384. Shuffle an Array **Math** Randomized Medium Array

Problem Description

functionalities: resetting the array to its original order and returning a randomly shuffled version of the array. The class should have the following methods implemented: <u>__init__(self, nums: List[int])</u>: This method initializes the object with the integer array nums. It stores both the original array and a copy that can be modified.

The given LeetCode problem asks us to design a class called Solution that can take an array of integers and provide two

- reset(self) -> List[int]: This method resets the modified array back to its original configuration and returns it. Any subsequent calls to shuffle should not be affected by previous shuffles.
- shuffle(self) -> List[int]: This function returns a new array that is a random shuffle of the original array. It is important that every permutation of the array is equally likely to ensure fairness.
- ntuition

The intuition behind the provided solution is derived from the well-known Fisher-Yates shuffle algorithm, also known as the Knuth

shuffle. The Fisher-Yates shuffle is an algorithm for generating a random permutation of a finite sequence—in this case, our

integer array. The algorithm produces an unbiased permutation: every permutation is equally likely. The process of the shuffle method works as follows: We iterate through the array from the beginning to the end. For each element at index i, we generate a random index j such that i <= j < len(nums).

This swapping ensures all possible permutations of the array are equally likely.

We then swap the elements at indices i and j.

This solution ensures that the shuffling is done in-place, meaning no additional memory is used for the shuffled array except for the input array.

 \circ A loop is initiated, starting from the first index (i = 0) up to the length of the array.

Solution Approach

The algorithm uses the following steps to implement the Solution class and its methods, based on the Fisher-Yates shuffle

algorithm:

Class Initialization (__init__): The constructor takes an array nums and stores it in self.nums.

• The reset method is straightforward; it creates a copy of the self.original array to revert self.nums to the original configuration. • This copy is returned to provide the current state of the array after reset, allowing users to perform shuffling again without any prior shuffle

affecting the outcome.

Reset Method (reset):

• The shuffle method is where the Fisher-Yates algorithm is applied to generate an unbiased random permutation of the array.

It then creates a copy of this array in self.original to preserve the original order for the reset method later.

- **Shuffle Method (shuffle):**
 - \circ Inside the loop, a random index j is chosen where the condition $i \ll j < len(nums)$ holds true. This is done using random randrange(i, len(self.nums)) to pick a random index in the remaining part of the array. • The elements at indices i and j are swapped. Python's tuple unpacking feature is a clean way to do this in one line: self.nums[i],

The Fisher-Yates shuffle ensures that every element has an equal chance of being at any position in the final shuffled array,

leading to each permutation of the array elements being equally likely. This implementation uses 0(n) time where n is the

number of elements in the array and O(n) space because it maintains a copy of the original array to support the reset method.

This process is repeated for each element until the end of the array is reached, resulting in a randomly shuffled array.

Example Walkthrough

self.nums[j] = self.nums[j], self.nums[i].

Suppose we have an array nums = [1, 2, 3].

Class Initialization (__init__):

Shuffle Method (shuffle):

array stays [3, 2, 1].

Let's walk through an example to illustrate how the Solution class and its methods work according to the Fisher-Yates shuffle algorithm:

 Upon initialization, self.nums will store [1, 2, 3], and self.original will also store [1, 2, 3]. Reset Method (reset):

Calling reset() anytime would return [1, 2, 3] since it simply copies the contents of self.original back into self.nums.

Next, we increment i to 1 and choose a new j such that 1 <= j < 3. Assume j remains 1 this time, so no swapping is needed, and the

- \circ Let's say we now call shuffle(). We start with i = 0 and choose a random index j such that $0 \ll j \ll 3$ (it could be 0, 1, or 2). Assume
 - ∘ Finally, for i = 2, we choose j such that 2 <= j < 3, which means j can only be 2. No swapping occurs since i equals j, and the shuffled array remains [3, 2, 1].

j turns out to be 2, so we swap nums [0] with nums [2]. Now the array is [3, 2, 1].

In practical implementations, shuffle() would likely produce different results each time, as j would be determined by a random number generator. Imagine calling shuffle() several times; you might see output like [2, 3, 1], [1, 3, 2], or any other

permutations of [1, 2, 3].

Solution Implementation

def init (self, nums: List[int]):

self.nums = self.original.copy()

Reset the nums list to the original configuration

Swap the current element with the randomly chosen one

self.nums[i], self.nums[j] = self.nums[j], self.nums[i]

* The following lines are typically provided in the problem statement on LeetCode.

std::vector<int> nums; // Vector to store the current state of the array.

std::vector<int> original; // Vector to store the original state of the array.

* They indicate how the Solution class can be used once implemented:

// Constructor to initialize the vectors with the input array.

std::copy(nums.begin(), nums.end(), original.begin());

// Resets the array to its original configuration and returns it.

std::copy(original.begin(), original.end(), nums.begin());

// Generate a random index i such that i <= j < n</pre>

int j = i + std::rand() % (nums.size() - i);

* Solution obj = new Solution(nums);

#include <cstdlib> // For std::rand

Solution(std::vector<int>& nums) {

this->nums = nums;

std::vector<int> reset() {

std::vector<int> shuffle() {

return nums;

return nums;

// Example of how to use the class

#include <algorithm> // For std::copy and std::swap

this->original.resize(nums.size());

// Returns a random shuffling of the array.

