768. Max Chunks To Make Sorted II

Hard Stack Greedy Array Sorting Monotonic Stack Leetcode Link

Problem Description

The given LeetCode problem focuses on an array partitioning and sorting strategy. The objective is to split the input integer array `arr` into contiguous chunks, sort each chunk individually, and then concatenate the chunks together. The goal is for the concatenated result to produce the same sequence as sorting the entire original array at once. The task is to determine the maximum number of such chunks we can create. A successful solution to this problem would allow us to find the most efficient way to break down an array into independently sortable pieces that, when combined, yield a fully sorted array.

Intuition

The intuition behind the solution lies in leveraging the idea of a monotonically increasing stack to determine the number of chunks. Since each chunk, once sorted, should be a part of the completely sorted array, we can track the maximum element of the current chunk we are forming. Each element in the stack represents the maximum of a chunk.

As we iterate through the array `arr`, we examine each element. If the current element is greater than or equal to the element at the top of the stack (which means it is greater than all elements in the current chunk), it can form a new chunk or be a part of the current chunk without breaking the sorting order after concatenation.

However, if the current element is smaller than the top of the stack, it belongs to one of the previous chunks. This is where we start popping from the stack while the top of the stack is greater than the current element, effectively merging those chunks. Once we finished popping from the stack (have found a chunk where the current element could fit), we push the maximum of the last popped chunk (the largest element before the merge) back onto the stack, ensuring that the sorting condition holds true for the updated chunk.

The number of elements left in the stack at the end of this process corresponds to the maximum number of chunks we can make since each stack element represents the max value of the individual sorted chunks.

Solution Approach

The solution makes use of a stack data structure to help track the largest number we can form within a chunk. Here's a step-by-step explanation of how the implementation tackles the problem:

1. We initialize an empty list `stk` that we will use as our stack. The stack is used to maintain the maximum elements of the chunks we form as we iterate through the array.

2. As we iterate over each value `v` in `arr`, we have two conditions to consider: a. If the stack is empty or the current value `v` is greater than or equal to the maximum value at the top of the stack (`stk[-1]`), we can push `v` onto the stack. This signifies either the start of a new chunk or that `v` can be a part of the current chunk without disrupting the sorted order when concatenated.

   b. If the current value `v` is less than the maximum value at the top of the stack, this indicates that `v` belongs to a previous chunk, and we need to merge some chunks to maintain the sorted order upon concatenation. We pop values from the stack until we find a value that is not greater than `v`. During this process, we keep track of the maximum value that we pop by assigning it to `mx`. This ensures that despite merging, the maximum value maintains its position to preserve the sorting requirement.

3. After the popping condition ends, we push the value `mx` back onto the stack. Now, `mx` represents the maximum value of the newly formed merged chunk.

4. Once we have finished iterating through the array, the stack will contain a certain number of elements. Each element corresponds to a chunk with its maximum value. Therefore, the length of the final stack represents the maximum number of chunks we are able to make such that when these chunks are sorted individually and concatenated, they form a sorted list equivalent to sorting the entire `arr`.

5. The function returns `len(stk)`, the number of individual chunks that can be created.

This approach effectively takes advantage of the stack data structure to dynamically manage the chunks during the array traversal, merging chunks as necessary to ensure that the final concatenation of sorted chunks results in a list sorted in non-decreasing order.

Example Walkthrough

Let's consider a small example to illustrate the solution approach:

Suppose we have the following array `arr`: `[2, 4, 1, 6, 5, 9, 7]`.

Here's how the solution algorithm would process this array:

1. Initialize an empty stack `stk`.
2. Iterate over the array `arr`. Position 0: `v = 2`. Stack is empty, so push `v` onto the stack. Stack now: `[2]`.
3. Position 1: `v = 4`. It's greater than the top of the stack, so push `v` onto the stack. Stack now: `[2, 4]`.
4. Position 2: `v = 1`. It's smaller than the top of the stack. Start popping from the stack until the top is not greater than `v`. We pop `4` and `2` from the stack, and the maximum value `mx` of popped elements is `4`. Stack is empty now, push the `mx` back. Stack now: `[4]`.
5. Position 3: `v = 6`. It's greater than the top of the stack, so push `v` onto the stack. Stack now: `[4, 6]`.
6. Position 4: `v = 5`. It's smaller than the top of the stack. Start popping from the stack. Pop `6` and `4`. The new `mx` is `6`. The stack is empty, so we push `mx` back. Stack now: `[6]`.
7. Position 5: `v = 9`. It's greater than the top of the stack, so push `v` onto the stack. Stack now: `[6, 9]`.
8. Position 6: `v = 7`. It's smaller than the top of the stack. Start popping from the stack. Pop `9`, the new `mx` is `9`. Push `mx` back onto the stack. Stack now: `[9]`.
9. Finish iterating over the array. The stack has executed multiple merges and now it has one element, which represents the number of chunks.

Each pass through the array adjusted the individual chunks in a way that they could be merged back together into a fully sorted array. At the end of this process, the stack size (which is now `1`) represents the maximum number of chunks that when sorted individually and concatenated form the sorted array.

Hence, for the given example array, the maximum number of chunks that we can form is `1`.

