Max Chunks To Make Sorted

Given an array `arr` that is a permutation of `[0, 1, ..., arr.length - 1]`, we split the array into some number of "chunks" (partitions), and individually sort each chunk. After concatenating them, the result equals the sorted array.

What is the most number of chunks we could have made?

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 i
sn'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 possibl
e.
```

Note:

- `arr` will have length in range `[1, 10]`.
- `arr[i]` will be a permutation of `[0, 1, ..., arr.length - 1]`.

Solution 1

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 chunck.
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 ver2 too.

```
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;
    }
}
```

written by shawngao original link here

Solution 2

Similiar to 763. Partition Labels.

```
int maxChunksToSorted(vector<int>& arr) {
    for (auto i = 0, max_i = 0, ch = 0; i <= arr.size(); ++i) {
      if (i == arr.size()) return ch;
      max_i = max(max_i, arr[i]);
      if (max_i == i) ++ch;
    }
}
```

written by votrubac original link here

Solution 3

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;
    }
```

The solution to Ver2 is very similar, with only slight modification:

https://discuss.leetcode.com/topic/117855/simple-java-solution-with-explanation

written by lily4ever original link here