iterating through the array and swapping each element with another randomly selected element before it. This ensures that every permutation of the array has an equal chance of being selected. Additionally, we need a mechanism to reset the array to its original configuration after shuffling. This can be done by simply storing the initial array and returning it during the reset operation.

Approach:

To solve the problem, we begin by storing a copy of the original array when the Solution object is initialized. This allows us to use the original array for resetting the configuration. The array used for shuffling is maintained separately. For the **reset** operation, we simply return the original array. For the **shuffle** operation, we implement the **Fisher-Yates shuffle** by iterating over the array from the last element to the second element and swapping each element with a randomly selected element from the elements before it, including itself. This method guarantees that all permutations of the array are equally likely.

Reset Operation:

The reset operation is designed to return the array to its original configuration. This is done by referencing the stored self.original array, which holds the array as it was when the Solution object was initialized. This approach is efficient since it doesn't require recalculating the original array but rather directly returns it. It operates in constant time, O(1), as it simply returns a reference to the original array without any further computation.

Shuffle Operation:

The shuffle operation is implemented using the **Fisher-Yates shuffle** algorithm. In this approach, starting from the last element of the array, each element is swapped with another element chosen randomly from the subarray that comes before it. The choice of a random index is uniformly distributed, which ensures that each permutation of the array has an equal probability of being returned. The time complexity of this operation is O(n), where n is the length of the array because we iterate over the array once and perform a constant-time swap for each element.

Time and Space Complexity:

The **reset** operation has a time complexity of O(1) because it simply returns the stored original array. The **shuffle** operation has a time complexity of O(n), where n is the length of the array, due to the single pass through the array and the constant-time operations involved in the swapping of elements. The space complexity of the algorithm is O(n), as we need to store both the original array and the array to be shuffled. This is a modest space requirement, given that we only store two arrays of size n.

Conclusion:

This solution efficiently solves the problem of shuffling an array and resetting it to its original configuration. By using the **Fisher-Yates shuffle** for random shuffling and storing the original array for the reset operation, we ensure that the solution is both time-efficient and space-efficient. The algorithm is well-suited for handling arrays of reasonable sizes, as required by the problem constraints. With O(n) time complexity for both operations and O(n) space complexity, this solution strikes a good balance between efficiency and simplicity.