## 31. Next Permutation

#### Medium Array **Two Pointers**

# **Problem Description**

arranging the integers in a different sequence. If we consider all possible permutations of an array sorted in lexicographical order (dictionary order), the 'next permutation' is the permutation that comes directly after the current one in this sorted list. The challenge here is to generate this next permutation efficiently, using constant extra space (in-place). If the array is sorted in descending order, there is no next permutation, and the array should be rearranged to its lowest possible

The problem asks us to find the next permutation of an array of integers in lexicographical order. A permutation refers to

order (ascending order). We need to identify a method for transforming the arrangement of the array to the next permutation without generating all permutations. For example, given nums = [1,2,3], the next permutation is [1,3,2]. Given nums = [3,2,1], as it's the last permutation in

lexicographical order, the next permutation would reset to [1,2,3].

To find the next permutation, we need to understand how permutations can be ordered lexicographically. We must identify the

### smallest change in the sequence that leads to a larger permutation. The process of finding the next permutation can typically be

broken down into the following steps: Find the Longest Decreasing Suffix: We traverse from the end of the array moving backward, looking for the first pair of elements where the previous element is smaller than the next one (nums[i] < nums[i+1]). This identifies the boundary of the

longest decreasing (non-increasing) suffix of the array.

necessary to form the next permutation.

- Identify the Pivot: The element just before the non-increasing suffix is called the pivot. If no pivot is found, it implies the entire array is non-increasing, which means we are at the last permutation and the next one is the first permutation (sorted in ascending order).
- larger than the pivot. This successor will be the one we swap with the pivot to ensure we get the next larger permutation. Perform the Swap and Reverse the Suffix: As the suffix is in decreasing order, to get the next permutation, we swap the pivot with its successor and then reverse the suffix, turning it from decreasing to increasing, which gives us the least increase

Find a Successor to the Pivot: We again traverse from the end of the array moving backward, looking for the first element

The intuition behind this approach is that we want to increase the sequence as little as possible - only enough to make it the "next" permutation lexicographically. Swapping the pivot with the smallest element larger than it in the decreasing suffix ensures this minimal increase. Reversing the decreasing suffix guarantees that the sequence remains as small as possible after the pivot.

The solution involves a carefully crafted algorithm, which includes identifying key positions in the array and manipulating it using basic operations like swapping elements and reversing a sub-array. Here's a step-by-step walk-through: Traverse to Find the Pivot: We start by traversing the array from the end to the beginning. We look for the first index i where

# completed in O(n) time complexity, where n is the length of the array.

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

the need to generate all permutations.

def nextPermutation(self, nums: List[int]) -> None:

for i in range(length -2, -1, -1):

if nums[j] > nums[pivot]:

nums[pivot + 1:] = reversed(nums[pivot + 1:])

# This mutates 'nums' in-place to the next permutation

# Swap the 'pivot' with this element

nums[pivot], nums[j] = nums[j], nums[pivot]

# to get the lowest possible sequence with the 'pivot' being the prefix

# Finally, reverse the elements following the 'pivot' (inclusive if pivot is -1)

if nums[i] < nums[i + 1]:</pre>

pivot = i

break

break

end--;

int temp = nums[i];

#include <algorithm> // For std::reverse

nums[i] = nums[j];

nums[j] = temp;

--i;

C++

public:

#include <vector>

class Solution {

// Helper function to swap elements in the array

// Rearranges numbers into the lexicographically next greater permutation

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

// Reverse the sequence from nums[i + 1] to the end of the array

\* Rearranges numbers into the lexicographically next greater permutation of numbers.

