1053. Previous Permutation With One Swap

Medium Greedy Array

Problem Description

Given an array of positive integers `arr`, the objective is to find the lexicographically largest permutation that is smaller than `arr`, with the condition that only one swap of two elements is allowed. If no such permutation exists (meaning `arr` is the smallest possible permutation), then the function should return the array as it is.

A permutation is considered lexicographically smaller if at the first position where the two permutations differ, the number in the smaller permutation is smaller than in the larger one. In simpler terms, it's similar to the alphabetical order, but with numbers.

The problem includes a `swap` operation, which involves exchanging the positions of two numbers in the array: `arr[i]` and `arr[j]`.

Intuition

To solve this problem, we need to think in reverse order, starting from the last element towards the first element of the array. Our goal is to find the first pair where the left element is greater than the right element (`arr[i - 1] > arr[i]`). This specific pair gives us the point beyond which all elements are in non-increasing order.

Once we have that pivot, we need to find the largest possible number to the right of it that is still smaller than the pivot number itself, because we want the largest permutation smaller than the current array. We must also take care of duplicates because swapping with a duplicate won't change the permutation.

We iterate from the end of the array until we find an element that satisfies these conditions. We swap these two elements (the pivot and the chosen element to its right) and return the modified array. If the loop finishes without finding such an element, no swap can satisfy the condition, and the input array is returned unchanged as it's already the smallest permutation.

Solution Approach

To implement the solution, we use a simple approach to iterate through the array without any complicated data structures or patterns. Here's a step-by-step breakdown of what the given Python function does:

1. Determine the length of the given `arr` and store it in variable `n`.
2. Start iterating over the array backwards using a `for` loop, checking each adjacent pair (`arr[i - 1], arr[i]`) to find the first instance where `arr[i - 1]` is greater than `arr[i]`. This means that `arr[i - 1]` is our potential element to be swapped as we aim to make the largest number smaller.
3. When we find such an element at `arr[i - 1]`, we need to find the best candidate to swap it with to get the lexicographically next smaller permutation. To do this, another loop iterates over the array from the end to `i - 1`.
4. Inside this nested loop, we look for an element `arr[j]` that is smaller than `arr[i - 1]` but also is not equal to `arr[j - 1]` (to handle duplicates). We are trying to find the highest valued element smaller than `arr[i - 1]` to make the simplest swap.
5. Once the correct element to swap with is found (`arr[j]`), we swap `arr[i - 1]` with `arr[j]` and return the array immediately.
6. If no such element is found while iterating, it means the array is already the smallest permutation possible with the given digits, so we return the original array without any modifications.

The solution's time complexity is O(n^2) in the worst case, where `n` is the size of the array. This occurs when we need to perform a nested loop iteration for every element in the array. However, on average, the time complexity could be better if the swap is found early.

Example Walkthrough

Let's take an example array `arr = [3, 2, 1, 4, 5]` and illustrate the solution approach to find the lexicographically largest permutation that is smaller than `arr` using only one swap.

1. Determine the length of `arr` which is `5` in this case.
2. Start iterating from the end to find the first adjacent pair where the left number is greater than the right number. We compare the pairs `(5, 4)`, `(4, 1)`, `(1, 2)`, and finally `(2, 3)`. Upon comparing `(2, 3)`, we find `3` is greater than `2`, which means we have our pivot element `3` at index `0`.
3. Now, we need to find the best candidate to swap with `3` to get the lexicographically next smaller permutation. So, we start another loop from the end to the index just right of our pivot (index `1`).
4. As we iterate, we look for an element smaller than our pivot `3`, so we consider `5`, but it's not smaller. We move to `4` and `1`. We find `1` fulfills our condition, being smaller than `3` and also not repeated. It is the largest number smaller than `3` to the right of the pivot.
5. We swap the pivot `3` with the number `1`, resulting in the array `[1, 2, 3, 4, 5]`.
6. Return the modified array, which is now `[1, 2, 3, 4, 5]`. This is the lexicographically largest permutation smaller than the original array which could be achieved through a single swap.

Summarizing the steps with the chosen array:

- Identify pivot (`3`) which is greater than its right neighbor (`2`).
- Find the best swap candidate (`1`) which is smaller than the pivot but still the largest on the right.
- Swap pivot with the candidate and return the result.

In this example, the iterations and swap were straightforward and led us to the lexicographically largest permutation smaller than the original array by just moving the pivot `3` to the position of `1`. The original array is left in an increasing order, as expected after the swap.

