<https://leetcode.com/problems/max-chunks-to-make-sorted/>

You are given an integer array arr of length n that represents a permutation of the integers in the range [0, n - 1].

We split arr into some number of **chunks** (i.e., partitions), and individually sort each chunk. After concatenating them, the result should equal the sorted array.

Return *the largest number of chunks we can make to sort the array*.

**Example 1:**

Input: arr = [4,3,2,1,0]

Output: 1

Explanation:

Splitting into two or more chunks will not return the required result.

For example, splitting into [4, 3], [2, 1, 0] will result in [3, 4, 0, 1, 2], which isn't sorted.

**Example 2:**

Input: arr = [1,0,2,3,4]

Output: 4

Explanation:

We can split into two chunks, such as [1, 0], [2, 3, 4].

However, splitting into [1, 0], [2], [3], [4] is the highest number of chunks possible.

**Constraints:**

n == arr.length

1 <= n <= 10

0 <= arr[i] < n

All the elements of arr are **unique**.

**Attempt 1: 2023-04-01**

**Solution 1: Monotonic Increasing Stack (120 min)**

class Solution {

    public int maxChunksToSorted(int[] arr) {

        int len = arr.length;

        Stack<Integer> stack = new Stack<Integer>();

        int largest = Integer.MIN\_VALUE;

        for(int num : arr) {

            largest = num;

            while(!stack.isEmpty() && stack.peek() > num) {

                int cur = stack.pop();

                largest = Math.max(largest, cur);

            }

            // Make sure we push the 'largest' not the 'num'

            stack.push(largest);

        }

        return stack.size();

    }

}

**Solution 2: Left max and right min (10 min)**

class Solution {

    public int maxChunksToSorted(int[] arr) {

        int len = arr.length;

        int[] maxLeft = new int[len];

        int[] minRight = new int[len];

        maxLeft[0] = arr[0];

        for(int i = 1; i < len; i++) {

            maxLeft[i] = Math.max(maxLeft[i - 1], arr[i]);

        }

        minRight[len - 1] = arr[len - 1];

        for(int i = len - 2; i >= 0; i--) {

            minRight[i] = Math.min(minRight[i + 1], arr[i]);

        }

        // Default as '1' because we can always sort array itself without split

        int result = 1;

        for(int i = 0; i < len - 1; i++) {

            // When you are at index i, you should compare max(0,..., i)

            // with min(i + 1, ..., len - 1)

            if(maxLeft[i] <= minRight[i + 1]) {

                result++;

            }

        }

        return result;

    }

}

**Solution 3: Greedy: compare maximum value till current position with index (60 min)**

class Solution {

    public int maxChunksToSorted(int[] arr) {

        int len = arr.length;

        int[] max = new int[len];

        max[0] = arr[0];

        for(int i = 1; i < len; i++) {

            max[i] = Math.max(max[i - 1], arr[i]);

        }

        int count = 0;

        for(int i = 0; i < len; i++) {

            if(max[i] == i) {

                count++;

            }

        }

        return count;

    }

}

===================================================

class Solution {

    public int maxChunksToSorted(int[] arr) {

        int len = arr.length;

        int max = 0;

        int count = 0;

        for(int i = 0; i < len; i++) {

            max = Math.max(max, arr[i]);

            if(max == i) {

                count++;

            }

        }

        return count;

    }

}

**This question has one difference than L768.Max Chunks To Make Sorted II: Its not a random array, its an integer array arr of length**

**n that represents a permutation of the integers in the range [0, n - 1].which gives us another thought to resolve the issue**

**Refer to**

<https://algo.monster/liteproblems/769>

**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:

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.

We use a for loop to iterate through each element represented by i and its value v in the arr.

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.

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.

We continue this process for all elements in the arr.

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]

Initially, we set mx and ans to 0. mx will track the maximum number we encounter, and ans will count the chunks.

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.

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**

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

}

}

**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.

**Refer to**

<https://leetcode.com/problems/max-chunks-to-make-sorted/solutions/113528/simple-java-o-n-solution-with-detailed-explanation>

The basic idea is to use max[] array to keep track of the max value until the current position, and compare it to the sorted array (indexes from 0 to arr.length - 1). If the max[i] equals the element at index i in the sorted array, then the final count++.

Update: As @AF8EJFE pointed out, the numbers range from 0 to arr.length - 1. So there is no need to sort the arr, we can simply use the index for comparison. Now this solution is even more straightforward with O(n) time complelxity.

For example,

original: 0, 2, 1, 4, 3, 5, 7, 6

max:      0, 2, 2, 4, 4, 5, 7, 7

sorted:  0, 1, 2, 3, 4, 5, 6, 7

index:    0, 1, 2, 3, 4, 5, 6, 7

The chunks are: 0 | 2, 1 | 4, 3 | 5 | 7, 6

    public int maxChunksToSorted(int[] arr) {

        if (arr == null || arr.length == 0) return 0;

        int[] max = new int[arr.length];

        max[0] = arr[0];

        for (int i = 1; i < arr.length; i++) {

            max[i] = Math.max(max[i - 1], arr[i]);

        }

        int count = 0;

        for (int i = 0; i < arr.length; i++) {

            if (max[i] == i) {

                count++;

            }

        }

        return count;

    }

Update2:

The code can be further simplified as follows.

    public int maxChunksToSorted(int[] arr) {

        if (arr == null || arr.length == 0) return 0;

        int count = 0, max = 0;

        for (int i = 0; i < arr.length; i++) {

            max = Math.max(max, arr[i]);

            if (max == i) {

                count++;

            }

        }

        return count;

    }

**Refer to**

<https://leetcode.com/problems/max-chunks-to-make-sorted/solutions/113528/simple-java-o-n-solution-with-detailed-explanation/comments/181573>

A more explicit explanation to this algorithm, or another perspective to solve the problem is to cut the array into a serious of continuous sequences, so that each chunk, after being sorted, could be directly used as a brick to construct the sorted array.

# Example

array: [1, 0, 2, 4, 3]

cut :  [1, 0, 2 | 4, 3]

# [1, 0, 2] and [4, 3] are both continuous sequences.

The key to understand this algorithms lies in the fact that when max[index] == index, all the numbers before index must be smaller than max[index] (also index), so they make up of a continuous unordered sequence, i.e {0,1,..., index}. This is because numbers in array only vary in range [0, 1, ..., arr.length - 1], so the most numbers you can find that are smaller than a certain number, say arr[k], would be arr[k] - 1, i.e [0, 1, ..., arr[k] - 1]. So when arr[k] is the max number in [arr[0], arr[1], ..., arr[k]], all the k - 1 numbers before it can only lies in [0, 1, ..., arr[k] - 1], so they made up of a continuous sequence. (You can also prove it using contradiction, which may be easier to understand)

array: [1, 0, 2, 4, 3]

cut :  [1, 0, 2 | 4, 3]

max:  [1, 1, 2 | 4, 4]

index: [0, 1, 2, 3, 4]

# max[2] == 2, and for the first three numbers, all the numbers in [0, 1, 2, 3, 4] that are smaller than 2 lie in [0, 1],

# so [1, 0, 2] makes a continuous unordered sequence, which can be used to construct [0, 1, 2, 3, 4] as a brick after been sorted into [0, 1, 2]

**Refer to**

[L768.Max Chunks To Make Sorted II (Ref.L769,L739)](note://BB68399B4CC048E6A2560D993E38F665)

[L739.Daily Temperatures](note://6FB3A11E2B7746DCA411301F941DD948)