769. Max Chunks To Make Sorted

Medium Stack Greedy Array Sorting Monotonic Stack

Problem Description

You are given an integer array called `arr`, which contains `n` elements. Each element in the array is unique and ranges from 0 to `n - 1`, essentially representing a permutation of these numbers. The goal is to divide this array into several non-overlapping "chunks" (subarrays), sort each of these chunks individually, and then concatenate them back together to form a single sorted array. The task is to find the maximum number of chunks that can be made such that when they are sorted individually and then concatenated, the result is a sorted version of the original array.

Intuition

The intuition behind the solution is based on the property of permutations and how the original array can be split into the maximum number of chunks. Since the array is a permutation of numbers from 0 to `n - 1`, if all the chunks are individually sorted, then regardless of where the chunks are split, the concatenated chunks will produce a sorted array. To maximize the number of chunks, we need to exploit the fact that if the maximum value in a chunk is equal to its last index, then this chunk can be independently sorted without affecting the sorting of the entire array.

For example, consider we are at index `i` in the array, and the maximum value encountered so far is `mx`. If `i` and `mx` are equal, it means all numbers from 0 to `i` are included in this chunk, so this chunk can be independently sorted. We increment our chunk count (`ans`) everytime such a condition occurs. The process is repeated as we iterate through the array, resulting in `ans` incrementing each time a self-contained chunk (where its maximum value matches its last index) is found. This logic ensures we are splitting the array into the largest possible number of valid chunks that, when sorted individually, will form the overall sorted array.

This approach's correctness lies in the fact that for every chunk ending at index `i`, if the maximum value in that chunk is also `i`, sorting that chunk will simply place all elements from 0 to `i` (which are already within the chunk) in their correct sorted position. No other elements outside this chunk are affected, guaranteeing that each such chunk contributes to the overall sorted array.

Solution Approach

The impementation of the solution is straightforward. The steps can be broken down as follows:

1. We initialize two variables, `mx` and `ans`. `mx` will keep track of the maximum value encountered so far as we iterate through the array, and `ans` will count the number of chunks.
2. We use a `for` loop to iterate through each element represented by `i` and its value `v` in the `arr`.
3. For every element `v` in the array, we update `mx` to be the maximum of `mx` and `v`, using the expression `max(mx, v)`. This ensures that `mx` reflects the highest number among all the elements we've encountered up to the current position.
4. We check if the current index `i` is equal to the maximum value `mx` we've found so far. If it is, it means we've found a valid chunk — all integers from 0 up to the current index `i` must be contained within this chunk, as the permutation contains every integer from 0 to `n - 1` exactly once. Therefore, we can independently sort this chunk without affecting the overall sorting of the array. When such a condition is met, we increment the `ans` by 1, indicating we can form one more chunk.
5. We continue this process for all elements in the `arr`.
6. After the loop completes, `ans` contains the largest number of chunks we can form such that individually sorting these chunks and then concatenating them results in a sorted array.

No additional data structures are needed, and the pattern used is simple iteration with a check for the current index against the current maximum value. This is an efficient solution as it achieves the task with a time complexity of O(n) where n is the length of `arr`, as it requires only a single pass through the array. The space complexity is also O(1) since we're not using any additional storage that scales with the input size.

Here is the implemented function:

```
class Solution:
    def maxChunksToSorted(self, arr: List[int]) -> int:
        mx = ans = 0
        for i, v in enumerate(arr):
            mx = max(mx, v)
            if i == mx:
                ans += 1
        return ans
```

This function takes the `arr` as input and returns the maximum number of chunks `ans`.

Example Walkthrough

Let's illustrate the solution approach using a small example array `arr`: `[1, 0, 2, 4, 3]`

1. Initially, we set `mx` and `ans` to 0. `mx` will track the maximum number we encounter, and `ans` will count the chunks.
2. We begin iterating through `arr` with a `for` loop:
   - At `i` = 0, `v` = `arr[0]` = 1; we update `mx` = `max(0, 1)` so `mx` becomes 1. Since `i` is not equal to `mx`, we do not increment `ans`.
   - At `i` = 1, `v` = `arr[1]` = 0; we update `mx` = `max(1, 0)` so `mx` stays 1. Now, `i` is equal to `mx`, which means all numbers from 0 to 1 are within this chunk. We increment `ans` to 1.
   - At `i` = 2, `v` = `arr[2]` = 2; we update `mx` = `max(1, 2)` so `mx` becomes 2. Again, `i` equals `mx`, indicating another chunk. We increment `ans` to 2.
   - At `i` = 3, `v` = `arr[3]` = 4; we update `mx` = `max(2, 4)` so `mx` becomes 4. `i` is not equal to `mx`, we do not increment `ans`.
   - At `i` = 4, `v` = `arr[4]` = 3; we update `mx` = `max(4, 3)` so `mx` stays 4. Now, `i` equals `mx`, indicating another valid chunk. We increment `ans` to 3.
3. Having completed the loop, we find that the array can be split into a maximum of `3` chunks: `[1, 0]`, `[2]`, and `[4, 3]`. When we sort these subarrays independently and then concatenate them, we get the sorted array: `[0, 1, 2, 3, 4]`.