Solution Implementation

Python

```python
class Solution:
    def prevPermOpt1(self, arr):
        # Length of the array
        length = len(arr)

        # Loop backwards through the array starting from the second last element
        for i in range(length - 1, 0, -1):
            # Check if the current element is greater than its following element
            # meaning a smaller permutation is possible
            if arr[i - 1] > arr[i]:
                # Find the element to swap with, which is the rightmost element
                # that is smaller than arr[i - 1] and not a duplicate of its previous element
                for j in range(length - 1, i - 1, -1):
                    # Swap the elements arr[i - 1] and arr[j] only if arr[j] is
                    # smaller than arr[i - 1] and different from arr[j - 1]
                    if arr[j] < arr[i - 1] and arr[j] != arr[j - 1]:
                        arr[i - 1], arr[j] = arr[j], arr[i - 1]  # Perform the swap
                        return arr  # Return the updated array

        # If no permutation could be performed that makes the original array smaller,
        # then return the original array itself
        return arr
```

Java

```java
class Solution {
    public int[] prevPermOpt1(int[] arr) {
        // Get the length of the array.
        int arrayLength = arr.length;

        // Start from the end of the array and look for the first pair where
        // the previous element is greater than the current element.
        for (int i = arrayLength - 1; i > 0; --i) {
            if (arr[i - 1] > arr[i]) {
                // We found a pair. now start from the end of the arrav aoain
                // and look for an element that is smaller than the element at
                // index i - 1 but not the same as its previous element (to avoid duplicates).
                for (int j = arrayLength - 1; j > i - 1; --j) {
                    if (arr[j] < arr[i - 1] && arr[j] != arr[j - 1]) {
                        // Swap the elements at i - 1 and j.
                        int temp = arr[i - 1];
                        arr[i - 1] = arr[j];
                        arr[j] = temp;
                        // Return the modified array.
                        return arr;
                    }
                }
            }
        }
        // If no swap was done, return the original array.
        return arr;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<int> prevPermOpt1(vector<int>& arr) {
        int length = arr.size();
        // Start from the end of the array and move backwards.
        for (int i = length - 1; i > 0; --i) {
            // If the current element is less than its previous one,
            // a swap is possible to get the previous permutation.
            if (arr[i - 1] > arr[i]) {
                // Look for the largest element which is smaller than arr[i - 1]
                // starting from the end of the array.
                for (int j = length - 1; j > i - 1; --j) {
                    // Ensure that we get the largest element which is not equal to
                    // its previous element to handle duplicates.
                    // This ensures the previous permutation is the largest one that is smaller than the current.
                    if (arr[j] < arr[i - 1] && arr[j] != arr[j - 1]) {
                        // Swap the found element with arr[i - 1] to get the correct previous permutation.
                        swap(arr[i - 1], arr[j]);
                        // Return the modified array as the result, no further action needed.
                        return arr;
                    }
                }
            }
        }
        // If no previous permutation is possible (array is sorted in increasing order),
        // return the unchanged array.
        return arr;
    }
};
```

TypeScript

```typescript
function prevPermOpt1(arr: number[]): number[] {
    // Calculate the length of the array just once to improve performance
    const length = arr.length;

    // Start looking for the element to swap from the end of the array
    for (let i = length - 1; i > 0; --i) {
        // Check if the current element is less than its previous element, indicating a swap is possible
        if (arr[i - 1] > arr[i]) {

            // Iterate over the array from the end to the current element
            for (let j = length - 1; j > i - 1; --j) {
                // Find the rightmost element that is smaller than the element at i - 1 and not a duplicate
                if (arr[j] < arr[i - 1] && arr[j] !== arr[j - 1]) {

                    // Perform the swap by exchanging values of arr[i - 1] and arr[j]
                    let temp = arr[i - 1];
                    arr[i - 1] = arr[j];
                    arr[j] = temp;

                    // Return the modified array as no further swaps are needed
                    return arr;
                }
            }
        }
    }

    // If no suitable previous permutation is found, return the original array
    return arr;
}
```

```python
class Solution:
    def prevPermOpt1(self, arr):
        # Length of the array
        length = len(arr)

        # Loop backwards through the array starting from the second last element
        for i in range(length - 1, 0, -1):
            # Check if the current element is greater than its following element
            # meaning a smaller permutation is possible
            if arr[i - 1] > arr[i]:
                # Find the element to swap with, which is the rightmost element
                # that is smaller than arr[i - 1] and not a duplicate of its previous element
                for j in range(length - 1, i - 1, -1):
                    # Swap the elements arr[i - 1] and arr[j] only if arr[j] is
                    # smaller than arr[i - 1] and different from arr[j - 1]
                    if arr[j] < arr[i - 1] and arr[j] != arr[j - 1]:
                        arr[i - 1], arr[j] = arr[j], arr[i - 1]  # Perform the swap
                        return arr  # Return the updated array

        # If no permutation could be performed that makes the original array smaller,
        # then return the original array itself
        return arr
```

Time and Space Complexity

The time complexity of the provided code is `O(n^2)` in the worst case. This complexity arises from the fact that we have a nested loop where the outer loop runs backwards through the list `arr` starting from the second last element to the first, and the inner loop also runs backwards, starting from the last element until it finds an element less than `arr[i - 1]`. The worst-case scenario occurs when we have to iterate over the entire array for the inner loop for each element in the outer loop.

The space complexity of the code is `O(1)` as we are only using a constant amount of extra space. The swapping of elements is done in-place and does not require any additional data structures that are dependent on the input size.