Problem 1920: Build Array from Permutation

Problem Information

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

Problem Description

Given a

zero-based permutation

nums

(

0-indexed

), build an array

ans

of the

same length

where

ans[i] = nums[nums[i]]

for each

0 <= i < nums.length

and return it.

A

zero-based permutation

nums

is an array of

distinct

integers from

0

to

nums.length - 1

(

inclusive

).

Example 1:

Input:

nums = [0,2,1,5,3,4]

Output:

[0,1,2,4,5,3]

Explanation:

The array ans is built as follows: ans = [nums[nums[0]], nums[nums[1]], nums[nums[2]], nums[nums[3]], nums[nums[4]], nums[nums[5]]] = [nums[0], nums[2], nums[1], nums[5], nums[3], nums[4]] = [0,1,2,4,5,3]

Example 2:

Input:

nums = [5,0,1,2,3,4]

Output:

[4,5,0,1,2,3]

Explanation:

The array ans is built as follows: ans = [nums[nums[0]], nums[nums[1]], nums[nums[2]], nums[nums[3]], nums[nums[4]], nums[nums[5]]] = [nums[5], nums[0], nums[1], nums[2], nums[3], nums[4]] = [4,5,0,1,2,3]

Constraints:

1 <= nums.length <= 1000

0 <= nums[i] < nums.length

The elements in

nums

are

distinct

.

Follow-up:

Can you solve it without using an extra space (i.e., O(1)

memory)?

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int[] buildArray(int[] nums) {

}
}
```

Python3:

```
class Solution:
def buildArray(self, nums: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def buildArray(self, nums):
"""
:type nums: List[int]
:rtype: List[int]
"""
```

JavaScript:

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

};
```

TypeScript:

```
function buildArray(nums: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] BuildArray(int[] nums) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* buildArray(int* nums, int numsSize, int* returnSize) {

}
```

Go:

```
func buildArray(nums []int) []int {

}
```

Kotlin:

```
class Solution {
fun buildArray(nums: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func buildArray(_ nums: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn build_array(nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @return {Integer[]}
def build_array(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer[]
*/
function buildArray($nums) {

}
}
```

Dart:

```
class Solution {
List<int> buildArray(List<int> nums) {

}
}
```

Scala:

```
object Solution {
def buildArray(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (build-array nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Build Array from Permutation
* Difficulty: Easy
* Tags: array
*
```

```
* 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:
vector<int> buildArray(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Build Array from Permutation
* Difficulty: Easy
* Tags: array
*
* 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 int[] buildArray(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Build Array from Permutation
Difficulty: Easy
Tags: array

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 buildArray(self, nums: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def buildArray(self, nums):
"""
:type nums: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Build Array from Permutation
* Difficulty: Easy
* Tags: array
*
* 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
* @return {number[]}
*/
var buildArray = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Build Array from Permutation
* Difficulty: Easy
* Tags: array
```

```
*
* 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
*/

function buildArray(nums: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Build Array from Permutation
* Difficulty: Easy
* Tags: array
*
* 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 int[] BuildArray(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Build Array from Permutation
* Difficulty: Easy
* Tags: array
*
* 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
*/

/**
```

```
* Note: The returned array must be malloced, assume caller calls free().
*/
int* buildArray(int* nums, int numsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Build Array from Permutation
// Difficulty: Easy
// Tags: array
//
// 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

func buildArray(nums []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun buildArray(nums: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func buildArray(_ nums: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Build Array from Permutation
// Difficulty: Easy
// Tags: array
```

```
//
// 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

impl Solution {
pub fn build_array(nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @return {Integer[]}
def build_array(nums)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer[]
*/
function buildArray($nums) {

}
}
```

Dart Solution:

```
class Solution {
List<int> buildArray(List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def buildArray(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (build-array nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```