The final answer `ans` is 3, which signifies there are 3 chunks that can be individually sorted to result in a fully sorted array. This is confirmed by our walkthrough with the example array.

Solution Implementation

Python

```
from typing import List

class Solution:
    def maxChunksToSorted(self, arr: List[int]) -> int:
        # Initialize the maximum value found so far and the answer counter
        max_value = 0
        chunks_count = 0

        # Enumerate over the array to get both index and value
        for index, value in enumerate(arr):
            # Update the maximum value found in the array thus far
            max_value = max(max_value, value)

            # If the current index equals the maximum value encountered,
            # it means we can form a chunk that, when sorted independently,
            # would still maintain the overall order when merged with adjacent chunks.
            if index == max_value:
                # Thus, we increase the chunks count by 1
                chunks_count += 1

        # Return the total number of chunks we can split the array into
        return chunks_count
```

Java

```
class Solution {
    public int maxChunksToSorted(int[] arr) {
        int chunkCount = 0; // Initialize the count of chunks
        int maxSoFar = 0; // Initialize the maximum value found so far in the array

        // Iterate through the array
        for (int index = 0; index < arr.length; ++index) {
            // Update the maximum value seen so far
            maxSoFar = Math.max(maxSoFar, arr[index]);

            // If the current index is equal to the maximum value encountered,
            // it means all values before this index are smaller or equal to 'index'
            // and this position is a valid chunk boundary
            if (index == maxSoFar) {
                // Increment the count of chunks
                ++chunkCount;
            }
        }

        return chunkCount; // Return the total number of chunks
    }
}
```

C++

```
#include <vector>
#include <algorithm> // For use of the max function

class Solution {
public:
    int maxChunksToSorted(vector<int>& arr) {
        int chunkCount = 0; // Variable to count chunks
        int maxElement = 0; // Variable to store the maximum element found so far

        // Iterate through the vector elements
        for (int index = 0; index < arr.size(); ++index) {
            maxElement = std::max(maxElement, arr[index]); // Update maxElement with the highest between maxElement and the current e

            // If the maximum element we've found so far is equal to the index,
            // it means all previous elements are s index and can form a chunk
            if (index == maxElement) {
                ++chunkCount; // Increment the chunk count as we can make a new chunk
            }
        }

        return chunkCount; // Return the total number of chunks
    }
};
```

TypeScript

```
// Function to determine the maximum number of chunks that the array can be
// split into so that after sorting each chunk individually and then concatenating
// them back in order, the resultant array is sorted.
function maxChunksToSorted(arr: number[]): number {
    const n: number = arr.length; // The length of the input array.
    let answer: number = 0; // The number of chunks that the input array can be split into.
    let currentMax: number = 0; // The maximum value found so far in the array up to the current index.

    // Iterate through the array to find chunks.
    for (let i: number = 0; i < n; i++) {
        // Update the current maximum value found.
        currentMax = Math.max(arr[i], currentMax);

        // If the current maximum is equal to the index, a sorted chunk is found.
        // This is based on the property that in a sorted array of distinct numbers ranging from 0 to n-1,
        // the number at index i should be i itself, if the chunks before are correctly placed.
        if (currentMax === i) {
            answer++; // Increment the answer as we can split here.
        }
    }
    // Return the number of sorted chunks possible.
    return answer;
}
```

```
from typing import List

class Solution:
    def maxChunksToSorted(self, arr: List[int]) -> int:
        # Initialize the maximum value found so far and the answer counter
        max_value = 0
        chunks_count = 0

        # Enumerate over the array to get both index and value
        for index, value in enumerate(arr):
            # Update the maximum value found in the array thus far
            max_value = max(max_value, value)

            # If the current index equals the maximum value encountered,
            # it means we can form a chunk that, when sorted independently,
            # would still maintain the overall order when merged with adjacent chunks.
            if index == max_value:
                # Thus, we increase the chunks count by 1
                chunks_count += 1

        # Return the total number of chunks we can split the array into
        return chunks_count
```

Time and Space Complexity

The given code snippet is designed to find the maximum number of chunks a given array can be divided into so that, when each chunk is sorted individually, the entire array is sorted.

Time Complexity:

The time complexity of the function is determined by the for loop iterating over each element of the array. Since there is only one loop in the function that goes through the array of `n` elements once, the time complexity is O(`n`), where `n` is the size of the array.

Space Complexity:

The space complexity is determined by the amount of additional memory used by the algorithm. Here, only a fixed number of variables `mx` and `ans` are used regardless of the input size, which means the space complexity is O(`1`) as they use constant space.