2717. Semi-Ordered Permutation

Easy Array Simulation Leetcode Link

Problem Description

In this problem, we are given an array `nums` which represents a permutation of `n` integers. To recall, a permutation is a sequence containing each number from `1` to `n` exactly once. The goal is to make this permutation semi-ordered using a specific operation. A permutation is said to be semi-ordered if the first element is `1` and the last element is `n`. The operation we can perform is to swap any two adjacent elements, and we can do this as many times as necessary.

The task is to return the minimum number of such swap operations required to transform the given permutation into a semi-ordered permutation.

Intuition

To arrive at the solution, consider the fact that we only need to focus on the positions of the numbers `1` and `n` in the permutation, because these are the elements that need to be at the start and end of the array, respectively.

Here's the step-by-step thought process:

1. Find the positions of `1` and `n` in the array. These positions will determine the number of swap operations needed.
2. If the number `1` is already at the start or the number `n` is already at the end of the array, no operations are needed for those numbers. Otherwise, we would need to swap each of them until they reach their correct positions. This can be calculated by the number of positions each needs to move.
3. Since we can swap any two adjacent elements, we can independently move `1` to the start and `n` to the end.
4. If the initial position of `1` is before the position of `n`, we have an overlap situation when moving both `1` and `n`. Therefore, one operation that moves `1` to the left also moves `n` to the right, saving us one operation.
5. Combining these observations, we can compute the minimum number of operations needed.

The intuition behind the algorithm is based on optimizing the process of arranging `1` at the start and `n` at the end, by counting the movements without actually performing the swaps, and then subtracting the overlapping move if `1` comes before `n`.

Solution Approach

The implementation of this solution involves a very straightforward approach without the need for any complicated data structures or algorithms. It follows directly from the observation that we only need to move `1` to the start and `n` to the end of the array to achieve a semi-ordered permutation. The Python code provided does just that in a few steps:

1. Determine the length of the array `nums` to know how many elements we are dealing with, as this tells us the index of the last element in a zero-indexed array.
2. Find the index of `1` in the `nums` array using the `index` method, which will return the position `i` where `1` is located. Similarly, find the index of `n` and store it in the variable `j`.
3. Calculate the number of operations needed to move `1` to the start of the array. This is simply the index `i` since `1` needs to move `i` places to the left to reach the starting position.
4. Similarly, calculate the operations needed to move `n` to the end. The index `j` represents the position of `n`, and since it needs to move to the last position, we subtract `j` from `(n-1)` to find out how many places `n` needs to move right. `n-1` is used because the array is zero-indexed.
5. Using a conditional expression, we determine if `1` comes before `n` (`i < j`). If so, there is one overlapping move when both `1` and `n` are moving towards their destined positions (since swapping to move `1` to the start simultaneously moves `n` closer to the end). In this case, one less operation (`k`) is needed.
6. Sum the operations needed to move `1` and `n`, and subtract the overlap `k` if there is any. This sum gives us the minimum number of operations required.
7. Return this sum as the solution to the problem.

The pattern used here is mainly observation of the problem constraints and taking advantage of the properties of the permutation to find an efficient solution that runs in constant time, as it only involves index lookups and arithmetic operations.

The code doesn't use any complex algorithms or data structures because it doesn't perform the actual swap operations but rather calculates the moves needed based on the indexes of `1` and `n`.

By following this simple logic, we achieve an optimal solution to the problem.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose the input array `nums` is `[4, 3, 2, 1, 5]`, which is a permutation of 5 integers. As per the problem, we need `1` at the start and `5` at the end in the fewest swaps possible.

Here's a step-by-step walkthrough:

1. Identify the initial positions of `1` and `5`. In `nums`, `1` is at index 3 (`i`) and `5` is at index 4 (`j`). Our task is to move `1` to index 0 and `5` to index 4 (since `5` is already at the end, it stays in place).
2. Since `1` is at index 3, it needs `i = 3` swaps to reach the start of the array.
3. For `5`, no swaps are needed since it is already at the last position, `j = 4`.
4. Check whether there's an overlap. Since `i < j`, there is no overlap because `1` can move to the start without affecting the position of `5`.
5. Now, simply calculate the total number of swaps, which in our case would be the swaps needed for `1`, plus the swaps for `5` (which are zero), minus any overlap (also zero in this case).

   We calculate swaps as follows:

   - Swaps for `1`: `i` (3 swaps)
   - Swaps for `5`: `(n-1) - j` (0 swaps since 5 is already at the end)
   - Overlap: `k` (0 because `i < j`)

   So, total swaps required: `3 + 0 - 0 = 3`

Therefore, we can transform the permutation `[4, 3, 2, 1, 5]` into a semi-ordered permutation with a minimum of 3 swap operations:

Starting with: `[4, 3, 2, 1, 5]` 1st swap: `[4, 3, 1, 2, 5]` 2nd swap: `[4, 1, 3, 2, 5]` 3rd swap: `[1, 4, 3, 2, 5]`

The array is now semi-ordered, with `1` at the start and `5` at the end. We return 3 as the solution to the number of swaps needed.

