Problem 1053: Previous Permutation With One Swap

Problem Information

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

Problem Description

Given an array of positive integers

arr

(not necessarily distinct), return

the

lexicographically

largest permutation that is smaller than

arr

, that can be

made with exactly one swap

. If it cannot be done, then return the same array.

Note

that a

swap

exchanges the positions of two numbers

arr[i]

and

arr[j]

Example 1:

Input:

arr = [3,2,1]

Output:

[3,1,2]

Explanation:

Swapping 2 and 1.

Example 2:

Input:

arr = [1,1,5]

Output:

[1,1,5]

Explanation:

This is already the smallest permutation.

Example 3:

Input:

arr = [1,9,4,6,7]

Output:

[1,7,4,6,9]

Explanation:

Swapping 9 and 7.

Constraints:

1 <= arr.length <= 10

4

1 <= arr[i] <= 10

4

Code Snippets

C++:

```
class Solution {
public:
vector<int> prevPermOpt1(vector<int>& arr) {

}
};
```

Java:

```
class Solution {
public int[] prevPermOpt1(int[] arr) {

}
```

```
}
```

Python3:

```
class Solution:
def prevPermOpt1(self, arr: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def prevPermOpt1(self, arr):
"""
:type arr: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} arr
* @return {number[]}
*/
var prevPermOpt1 = function(arr) {

};
```

TypeScript:

```
function prevPermOpt1(arr: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] PrevPermOpt1(int[] arr) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* prevPermOpt1(int* arr, int arrSize, int* returnSize) {

}
```

Go:

```
func prevPermOpt1(arr []int) []int {

}
```

Kotlin:

```
class Solution {
fun prevPermOpt1(arr: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func prevPermOpt1(_ arr: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn prev_perm_opt1(arr: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} arr
# @return {Integer[]}
def prev_perm_opt1(arr)
```

```
end
```

PHP:

```
class Solution {

/**
* @param Integer[] $arr
* @return Integer[]
*/
function prevPermOpt1($arr) {

}
}
```

Dart:

```
class Solution {
List<int> prevPermOpt1(List<int> arr) {

}
}
```

Scala:

```
object Solution {
def prevPermOpt1(arr: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec prev_perm_opt1(arr :: [integer]) :: [integer]
def prev_perm_opt1(arr) do

end
end
```

Erlang:

```
-spec prev_perm_opt1(Arr :: [integer()]) -> [integer()].
prev_perm_opt1(Arr) ->
.
```

Racket:

```
(define/contract (prev-perm-opt1 arr)
(-> (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Previous Permutation With One Swap
* Difficulty: Medium
* Tags: array, graph, greedy
*
* 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> prevPermOpt1(vector<int>& arr) {

}
};
```

Java Solution:

```
/**
* Problem: Previous Permutation With One Swap
* Difficulty: Medium
* Tags: array, graph, greedy
*
* 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[] prevPermOpt1(int[] arr) {

}
}
```

Python3 Solution:

```
"""
Problem: Previous Permutation With One Swap
Difficulty: Medium
Tags: array, graph, greedy

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 prevPermOpt1(self, arr: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def prevPermOpt1(self, arr):
"""
:type arr: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Previous Permutation With One Swap
* Difficulty: Medium
* Tags: array, graph, greedy
*
```

```
* 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[]} arr
* @return {number[]}
*/
var prevPermOpt1 = function(arr) {

};
```

TypeScript Solution:

```
/**
* Problem: Previous Permutation With One Swap
* Difficulty: Medium
* Tags: array, graph, greedy
*
* 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 prevPermOpt1(arr: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Previous Permutation With One Swap
* Difficulty: Medium
* Tags: array, graph, greedy
*
* 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[] PrevPermOpt1(int[] arr) {

}
}
```

C Solution:

```
/*
* Problem: Previous Permutation With One Swap
* Difficulty: Medium
* Tags: array, graph, greedy
*
* 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* prevPermOpt1(int* arr, int arrSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Previous Permutation With One Swap
// Difficulty: Medium
// Tags: array, graph, greedy
//
// 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 prevPermOpt1(arr []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun prevPermOpt1(arr: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func prevPermOpt1(_ arr: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Previous Permutation With One Swap
// Difficulty: Medium
// Tags: array, graph, greedy
//
// 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 prev_perm_opt1(arr: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} arr
# @return {Integer[]}
def prev_perm_opt1(arr)

end
```

PHP Solution:

```
class Solution {
```

```
/**
* @param Integer[] $arr
* @return Integer[]
*/
function prevPermOpt1($arr) {

}
}
```

Dart Solution:

```
class Solution {
List<int> prevPermOpt1(List<int> arr) {

}
}
```

Scala Solution:

```
object Solution {
def prevPermOpt1(arr: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec prev_perm_opt1(arr :: [integer]) :: [integer]
def prev_perm_opt1(arr) do

end
end
```

Erlang Solution:

```
-spec prev_perm_opt1(Arr :: [integer()]) -> [integer()].
prev_perm_opt1(Arr) ->
.
```

Racket Solution:

```
(define/contract (prev-perm-opt1 arr)
(-> (listof exact-integer?) (listof exact-integer?))
)
```