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

You are given an integer array arr.

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 = [5,4,3,2,1]

Output: 1

Explanation:

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

For example, splitting into [5, 4], [3, 2, 1] will result in [4, 5, 1, 2, 3], which isn't sorted.

**Example 2:**

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

Output: 4

Explanation:

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

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

**Constraints:**

* 1 <= arr.length <= 2000
* 0 <= arr[i] <= 108

**Attempt 1: 2023-03-27**

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

class Solution {

public int maxChunksToSorted(int[] arr) {

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);

}

stack.push(largest);

}

return stack.size();

}

}

**Refer to**

<https://leetcode.com/problems/max-chunks-to-make-sorted-ii/solutions/595713/monotonic-stack-solution-with-detailed-explanation/>

If you are not familiar with **monotonic stack**, please refer to [this](https://leetcode.com/problems/sum-of-subarray-minimums/discuss/178876/stack-solution-with-very-detailed-explanation-step-by-step) first.

Understanding how **monotonic stack** works (build an increasing or decreasing stack based on the input) and what it can do is relatively easy. Unfortunately the hardest part is solution based on monotonic stack is sometimes not intuitive.

Here's the thought based on this question.

First question, what is the largest number of chunks for an array given the length as n? it's simple, just n, namely every single element makes up the single-element chunk. For a given array, the largest number of chunks happens when it is increasing and every element is already at the right place.[0, 1, 3, 4] => [0, 1], [3], [4]

So as long as we are seeing an array that has increasing order, we just keep counting the number of elements in the array. (If you are familiar with **monotonic stack**, have you already smelled that it could be something useful to solve this question?)

Let's keeping counting, until we see the next element 2. So now the input is [0, 1, 3, 4, 2]Well what does a suddenly jumped out smaller number mean ?? It means, 2 is not at the right place in the sorted array (we can see 2 actually should sit at the 3rd place in the array right?), **and** **SOMETHING ELSE** before 2 is also at the wrong place, which we didn't realize until we see 2. So we need to look back.

(While, another hint, we are expecting an increasing order array and we have to look backwards when the order is broken. Does this smell like **monotonic stack** again?)

So what exactly are we looking for when we are looking back? We want to find the correct place that 2 should sit at, which means: **We are looking for the largest number that is smaller than 2 in an ascending array**. Why we care about where 2 should be at? because anything between the current position and the should-be position for 2 is not at the correct position that it should be (otherwise how could the poor 2's potion being occupied?). If we can find the elements (including 2) that are not at the right position, they need to be put in a chunk, and sorting them inside the chunk is the only way to put them in right place.

So in this example: [0, 1, 3, 4, 2], we found 1 is the largest number smaller than 2 given our monotonic stack [0, 1, 3, 4]. So everything between 3 and 2 (inclusive) is messed up and they [3, 4, 2] has to be put in a chunk, so we can sort them back to [2, 3, 4] and put them back to the right place.

Normally, we rebuild the monotonic stack from [0,1, 3, 4] to [0, 1, 2] by popping out the elements larger than 2 and push 2 back to the queue. However this doesn't work for this question.

Let's think about what exactly the meaning we want it to be for each element in the increasing stack.

We want it to mean: **The largest value in each chunk when the chunk cannot be partitioned to smaller ones**. So the number of elements in the stack is exactly the number we are looking for.

So now the stack became [0, 1, 4] instead of [0, 1, 2] and each 0 => [0], 1 => [1], 4 => [3, 4, 2].

class Solution:

def maxChunksToSorted(self, arr: List[int]) -> int:

stack = []

for num in arr:

largest = num

while stack and stack[-1] > num:

largest = max(largest, stack.pop())

stack.append(largest)

return len(stack)

**Solution 2: Left max and right min (120 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;

}

}

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

Don't need two arrays, use a max variable to track max of left (including i)

class Solution {

public int maxChunksToSorted(int[] arr) {

int len = arr.length;

//int[] maxLeft = new int[len];

int maxLeft = Integer.MIN\_VALUE;

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)

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

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

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

result++;

}

}

return result;

}

}

Space Complexity: O(n)

Time Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/max-chunks-to-make-sorted-ii/solutions/113462/java-solution-left-max-and-right-min/>

<https://leetcode.com/problems/max-chunks-to-make-sorted-ii/solutions/113462/java-solution-left-max-and-right-min/comments/1165311>

Algorithm: Iterate through the array, each time all elements to the left are smaller (or equal) to all elements to the right, there is a new chunk.

Use two arrays to store the left max and right min to achieve O(n) time complexity. Space complexity is O(n) too.

This algorithm can be used to solve ver1 too.

When making the maximum number of chunks, we can make the valid external chunks and merge two chunks only if all the elements on the left are smaller or equal to the elements on right. For example, [1,3,2] and [6,4,5] -> if we sort them individually we get [1,2,3] and [4,5,6].. If we merge them, [1,2,3,4,5,6] boom the merged array is sorted. Now take the example of [1,4,2] and [6,3,5]. Sorting individually, we get -> [1,2,4] and [3,5,6] and when we concatenate them -> [1,2,4,3,5,6] which is not sorted result we get.

Important part1: **An observation we can make here is, the elements on the left all have to be smaller or equal to the smallest elements on right to make chunks.** Suppose left\_max stores maximum num so far until the current index and includes current index. right\_max stores minimum so far from the right until but excluding current number.

Important part2: So from here: **if left\_max[idx] <= right\_min[idx] satisfies the condition, we know that this is a feasible point for making chunks i.e splitting the array.** The question is asking to count the maximum number of chunks we can make, so every time we counter such a condition met, we increment the count of chunk. The minimum number of chunks will always be 1 no matter what since we can take just one giant chunk and sort it to make it sorted.

class Solution {

public int maxChunksToSorted(int[] arr) {

int n = arr.length;

int[] maxOfLeft = new int[n];

int[] minOfRight = new int[n];

maxOfLeft[0] = arr[0];

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

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

}

minOfRight[n - 1] = arr[n - 1];

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

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

}

int res = 0;

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

if (maxOfLeft[i] <= minOfRight[i + 1]) res++;

}

return res + 1;

}

}

**Refer to**

<https://leetcode.com/problems/max-chunks-to-make-sorted-ii/solutions/113462/java-solution-left-max-and-right-min/comments/182522>

You actually don't need two arrays. You can use a max variable to track max of left (including i)

public int maxChunksToSorted(int[] arr) {

int n = arr.length;

int[] minOfRight = new int[n];

minOfRight[n - 1] = arr[n - 1];

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

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

}

int res = 0;

int max = Integer.MIN\_VALUE;

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

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

if (max <= minOfRight[i + 1]) res++;

}

return res + 1;

}