// This is to get the smallest possible permutation after i

// Find the first element that is not in a non-increasing sequence from the end

// Find the smallest element greater than nums[i] to the right of it

private void swap(int[] nums, int i, int j) {

void nextPermutation(std::vector<int>& nums) {

while (i >= 0 && nums[i] >= nums[i + 1]) {

if (i >= 0) { // If such an element is found

std::reverse(nums.begin() + i + 1, nums.end());

for (int j = n - 1; j > i; ---j) {

if (nums[j] > nums[i]) {

break;

int n = nums.size(); // Size of the given array

int i = n - 2; // Start from the second last element

Solution Implementation

**Python** 

Java

class Solution:

Solution Approach

Find the Successor to the Pivot: If a pivot is found (the array is not entirely non-increasing), we again traverse the array from the end to find the first index j where nums[j] > nums[i]. The element at j is the smallest element greater than the pivot within the suffix. This step ensures we get the next permutation.

nums[i] < nums[i + 1]. This step locates the longest non-increasing suffix, and i is identified as the pivot. This is

Swap the Pivot and its Successor: We swap the elements at i and j. Now, the pivot is at the place of its immediate

In-Place and Constant Space: The entire operation does not need any additional storage as all operations are performed on

Reverse the Suffix: Finally, the suffix starting from i+1 till the end of the array is reversed. Since the suffix was in a nonincreasing order, reversing it will change it to non-decreasing order. This ensures the remainder of the array is as low as possible to be the next permutation after the incremented pivot.

the input array itself. The space complexity is 0(1) since no additional space is required regardless of the input size.

successor which is the minimum necessary increment to the current permutation.

i = next((i for i in range(n - 2, -1, -1) if nums[i] < nums[i + 1]), -1)

# Reverse the suffix. This is done in place, and Python's slicing is used.

# The default value -1 is used if no pivot is found.

# If a pivot is found (i is not -1 after applying ~ operator)

detailed approach referring to the provided code: def nextPermutation(self, nums: List[int]) -> None: n = len(nums)# Finding the pivot using a generator expression.

The provided solution uses Python's generator expressions and the tilde operator (~) for a concise implementation. Here is the

if ~i: # Finding the successor to pivot (nums[i] > nums[i]). j = next((j for j in range(n - 1, i, -1) if nums[j] > nums[i]))# Swap pivot with its successor. nums[i], nums[j] = nums[j], nums[i]

```
nums[i + 1 :] = nums[i + 1 :][::-1]
  Here the next method is used from Python's built-in functionality, which returns the next item from the iterator. If the iterator is
  exhausted, a default value is returned (in this case, -1). By using a generator expression within the next method, we effectively
  condense the loop to find the pivot and successor in a single line each. The ~ operator is a bitwise complement operator, which,
  when applied to -1, results in 0, thereby allowing us to check if i is not -1 in a concise manner.
  Understanding this algorithm's steps and carefully executing them in the correct order are crucial to finding the next permutation
  effectively.
Example Walkthrough
  Let's walk through a small example to illustrate the solution approach. Consider the following array of integers:
```

approach: Traverse to Find the Pivot: We start from the end of the array and look for the first pair where the number is less than its successor. Traverse: 2 < 4 (stop here). So, the pivot is 3 at index 1. Find the Successor to the Pivot: We need to find the smallest number greater than the pivot 3, starting from the end of the

We want to find the next permutation for this array in lexicographical order. Following the steps mentioned in the solution

Reverse the Suffix: We reverse the suffix starting right after the pivot's new position (index 1) till the end. The suffix is [5, 3, 2] and its reverse is [2, 3, 5]. The array after reversing the suffix is: [1, 4, 2, 3, 5].

By performing these steps, we successfully found the next permutation in lexicographical order using constant space and without

# The length of the 'nums' list length = len(nums)# First, find the first element from the right that is smaller than the element next to it. pivot = -1

array. Traverse: 2 (not greater), 4 (greater, stop here). The successor is 4 at index 3.

Swap the Pivot and its Successor: We swap 3 and 4. The array is now: [1, 4, 5, 3, 2].

The resulting array [1, 4, 2, 3, 5] is the next permutation of the input array [1, 3, 5, 4, 2].

