915. Partition Array into Disjoint Intervals

Medium <u>Array</u>

Problem Description

The problem gives us an integer array called nums. We are required to split this array into two contiguous subarrays named left and right. The conditions for the split are:

1. Every element in left should be less than or equal to every element in right.

2. Both subarrays left and right should be non-empty.

3. The subarray left should be as small as possible.

The goal is to return the length of the left subarray after partitioning it according to the rules mentioned above.

For clarity: If nums is [5,0,3,8,6], one correct partition could be left = [5,0,3] and right = [8,6], and we would return 3 because left has a length of 3.

Intuition

To solve this problem, we need to find the point where we can divide the array into left and right such that every element in

Intuitively, as we traverse nums from left to right, we can keep track of the running maximum value. This value will indicate the

highest number that must be included in the left subarray to satisfy the condition that all elements in left are smaller or equal to

left is smaller or equal to the elements in right.

the elements in right.

Conversely, if we traverse the array from right to left, we can compute the running minimum. This minimum will provide us with the lowest value at each index that right can have.

minimum on the right side. This means that all values to the left of the partition point can form the left subarray, as they are guaranteed to be less than or equal to all values to the right of the partition point, which would form the right subarray.

The partition point should be at a place where the running maximum from the left side is less than or equal to the running

Solution Approach

The implementation follows a two-step process:

• An array mi is initialized to store the running minimums from the right-hand side of the given nums array. This new array will have n+1 elements, where n is the length of nums, and is initialized with inf which represents infinity, guaranteeing that all actual numbers in nums are smaller than

inf.

Step 1: Calculate Minimums From Right to Left

• We iterate over nums starting from the last element to the first (right to left), updating mi such that every mi[i] contains the smallest element

Here's a snippet of the implementation described:

for i in range(n - 1, -1, -1):

subarray that satisfies the given partitioning conditions.

- from nums[i] to nums[n-1].

 Step 2: Find Partition Index
- A variable mx is initialized to zero to keep track of the running maximum as we iterate through the array from left to right (important for determining the left subarray).
 We iterate over the nums, using enumerate() to get both the index and the value at each step.

• For each element in nums, we update mx to be the maximum of mx and the current element v. This running maximum represents the largest

element that would have to be included in the left subarray for all elements in left to be smaller or equal to any element in right. • We then check if my is less than or equal to milit at this index. If this condition holds, it signifies that all elements up to this index in

- We then check if mx is less than or equal to mi[i] at this index. If this condition holds, it signifies that all elements up to this index in nums can form the left subarray, and they are all less than or equal to any element to their right. The index i is then returned as the size of left.
- The algorithm described above uses a greedy approach to find the earliest point where the left could end, ensuring the conditions for the partition are met. By leveraging the pre-computed minimums and maintaining a running maximum, we can make this determination in a single pass over the input array after the initial setup, resulting in an efficient solution.
- class Solution:
 def partitionDisjoint(self, nums: List[int]) -> int:
 n = len(nums)
 mi = [float('inf')] * (n + 1)

mi[i] = min(nums[i], mi[i + 1])
mx = 0
for i, v in enumerate(nums, 1):
 mx = max(mx, v)
 if mx <= mi[i]:
 return i</pre>

The above python code snippet represents the full implementation of the solution approach for finding the length of the left

Step 1: Calculate Minimums From Right to Left

mi = [inf, 12, 6, 6, 0, 0, inf]

Example Walkthrough

(Note: The example is following 1-based indexes for this explanation)

We traverse nums from left to right, keeping track of the running maximum (mx).

Let's walk through a small example to illustrate the solution approach described above.

We need to initialize an array mi to store the running minimums from the right-hand side of the nums array.

Step 2: Find Partition Index

Consider the array nums = [1, 1, 1, 0, 6, 12].

Starting from the last element of nums, we fill mi as follows:

```
At each index, our mx and mi arrays will look like this:

• Index i = 1, nums[i] = 1, mx = 1, mi[i] = 12. Condition mx <= mi[i] is True.
```

Index i = 5, nums[i] = 6, mx = 6, mi[i] = 0. Condition mx <= mi[i] is False.
 Index i = 6, nums[i] = 12, mx = 12, mi[i] = inf. Condition mx <= mi[i] is True.

As we can see, the last true condition for $mx \ll mi[i]$ occurred at index i = 3.

• Index i = 2, nums[i] = 1, mx = 1, mi[i] = 6. Condition mx <= mi[i] is True.

• Index i = 3, nums[i] = 1, mx = 1, mi[i] = 6. Condition mx <= mi[i] is True.

• Index i = 4, nums[i] = 0, mx = 1, mi[i] = 0. Condition mx <= mi[i] is False.

Therefore, the smallest index i we found that satisfies the conditions is 3. This suggests our left subarray should be [1, 1, 1], which has a length of 3, and that is the answer returned by our function.

Solution Implementation

length = len(nums)

left max = 0

Python

C++

The implementation for this example would execute the partitionDisjoint method on the array and return 3. solution = Solution() assert solution.partitionDisjoint([1, 1, 1, 0, 6, 12]) == 3

Create a list to store the minimum values encountered from right to left.

Initialize each position with positive infinity for later comparison.

Update the running maximum value found in the left partition.

class Solution:
 def partitionDisjoint(self, nums: List[int]) -> int:
 # Get the length of the input array

Initialize a variable to keep track of the maximum value seen so far from left to right.

Check if the left_max value is less than or equal to the right_min at the current position.

