<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: 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://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]