# If such an element was found, then we can form the next permutation if pivot !=-1: # Now, we find the smallest element greater than the 'pivot', starting from the end for j in range(length - 1, pivot, -1):

```
class Solution {
   public void nextPermutation(int[] nums) {
       // Length of the array
       int length = nums.length;
       // Initialize the index i to start from the second last element
       int i = length - 2;
       // Find the first pair of two successive numbers a[i] and a[i+1], from right to left, which satisfy a[i] < a[i+1]
       while (i >= 0 && nums[i] >= nums[i + 1]) {
          i--;
       // If such pair was found, i denotes the pivot
       if (i >= 0) {
           // Find the rightmost successor to the pivot
           for (int i = length - 1; i > i; j--) {
               if (nums[i] > nums[i]) {
                   // Swap the successor and the pivot
                    swap(nums, i, j);
                   break;
       // Reverse the suffix starting right after the pivot point
       int start = i + 1, end = length - 1;
       while (start < end) {</pre>
           // Swap the start and end elements of the suffix
           swap(nums, start, end);
           start++;
```

**}**;

/\*\*

**TypeScript** 

\* Modifies the array in place.

```
* @param {number[]} nums - An array of integers to find the next permutation for.
function nextPermutation(nums: number[]): void {
   // Get the length of the numbers array.
   const length = nums.length;
   // Start from the second last element and move backwards to find the first pair where the left index is less than the right index
   let pivotIndex = length - 2;
   while (pivotIndex >= 0 && nums[pivotIndex] >= nums[pivotIndex + 1]) {
       pivotIndex--;
   // If such pair is found, find the number just larger than the pivot to swap with.
   if (pivotIndex >= 0) {
        for (let swapIndex = length - 1; swapIndex > pivotIndex; swapIndex--) {
            if (nums[swapIndex] > nums[pivotIndex]) {
                // Swap the pivot with the number just larger than it.
                [nums[pivotIndex], nums[swapIndex]] = [nums[swapIndex], nums[pivotIndex]];
               break;
   // Reverse the numbers to the right of pivotIndex to get the next smallest lexicographic permutation.
   let start = pivotIndex + 1;
   let end = length -1;
   while (start < end) {</pre>
        [nums[start], nums[end]] = [nums[end], nums[start]];
       start++;
       end--;
class Solution:
   def nextPermutation(self, nums: List[int]) -> None:
       # The length of the 'nums' list
        length = len(nums)
```

# First, find the first element from the right that is smaller than the element next to it.

# Now, we find the smallest element greater than the 'pivot', starting from the end

# If such an element was found, then we can form the next permutation

nums[pivot], nums[j] = nums[j], nums[pivot]

# to get the lowest possible sequence with the 'pivot' being the prefix

# Finally, reverse the elements following the 'pivot' (inclusive if pivot is -1)

# Swap the 'pivot' with this element

\* If such arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).

# Time and Space Complexity The function nextPermutation manipulates an array nums to transform it into the next permutation in lexicographically increasing

elements to the left of j.

pivot = -1

for i in range(length -2, -1, -1):

for i in range(length - 1, pivot, -1):

nums[pivot + 1:] = reversed(nums[pivot + 1:])

# This mutates 'nums' in-place to the next permutation

if nums[j] > nums[pivot]:

if nums[i] < nums[i + 1]:</pre>

pivot = i

break

break

if pivot !=-1:

order. Analyzing the code block by block: i is found by iterating backwards, starting from the second to last element, to find the first number that is smaller than the

- one directly after it. In the worst case, this takes O(n) time, where n is the size of the nums list. If such an i is found (i.e., the permutation is not the last one), the code looks for the smallest number greater than nums[i] after the index i, and swaps nums[i] with nums[j]. This operation is also 0(n) since in the worst case it might scan all
- The last line reverses the segment of the list after the ith index. This operation has a complexity of O(n), since reversing a list is linear in the length of the segment being reversed.
- There are no additional data structures that depend on the size of the input, so the space complexity remains constant, i.e., 0(1).

In summary, even though there are multiple sequential O(n) operations, the overall time complexity is still O(n) because we do not perform these operations more than once for the entire input. Therefore, the reference answer's claim of O(n) time complexity and 0(1) space complexity is correct.