👻

384. Shuffle an Array

👾 Algo	A
▨ DESC 🦚	**Fisher-Yates shuffle algorithm**
▨ DSA 🦫	
▨ No	11
▨ Topic 🦦	

Python: mutable object references.

🔍 Problem With Using the List Directly

If you do this in `__init__`:

```
self.nums = nums
self.original = nums  # ❌ Both point to the same list
```

Then `self.original` **and** `self.nums` **refer to the exact same list object in memory.** So when you shuffle `self.nums`, it **also changes** `self.original`, because they're the same list.

✅ Why Use `nums[:]` or `nums.copy()`

When you do:

```
self.original = nums[:]
```

You are creating a **shallow copy** of the list. Now `self.original` is a **separate object** from `self.nums`. So even if `self.nums` gets modified (e.g., shuffled), you still have the **original unmodified version** saved.

That's why in `reset()`, you can safely do:

```
self.nums = self.original[:]
```

to return a fresh, unshuffled copy.

✅ Summary

Approach	Description	Problem
`self.original = nums`	Direct reference	`original` changes if `nums` changes
`self.original = nums[:]` or `nums.copy()`	Shallow copy (safe)	Preserves original values

✅ Best Practice Implementation

```
import random
from typing import List

class Solution:

    def __init__(self, nums: List[int]):
        self.original = nums[:]
        self.nums = nums[:]

    def reset(self) -> List[int]:
        self.nums = self.original[:]
        return self.nums

    def shuffle(self) → List[int]:
        shuffled = self.nums[:]
        random.shuffle(shuffled)
        return shuffled
```

Fisher-Yates Shuffle Algorithm

The Fisher-Yates shuffle works by:

1. Starting from the last element
2. For each position, randomly select an element from the remaining unshuffled portion
3. Swap the current element with the randomly selected one

```
# Tc: O(n)
# Sc: O(1)

class Solution:

    def __init__(self, nums: List[int]):
        self.nums = nums
        self.original = nums[:]

    def reset(self) → List[int]:
        self.nums = self.original[:]
        return self.nums



    def shuffle(self) → List[int]:
        for i in reversed(range(len(self.nums))):
            j = random.randint(0, i)

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

        return self.nums



# Your Solution object will be instantiated and called as such:
# obj = Solution(nums)
# param_1 = obj.reset()
# param_2 = obj.shuffle()
```

Key Points to Understand:

1. The Fisher-Yates Algorithm:

- Start from the last element (index `n-1`)
- For each position `i`, randomly pick an index `j` from `0` to `i`
- Swap elements at positions `i` and `j`
- Move to the previous position and repeat

2. Why This Works:

- Each element has an equal probability of ending up in any position
- The algorithm generates all `n!` permutations with equal probability
- Time complexity: O(n), Space complexity: O(1)

3. Important Implementation Details:

- **Copy the original array**: Use `nums[:]` to create a shallow copy, not just assign the reference
- **Reset properly**: When resetting, create a new copy of the original to avoid modifying it
- **Random range**: Use `random.randint(0, i)` to include both endpoints

4. Common Mistakes to Avoid:

- Don't use `self.original = nums` without copying - this creates a reference to the same list
- Don't shuffle from index 0 to n-1 - this doesn't produce uniform distribution
- Don't use `random.choice()` and remove elements - this is less efficient