// Swap nums[i] with nums[i]

std::swap(nums[i], nums[j]);

// Example of how these functions might be used:

console.log(resetNums); // Output: [1, 2, 3, 4, 5]

console.log(shuffledNums); // Output: [3, 1, 4, 5, 2] (example output, actual output will vary)

Make a copy of the original list to keep it intact for reset purposes

Shuffle the list of numbers in-place using the Fisher-Yates algorithm

Swap the current element with the randomly chosen one

self.nums[i], self.nums[j] = self.nums[j], self.nums[i]

Pick a random index from i (inclusive) to the end of the list (exclusive)

// Reset the array to its original state

def init (self, nums: List[int]):

self.original = nums.copy()

self.nums = self.original.copy()

for i in range(len(self.nums)):

def reset(self) -> List[int]:

Return the reset list

def shuffle(self) -> List[int]:

Time and Space Complexity

method:

Store the original list of numbers

Reset the nums list to the original configuration

i = random.randrange(i, len(self.nums))

// Initialize the array

let resetNums = reset():

from typing import List

let shuffledNums = shuffle();

self.nums = nums

return self.nums

// Shuffle the array

import random

class Solution:

initNums([1, 2, 3, 4, 5]);

for (int i = 0; i < nums.size(); ++i) {</pre>

* int[] param 1 = obj.reset();

* int[] param_2 = obj.shuffle();

def reset(self) -> List[int]:

Return the reset list

def shuffle(self) -> List[int]:

Return the shuffled list

Example of how this class could be used:

from typing import List

irrespective of how many times or how the array has been shuffled previously.

It's important to note that after shuffling, if we call reset(), we will always get the original nums array [1, 2, 3] back,

Store the original list of numbers self.nums = nums # Make a copy of the original list to keep it intact for reset purposes self.original = nums.copy()

Shuffle the list of numbers in-place using the Fisher-Yates algorithm for i in range(len(self.nums)): # Pick a random index from i (inclusive) to the end of the list (exclusive) j = random.randrange(i, len(self.nums))

return self.nums

obj = Solution(nums)

param 1 = obi_reset()

param_2 = obj.shuffle()

return self.nums

Java

Python

import random

class Solution:

```
import java.util.Random:
import java.util.Arrays;
class Solution {
   private int[] nums;  // Array to store the current state (which can be shuffled)
   private int[] original; // Array to store the original state
   private Random rand;
                            // Random number generator
   // Constructor that takes an array of integers.
   // The incoming array represents the initial state.
   public Solution(int[] nums) {
       this nums = nums; // Initialize current state with the incoming array
       this.original = Arrays.copyOf(nums, nums.length); // Copy the original array
       this.rand = new Random(); // Instantiate the Random object
   // This method resets the array to its original configuration and returns it.
   public int[] reset() {
       // Restore the original state of array
       nums = Arrays.copyOf(original, original.length);
       return nums;
   // This method returns a random shuffling of the array.
   public int[] shuffle() {
       // Loop over the array elements
        for (int i = 0; i < nums.length; ++i) {
           // Swap the current element with a randomly selected element from the remaining
           // portion of the array, starting at the current index to the end of the array.
           swap(i, i + rand.nextInt(nums.length - i));
       // Return the shuffled array
       return nums;
   // Helper method to swap two elements in the arrav.
   // Takes two indices and swaps the elements at these indices.
   private void swap(int i, int i) {
        int temp = nums[i]; // Temporary variable to hold the value of the first element
       nums[i] = nums[j]; // Assign the value of the second element to the first
       nums[j] = temp;  // Assign the value of the temporary variable to the second
```

};

/*

C++

public:

#include <vector>

class Solution {

```
Solution* obj = new Solution(nums); // Create an object of Solution with the initial array nums
std::vector<int> param 1 = obi->reset(): // Reset the array to its original configuration
std::vector<int> param 2 = obj->shuffle(); // Get a randomly shuffled array
delete obj; // Don't forget to delete the object when done to free resources
TypeScript
// Array to hold the original sequence of numbers.
let originalNums: number[] = [];
// Function to initialize the array with a set of numbers.
function initNums(nums: number[]): void {
    originalNums = nums;
// Function to return the array to its original state.
function reset(): number[] {
    // Returning a copy of the original array to prevent outside modifications.
    return [...originalNums];
// Function to randomly shuffle the elements of the array.
function shuffle(): number[] {
    const n = originalNums.length;
    // Creating a copy of the original array to shuffle.
    let shuffledNums = [...originalNums];
    // Implementing Fisher-Yates shuffle algorithm
    for (let i = 0; i < n; i++) {
        // Picking a random index within the arrav.
        const j = Math.floor(Math.random() * (i + 1));
        // Swapping elements at indices i and i.
        [shuffledNums[i], shuffledNums[j]] = [shuffledNums[j], shuffledNums[i]];
    return shuffledNums;
```

Return the shuffled list return self.nums # Example of how this class could be used: # obi = Solution(nums)

param 1 = obj.reset()

param_2 = obj.shuffle()

 Time Complexity: O(n) where n is the length of the nums list, because nums.copy() takes O(n) time. • Space Complexity: O(n), as we are creating a copy of the nums list, which requires additional space proportional to the size of the input list.

- reset method: • Time Complexity: O(n) due to the self.original.copy() operation, which again takes linear time relative to the size of the nums list.
- Space Complexity: O(n) for the new list created by self.original.copy(). shuffle method:
- Time Complexity: O(n), since it loops through the nums elements once. The operations within the loop each have a constant time complexity (j = random.randrange(i, len(self.nums)) and the swap operation), thus maintaining O(n) overall.

• Space Complexity: O(1), because the shuffling is done in place and no additional space proportional to the input size is used.