Python Solution

```python
from typing import List

class Solution:
    def semiOrderedPermutation(self, nums: List[int]) -> int:
        # Calculate the length of the input list
        length_of_nums = len(nums)

        # Find the index positions of the smallest and largest elements (1 and n)
        index_of_min = nums.index(1)
        index_of_max = nums.index(length_of_nums)

        # Determine the correction factor based on the relative position of min and max
        correction_factor = 1 if index_of_min < index_of_max else 2

        # Calculate and return the result as per the given formula
        result = index_of_min + length_of_nums - index_of_max - correction_factor
        return result

```

Java Solution

```java
class Solution {
    // Method to calculate the semi-ordered permutation value
    public int semiOrderedPermutation(int[] nums) {
        int length = nums.length; // Total number of elements in the array
        int indexOne = 0; // Initialize the index of the element '1'
        int indexN = 0; // Initialize the index of the element 'n', where 'n' is the length of the array

        // Iterate through the array to find the indices of 1 and n
        for (int k = 0; k < length; ++k) {
            if (nums[k] == 1) {
                indexOne = k; // Found the index of 1
            }
            if (nums[k] == length) {
                indexN = k; // Found the index of the last element 'n'
            }
        }

        // Determine the subtraction factor depending on the positions of 1 and n
        int subtractionFactor = indexOne < indexN ? 1 : 2;

        // Calculate and return the final semi-ordered permutation value
        // by adding the position of 1, subtracting the position of n
        // and then adjusting with the subtraction factor
        return indexOne + length - indexN - subtractionFactor;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm library for std::find

class Solution {
public:
    // Function to calculate the semi-ordered permutation count.
    int semiOrderedPermutation(vector<int>& nums) {
        int size = nums.size(); // Get the size of the nums vector
        // Find the position (index) of the smallest element (1)
        int indexMin = std::find(nums.begin(), nums.end(), 1) - nums.begin();
        // Find the position (index) of the largest element (n)
        int indexMax = std::find(nums.begin(), nums.end(), size) - nums.begin();

        // Determine if smallest element comes before largest element
        // if so, we use k = 1, if not k = 2 for the count adjustment
        int k = (indexMin < indexMax) ? 1 : 2;

        // Calculate the semi-ordered permutation count by subtracting
        // the irrelevant elements using the indexes and k
        return indexMin + size - indexMax - k;
    }
};

```

Typescript Solution

```typescript
function semiOrderedPermutation(nums: number[]): number {
    // Get the length of the input array.
    const length = nums.length;

    // Find the index of the smallest element, which is 1.
    const index1 = nums.indexOf(1);

    // Find the index of the largest element, which is the length of the array.
    const indexN = nums.indexOf(length);

    // Determine if the smallest element's index is less than that of the largest.
    // Choose 1 if smallest is before largest, otherwise 2.
    const adjustmentFactor = index1 < indexN ? 1 : 2;

    // Return the difference between the sum of indexes and the adjustment factor.
    // This computes the gap between '1' and 'n', adjusting for inclusive/exclusive indexing.
    return index1 + length - indexN - adjustmentFactor;
}

```

Time and Space Complexity

The given Python function `semiOrderedPermutation` calculates a result based on the positions of the smallest and largest elements in a list.

Time Complexity

The time complexity of the function is determined by the following operations:

1. `n = len(nums)` — Getting the length of the list, which is an O(1) operation.
2. `i = nums.index(1)` — Finding the index of the element 1 in the list, which is an O(N) operation in the worst case, where N is the number of elements in the list.
3. `j = nums.index(n)` — Finding the index of the largest element (which is `n`), also an O(N) operation in the worst case.
4. The comparison `i < j` — A simple comparison, which is an O(1) operation.
5. The last line is simple arithmetic and an assignment, all of which are O(1) operations.

The function contains no loops or recursive calls that depend on the size of the input list. The most time-consuming operations are the two index lookups, each of which is O(N). Therefore, the overall time complexity of the function is O(N), where N is the length of the input list `nums`.

Space Complexity

The space complexity is determined by the additional memory used by the function:

1. Variables `n`, `i`, `j`, and `k` are all integers, each requiring a constant amount of space.
2. The function does not use any additional data structures that grow with the input size.

Hence, the space complexity of the function is O(1), which means it uses a constant amount of additional space regardless of the input size.