This means all values to the left are less than or equal to values to the right, satisfying the condition.

```
# Populate the right_min list with the minimum values from right to left.
for i in range(length - 1, -1, -1):
    right_min[i] = min(nums[i], right_min[i + 1])
```

Iterate through the array to find the partition point.

right_min = [float('inf')] * (length + 1)

for i, value in enumerate(nums):

left_max = max(left_max, value)

if left_max <= right_min[i + 1]:</pre>

if (maxLeft <= minRightArray[i]) {</pre>

// Return the partition index.

return i;

return 0;

let leftMax = 0;

for (let i = 1; i <= size; ++i) {

let currentValue = nums[i - 1];

};

return i; // The partition index is i.

```
# Return the position as the partition index.
               # i + 1 is used because the left partition includes the current element at index i.
                return i + 1
       # The function will always return within the loop above, as the partition is guaranteed to exist.
Java
class Solution {
    public int partitionDisjoint(int[] nums) {
       int length = nums.length;
       int[] minRightArray = new int[length + 1]; // This will store the minimum from right to left.
       minRightArray[length] = nums[length - 1]; // Initialize the last element.
       // Fill minRightArray with the minimum values starting from the end.
       for (int i = length - 1; i >= 0; --i) {
           minRightArray[i] = Math.min(nums[i], minRightArray[i + 1]);
       int maxLeft = 0; // This will hold the maximum value in the left partition.
       // Iterate through the array to find the partition.
       for (int i = 1; i <= length; ++i) {</pre>
           int currentValue = nums[i - 1];  // Current value from nums array.
           maxLeft = Math.max(maxLeft, currentValue); // Update maxLeft with the current value if it's greater.
           // If maxLeft is less than or equal to the minimum in the right partition,
           // we have found the partition point.
```

return 0; // Default return value if no partition is found (won't occur given problem constraints).

```
#include <vector>
#include <limits>
class Solution {
public:
    int partitionDisjoint(std::vector<int>& nums) {
       // Get the size of the input array.
        int size = nums.size();
       // Create a vector to keep minimums encountered from the right side of the array.
       // Initialize every element to INT_MAX.
        std::vector<int> rightMinimums(size + 1, std::numeric_limits<int>::max());
       // Fill the rightMinimums array with the actual minimums encountered from the right.
        for (int i = size - 1; i >= 0; --i) {
            rightMinimums[i] = std::min(nums[i], rightMinimums[i + 1]);
       // Track the maximum element found so far from the left.
        int leftMax = 0;
       // Iterate to find the partition point where every element on the left is less than or equal
       // to every element on the right of the partition.
        for (int i = 1; i <= size; ++i) {
            int currentVal = nums[i - 1];
            leftMax = std::max(leftMax, currentVal);
           // If the current maximum of the left is less than or equal to the minimum of the right,
           // we found our partition point.
           if (leftMax <= rightMinimums[i]) {</pre>
```

```
function partitionDisjoint(nums: number[]): number {
    // Get the size of the input array.
    let size = nums.length;

    // Create an array to keep track of the minimums encountered from the right side of the array.
    // Initialize every element to Infinity.
    let rightMinimums: number[] = new Array(size + 1).fill(Infinity);

    // Populate the rightMinimums array with the actual minimum values encountered from the right.
    for (let i = size - 1; i >= 0; --i) {
        rightMinimums[i] = Math.min(nums[i], rightMinimums[i + 1]);
    }
```

// All elements to the left of this point are less than or equal to every element on the right.

// Variable to track the maximum element found so far from the left side.

// Iterate through the array to find the partition point.

leftMax = Math.max(leftMax, currentValue);

// Return 0 as a default (though the problem guarantees a solution will be found before this).

```
// Check if the current maximum of the left is less than or equal to the minimum on the right.
    if (leftMax <= rightMinimums[i]) {
        // If so, the current index is the correct partition index, so return it.
        return i;
    }
}

// The problem statement guarantees a solution before reaching this point.
// Return 0 as a default (non-achievable in this problem context).
return 0;
}

class Solution:
    def partitionDisjoint(self, nums: List[int]) -> int:
        # Get the length of the input array
        length = len(nums)

# Create a list to store the minimum values encountered from right to left.
# Initialize each position with positive infinity for later comparison.
        right_min = [float('inf')] * (length + 1)
```

Initialize a variable to keep track of the maximum value seen so far from left to right.

Check if the left_max value is less than or equal to the right_min at the current position.

The function will always return within the loop above, as the partition is guaranteed to exist.

Populate the right_min list with the minimum values from right to left.

Update the running maximum value found in the left partition.

```
# This means all values to the left are less than or equal to values to the right, satisfying the condition.
if left_max <= right_min[i + 1]:
    # Return the position as the partition index.
    # i + 1 is used because the left partition includes the current element at index i.
    return i + 1
```

Time and Space Complexity

left_max = 0

for i in range(length -1, -1, -1):

for i, value in enumerate(nums):

left max = max(left max, value)

right_min[i] = min(nums[i], right_min[i + 1])

Iterate through the array to find the partition point.

The time complexity of the provided code is O(n), where n is the length of the array nums. This is because there are two separate for-loops that each iterate over the array once. The first for-loop goes in reverse to fill the mi array with the minimum value encountered so far from the right side of the array. The second for-loop finds the maximum so far and compares it with values in the mi array to find the partition point. Each loop runs independently of the other and both iterate over the array once, hence maintaining a linear time complexity.

The space complexity of the code is also O(n). This is due to the additional array mi that stores the minimum values up to the current index from the right side of the array. This mi array is the same length as the input array nums, so the space complexity is directly proportional to the input size.