Problem 384: Shuffle an Array

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given an integer array

nums

, design an algorithm to randomly shuffle the array. All permutations of the array should be

equally likely

as a result of the shuffling.

Implement the

Solution

class:

Solution(int[] nums)

Initializes the object with the integer array

nums

.

int[] reset()

Resets the array to its original configuration and returns it.

int[] shuffle()

Returns a random shuffling of the array.

Example 1:

Input

["Solution", "shuffle", "reset", "shuffle"] [[[1, 2, 3]], [], [], []]

Output

[null, [3, 1, 2], [1, 2, 3], [1, 3, 2]]

Explanation

Solution solution = new Solution([1, 2, 3]); solution.shuffle(); // Shuffle the array [1,2,3] and return its result. // Any permutation of [1,2,3] must be equally likely to be returned. // Example: return [3, 1, 2] solution.reset(); // Resets the array back to its original configuration [1,2,3]. Return [1, 2, 3] solution.shuffle(); // Returns the random shuffling of array [1,2,3]. Example: return [1, 3, 2]

Constraints:

1 <= nums.length <= 50

-10

6

<= nums[i] <= 10

6

All the elements of

nums

are

unique

.

At most

10

4

calls

in total

will be made to

reset

and

shuffle

.

Code Snippets

C++:

```
class Solution {
public:
Solution(vector<int>& nums) {

}

vector<int> reset() {
}

vector<int> shuffle() {

}
};

/**
* Your Solution object will be instantiated and called as such:
* Solution* obj = new Solution(nums);
* vector<int> param_1 = obj->reset();
* vector<int> param_2 = obj->shuffle();
*/
```

Java:

```
class Solution {

public Solution(int[] nums) {

}

public int[] reset() {

}

public int[] shuffle() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(nums);
* int[] param_1 = obj.reset();
* int[] param_2 = obj.shuffle();
*/
```

Python3:

```
class Solution:
```

```
def __init__(self, nums: List[int]):


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


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



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

Python:

```
class Solution(object):

def __init__(self, nums):
"""
:type nums: List[int]
"""



def reset(self):
"""
:rtype: List[int]
"""



def shuffle(self):
"""
:rtype: List[int]
"""




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

JavaScript:

```
/**
* @param {number[]} nums
*/
var Solution = function(nums) {

};

/**
* @return {number[]}
*/
Solution.prototype.reset = function() {

};

/**
* @return {number[]}
*/
Solution.prototype.shuffle = function() {

};

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

TypeScript:

```
class Solution {
constructor(nums: number[]) {

}

reset(): number[] {

}

shuffle(): number[] {
}
}

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

C#:

```
public class Solution {

public Solution(int[] nums) {

}

public int[] Reset() {

}

public int[] Shuffle() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(nums);
* int[] param_1 = obj.Reset();
* int[] param_2 = obj.Shuffle();
*/
```

C:

```
typedef struct {
} Solution;


Solution* solutionCreate(int* nums, int numsSize) {

}

int* solutionReset(Solution* obj, int* retSize) {

}

int* solutionShuffle(Solution* obj, int* retSize) {

}

void solutionFree(Solution* obj) {

}

/**
* Your Solution struct will be instantiated and called as such:
* Solution* obj = solutionCreate(nums, numsSize);
* int* param_1 = solutionReset(obj, retSize);

* int* param_2 = solutionShuffle(obj, retSize);

* solutionFree(obj);
*/
```

Go:

```
type Solution struct {

}


func Constructor(nums []int) Solution {

}


func (this *Solution) Reset() []int {
}


func (this *Solution) Shuffle() []int {

}


/**
* Your Solution object will be instantiated and called as such:
* obj := Constructor(nums);
* param_1 := obj.Reset();
* param_2 := obj.Shuffle();
*/
```

Kotlin:

```
class Solution(nums: IntArray) {

fun reset(): IntArray {

}

fun shuffle(): IntArray {

}

}

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

Swift:

```

class Solution {
```

```
init(_ nums: [Int]) {

}

func reset() -> [Int] {

}

func shuffle() -> [Int] {

}
}

/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution(nums)
* let ret_1: [Int] = obj.reset()
* let ret_2: [Int] = obj.shuffle()
*/
```

Rust:

```
struct Solution {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Solution {

fn new(nums: Vec<i32>) -> Self {

}

fn reset(&self) -> Vec<i32> {

}

fn shuffle(&self) -> Vec<i32> {
}
}

/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution::new(nums);
* let ret_1: Vec<i32> = obj.reset();
* let ret_2: Vec<i32> = obj.shuffle();
*/
```

Ruby:

```
class Solution

=begin
:type nums: Integer[]
=end
def initialize(nums)

end



=begin
:rtype: Integer[]
=end
def reset()

end



=begin
:rtype: Integer[]
=end
def shuffle()

end



end

# Your Solution object will be instantiated and called as such:
```

```
# obj = Solution.new(nums)
# param_1 = obj.reset()
# param_2 = obj.shuffle()
```

PHP:

```
class Solution {
/**
* @param Integer[] $nums
*/
function __construct($nums) {

}

/**
* @return Integer[]
*/
function reset() {

}

/**
* @return Integer[]
*/
function shuffle() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* $obj = Solution($nums);
* $ret_1 = $obj->reset();
* $ret_2 = $obj->shuffle();
*/
```

Dart:

```
class Solution {

Solution(List<int> nums) {

}

List<int> reset() {

}

List<int> shuffle() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = Solution(nums);
* List<int> param1 = obj.reset();
* List<int> param2 = obj.shuffle();
*/
```

Scala:

```
class Solution(_nums: Array[Int]) {

def reset(): Array[Int] = {

}

def shuffle(): Array[Int] = {

}

}

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

Elixir:

```
defmodule Solution do
@spec init_(nums :: [integer]) :: any
def init_(nums) do

end

@spec reset() :: [integer]
def reset() do

end

@spec shuffle() :: [integer]
def shuffle() do

end
end

# Your functions will be called as such:
# Solution.init_(nums)
# param_1 = Solution.reset()
# param_2 = Solution.shuffle()

# Solution.init_ will be called before every test case, in which you can do some necessary initializations.
```

Erlang:

```
-spec solution_init_(Nums :: [integer()]) -> any().
solution_init_(Nums) ->
.

-spec solution_reset() -> [integer()].
solution_reset() ->
.

-spec solution_shuffle() -> [integer()].
solution_shuffle() ->
.



%% Your functions will be called as such:
%% solution_init_(Nums),
%% Param_1 = solution_reset(),
%% Param_2 = solution_shuffle(),

%% solution_init_ will be called before every test case, in which you can do
some necessary initializations.
```

Racket:

```
(define solution%
(class object%
(super-new)

; nums : (listof exact-integer?)
(init-field
nums)

; reset : -> (listof exact-integer?)
(define/public (reset)
)
; shuffle : -> (listof exact-integer?)
(define/public (shuffle)
)))

;; Your solution% object will be instantiated and called as such:
;; (define obj (new solution% [nums nums]))
;; (define param_1 (send obj reset))
;; (define param_2 (send obj shuffle))
```

Solutions

C++ Solution:

```
/*
* Problem: Shuffle an Array
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class Solution {
public:
Solution(vector<int>& nums) {

}

vector<int> reset() {

}

vector<int> shuffle() {

}
};

/**
* Your Solution object will be instantiated and called as such:
* Solution* obj = new Solution(nums);
* vector<int> param_1 = obj->reset();
* vector<int> param_2 = obj->shuffle();
*/
```

Java Solution:

```
/**
* Problem: Shuffle an Array
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {

public Solution(int[] nums) {

}

public int[] reset() {

}

public int[] shuffle() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(nums);
* int[] param_1 = obj.reset();
* int[] param_2 = obj.shuffle();
*/
```

Python3 Solution:

```
"""
Problem: Shuffle an Array
Difficulty: Medium
Tags: array, math

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:

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



def reset(self) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
```

```
def __init__(self, nums):
"""
:type nums: List[int]
"""



def reset(self):
"""
:rtype: List[int]
"""



def shuffle(self):
"""
:rtype: List[int]
"""




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

JavaScript Solution:

```
/**
* Problem: Shuffle an Array
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} nums
*/
var Solution = function(nums) {
};

/**
* @return {number[]}
*/
Solution.prototype.reset = function() {

};

/**
* @return {number[]}
*/
Solution.prototype.shuffle = function() {

};

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

TypeScript Solution:

```
/**
* Problem: Shuffle an Array
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
constructor(nums: number[]) {

}

reset(): number[] {
}

shuffle(): number[] {

}
}

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

C# Solution:

```
/*
* Problem: Shuffle an Array
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {

public Solution(int[] nums) {

}

public int[] Reset() {

}

public int[] Shuffle() {

}
}
```

```
/**
* Your Solution object will be instantiated and called as such:
* Solution obj = new Solution(nums);
* int[] param_1 = obj.Reset();
* int[] param_2 = obj.Shuffle();
*/
```

C Solution:

