Arrays_4

@kbbhatt04

@August 6, 2023

Partition Array into Disjoint Intervals

Given an integer array `nums`, partition it into two (contiguous) subarrays `left` and `right` so that:

- Every element in `left` is less than or equal to every element in `right`.
- `left` and `right` are non-empty.
- `left` has the smallest possible size.

Return *the length of* `left` *after such a partitioning*.

Test cases are generated such that partitioning exists.

- Approach
    - Brute-force
        - Instead of checking whether `all(L <= R for L in left for R in right)`, for each index let's only check whether the **largest element to the left** of the current index (inclusive) is less than or equal to the **smallest element to the right** of the current index (`max(left) <= min(right)`).
        - Take 2 arrays, `left` and `right` and copy all elements of nums into both arrays
        - Iterate over `left` from beginning and modify elements such that, element[i] is maximum element observed till that index i.e. `left[i] = max(left[i], left[i-1])`
        - Iterate over `right` from end and modify elements such that, element[i] is minimum element observed till that index from the end i.e. `right[i] = min(right[i], right[i+1])`
        - Then we need to find the breakpoint i.e. the index `i` such that `left[i-1] ≤ right[i]` and return `i`

- Time Complexity: $O(3*n)$
- Space Complexity: $O(2*n)$

- Better
  - Notice, in the first approach, we iterated from `1` to `N` twice. Once to create `max_left` and once to find which index to split the array at. We can slightly optimize our approach by performing both of these steps in the same for loop. Doing so will allow us to replace the `max_left` array with a single variable that tracks the maximum value seen so far (`curr_max`).
  - Initialize `right` array same as mentioned above
  - Initialize `curr_max` as the leftmost value in nums
  - Iterate over nums from left to right and at each iteration, update `curr_max` as the maximum value seen so far. When `curr_max` is less than or equal to the minimum value to the right, then the current index is where `nums` should be split
  - Time Complexity: $O(2*n)$
  - Space Complexity: $O(n)$
- Optimal
  - Looping through each element `A[i]` we will keep track of the `max_so_far` and `disjoint` index
  - If we see a value `A[i] < max_so_far` we know that value must be in the left partition, so we update the `disjoint` location and check if we have a new `max_so_far` in the left partition
  - Once we go through the list, every element on the right side of `disjoint` is guarenteed to be larger than elements left of `disjoint`
  - Time Complexity: $O(n)$
  - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
class Solution:
    def partitionDisjoint(self, nums: List[int]) -> int:
        N = len(nums)
        max_left = [None] * N
        min_right = [None] * N

        max_left[0] = nums[0]
        min_right[-1] = nums[-1]

        for i in range(1, N):
            max_left[i] = max(max_left[i - 1], nums[i])

        for i in range(N - 2, -1, -1):
            min_right[i] = min(min_right[i + 1], nums[i])

        for i in range(1, N):
            if max_left[i - 1] <= min_right[i]:
                return i
```

```
# Python3
# Better Solution
class Solution:
    def partitionDisjoint(self, nums: List[int]) -> int:
        N = len(nums)
        min_right = [None] * N
        min_right[-1] = nums[-1]

        for i in range(N - 2, -1, -1):
            min_right[i] = min(min_right[i + 1], nums[i])

        curr_max = nums[0]
        for i in range(1, N):
            curr_max = max(curr_max, nums[i - 1])
```

```
            if curr_max <= min_right[i]:
                return i
```

```
# Python3
# Optimal Solution
class Solution:
    def partitionDisjoint(self, nums: List[int]) -> int:
        ans = 0
        left_max = max_so_far = nums[0]

        for i in range(1, len(nums)):
            max_so_far = max(max_so_far, nums[i])
            if nums[i] < left_max:
                ans = i
                left_max = max_so_far
        return ans+1
```

```
// C++
// Optimal Solution
class Solution {
public:
    int partitionDisjoint(vector<int>& nums) {
        int ans = 0;
        int left_max = nums[0];
        int max_so_far = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            max_so_far = max(max_so_far, nums[i]);
            if (nums[i] < left_max) {
                ans = i;
                left_max = max_so_far;
            }
        }
        return ans + 1;
```

```
    }
};
```

Template

- Approach
    - Brute-force
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Better
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$
    - Optimal
        -
        - Time Complexity: $O(n^3)$
        - Space Complexity: $O(1)$

```
# Python3
# Brute-force Solution
```

```
# Python3
# Better Solution
```

```
# Python3
# Optimal Solution
```

```
// C++
// Optimal Solution
```