Python Solution

```python
from typing import List

class Solution:
    def maxChunksToSorted(self, arr: List[int]) -> int:
        # Initialize an empty stack to keep track of the current maximum in each chunk
        stack = []

        # Iterate through each value in the array
        for value in arr:
            # If the stack is empty or the current value is greater than or equal
            # to the top of the stack, this value can start a new chunk.
            if not stack or value >= stack[-1]:
                stack.append(value)
            else:
                # If the current value is smaller than the top of the stack, we need
                # to merge this value into the previous chunk. So we remove elements
                # from the stack until we find a value smaller than the current one.
                # This ensures all earlier elements are part of a sorted subarray.
                max_in_chunk = stack.pop()
                while stack and stack[-1] > value:
                    stack.pop()

                # After merging, we push the maximum value of the merged chunk back onto
                # the stack. This represents the maximum value of the new chunk after merging.
                stack.append(max_in_chunk)

        # The length of the stack represents the number of chunks since each stack element
        # corresponds to the maximum value of a sorted chunk.
        return len(stack)

```

Java Solution

```java
class Solution {
    public int maxChunksToSorted(int[] arr) {
        Deque<Integer> stack = new ArrayDeque<>(); // Use a deque as a stack to store values

        for (int value : arr) { // Iterate through each value in the array
            if (stack.isEmpty() || stack.peek() <= value) { // If stack is empty or the top is less than or equal to current value
                stack.push(value); // Push the current value onto the stack
            } else {
                // If the current value is less, this means a new chunk must be formed
                int maxInChunk = stack.pop(); // Pop the top value, which is the maximum in the current chunk
                // Pop all values in the chunk which are greater than current value
                while (!stack.isEmpty() && stack.peek() > value) {
                    stack.pop();
                }
                // Push the max value back to represent the current chunk
                stack.push(maxInChunk);
            }
        }

        return stack.size(); // The number of chunks is the size of the stack
    }
}

```

C++ Solution

```cpp
#include<vector>
#include<stack>
using namespace std;

class Solution {
public:
    int maxChunksToSorted(vector<int>& arr) {
        stack<int> valueStack; // Use a stack to store values

        // Iterate over each element in the array
        for (int value : arr) {
            // If the stack is empty or the current value is greater than or equal
            // to the top of the stack, push the current value onto the stack.
            if (valueStack.empty() || valueStack.top() <= value) {
                valueStack.push(value);
            } else {
                // If the current value is less than the top of the stack,
                // then we need to merge the current chunk with the previous chunks
                // until the value at the top of the stack is no longer greater
                // than the current value.

                // Store the maximum value in the current chunk
                int maxValueInChunk = valueStack.top();
                valueStack.pop(); // Remove the top element as we're going to merge chunks

                // Keep removing elements from the stack until the top is not greater
                // than the current value.
                while (!valueStack.empty() && valueStack.top() > value) {
                    valueStack.pop();
                }

                // Push the maximum value of the merged chunks back onto the stack
                // because it represents the maximum value up to the current point,
                // and chunks need to be sorted independently.
                valueStack.push(maxValueInChunk);
            }
        }

        // The size of the stack represents the maximum number of chunks that are sorted
        // when taken independently. Hence, we can return the stack size directly.
        return valueStack.size();
    }
};

```

Typescript Solution

```typescript
function maxChunksToSorted(arr: number[]): number {
    // Initialize an empty stack to keep track of the max value of each chunk
    const stack: number[] = [];

    // Iterate through each number in the input array
    for (const num of arr) {
        // If the current number is smaller than the last number on the stack,
        // we need to merge the current chunk with the previous one.
        if (stack.length > 0 && num < stack[stack.length - 1]) {
            // The current chunk's max value is stored.
            const currentMax = stack.pop();

            // Keep removing elements from the stack until we find a value
            // not greater than the current number to ensure chunks are sorted.
            while (stack.length > 0 && num < stack[stack.length - 1]) {
                stack.pop();
            }

            // Push the max value of the merged chunk back onto the stack.
            stack.push(currentMax);
        } else {
            // As the current number is not smaller than the last chunk's max,
            // push the number onto the stack to start a new chunk.
            stack.push(num);
        }
    }

    // The number of chunks is the stack's length after merging.
    return stack.length;
}

```

Time and Space Complexity

The given code snippet is designed to solve the problem of finding the maximum number of chunks in which an array can be split such that when the individual chunks are sorted, the entire array is sorted. This is achieved by maintaining a stack, `stk`, and iterating over the elements of the array, `arr`.

Time Complexity

The time complexity of the code is `O(n)`. This is because there is a single for-loop iterating over the elements of the array `arr`. Although there is a while-loop inside the for-loop, each element from the array is pushed onto the stack at most once and popped from the stack at most once. This means that even with the inner while-loop, each operation on an element (either push or pop) is done only a constant number of times. Overall, the number of operations is proportional to the number of elements `n`.

Space Complexity

The space complexity of the code is `O(n)`. In the worst case, the stack `stk` could potentially store all the elements of the array if the array is in ascending order. As a result, the amount of space used is proportional to the number of elements in the input array, `n`.