```
/*
* Problem: Shuffle an Array
* Difficulty: Medium
* Tags: array, math
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/




typedef struct {

} Solution;


Solution* solutionCreate(int* nums, int numsSize) {

}

int* solutionReset(Solution* obj, int* retSize) {

}

int* solutionShuffle(Solution* obj, int* retSize) {

}

void solutionFree(Solution* obj) {
}

/**
* Your Solution struct will be instantiated and called as such:
* Solution* obj = solutionCreate(nums, numsSize);
* int* param_1 = solutionReset(obj, retSize);

* int* param_2 = solutionShuffle(obj, retSize);

* solutionFree(obj);
*/
```

Go Solution:

```
// Problem: Shuffle an Array
// Difficulty: Medium
// Tags: array, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

type Solution struct {

}


func Constructor(nums []int) Solution {

}


func (this *Solution) Reset() []int {

}


func (this *Solution) Shuffle() []int {

}
```

}

```
/**
* Your Solution object will be instantiated and called as such:
* obj := Constructor(nums);
* param_1 := obj.Reset();
* param_2 := obj.Shuffle();
*/
```

Kotlin Solution:

```
class Solution(nums: IntArray) {

fun reset(): IntArray {

}

fun shuffle(): IntArray {

}

}

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

Swift Solution:

```
class Solution {

init(_ nums: [Int]) {

}

func reset() -> [Int] {
}

func shuffle() -> [Int] {

}
}

/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution(nums)
* let ret_1: [Int] = obj.reset()
* let ret_2: [Int] = obj.shuffle()
*/
```

Rust Solution:

```
// Problem: Shuffle an Array
// Difficulty: Medium
// Tags: array, math
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

struct Solution {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Solution {

fn new(nums: Vec<i32>) -> Self {

}

fn reset(&self) -> Vec<i32> {
}

fn shuffle(&self) -> Vec<i32> {

}
}

/**
* Your Solution object will be instantiated and called as such:
* let obj = Solution::new(nums);
* let ret_1: Vec<i32> = obj.reset();
* let ret_2: Vec<i32> = obj.shuffle();
*/
```

Ruby Solution:

```
class Solution

=begin
:type nums: Integer[]
=end
def initialize(nums)

end



=begin
:rtype: Integer[]
=end
def reset()

end



=begin
:rtype: Integer[]
=end
def shuffle()

end
```

```
end

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

PHP Solution:

```
class Solution {
/**
* @param Integer[] $nums
*/
function __construct($nums) {

}

/**
* @return Integer[]
*/
function reset() {

}

/**
* @return Integer[]
*/
function shuffle() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* $obj = Solution($nums);
* $ret_1 = $obj->reset();
* $ret_2 = $obj->shuffle();
*/
```

Dart Solution:

```
class Solution {

Solution(List<int> nums) {

}

List<int> reset() {

}

List<int> shuffle() {

}
}

/**
* Your Solution object will be instantiated and called as such:
* Solution obj = Solution(nums);
* List<int> param1 = obj.reset();
* List<int> param2 = obj.shuffle();
*/
```

Scala Solution:

```
class Solution(_nums: Array[Int]) {

def reset(): Array[Int] = {

}

def shuffle(): Array[Int] = {

}

}

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

Elixir Solution:

```
defmodule Solution do
@spec init_(nums :: [integer]) :: any
def init_(nums) do

end

@spec reset() :: [integer]
def reset() do

end

@spec shuffle() :: [integer]
def shuffle() do

end
end

# Your functions will be called as such:
# Solution.init_(nums)
# param_1 = Solution.reset()
# param_2 = Solution.shuffle()

# Solution.init_ will be called before every test case, in which you can do
some necessary initializations.
```

Erlang Solution:

```
-spec solution_init_(Nums :: [integer()]) -> any().
solution_init_(Nums) ->
.

-spec solution_reset() -> [integer()].
solution_reset() ->
.

-spec solution_shuffle() -> [integer()].
solution_shuffle() ->
.
```

```
%% Your functions will be called as such:
%% solution_init_(Nums),
%% Param_1 = solution_reset(),
%% Param_2 = solution_shuffle(),

%% solution_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket Solution:

```
(define solution%
(class object%
(super-new)

; nums : (listof exact-integer?)
(init-field
nums)

; reset : -> (listof exact-integer?)
(define/public (reset)
)
; shuffle : -> (listof exact-integer?)
(define/public (shuffle)
)))

;; Your solution% object will be instantiated and called as such:
;; (define obj (new solution% [nums nums]))
;; (define param_1 (send obj reset))
;; (define param_2 (send obj shuffle))
```