**Assignment Questions 18**

1. **Merge Intervals**

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

**Example 1:**

Input: intervals = [[1,3],[2,6],[8,10],[15,18]]

Output: [[1,6],[8,10],[15,18]]

Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

**Example 2:**

Input: intervals = [[1,4],[4,5]]

Output: [[1,5]]

Explanation: Intervals [1,4] and [4,5] are considered overlapping.

**Constraints:**

* 1 <= intervals.length <= 10000
* intervals[i].length == 2
* 0 <= starti <= endi <= 10000

**CODE:**

**def** merge\_intervals(intervals):

*# Sort the intervals based on the start time*

intervals**.**sort(key**=lambda** x: x[0])

*# Initialize the result list with the first interval*

merged **=** [intervals[0]]

*# Iterate through the intervals*

**for** interval **in** intervals[1:]:

*# Check if the current interval overlaps with the last interval in the result list*

**if** interval[0] **<=** merged[**-**1][1]:

*# Update the end time of the last interval if necessary*

merged[**-**1][1] **=** max(merged[**-**1][1], interval[1])

**else**:

*# Append the current interval to the result list*

merged**.**append(interval)

**return** merged

2. **Sort Colors**

Given an array nums with n objects colored red, white, or blue, sort them [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

**Example 1:**

Input: nums = [2,0,2,1,1,0]

Output: [0,0,1,1,2,2]

**Example 2:**

Input: nums = [2,0,1]

Output: [0,1,2]

**Constraints:**

* n == nums.length
* 1 <= n <= 300
* nums[i] is either 0, 1, or 2.

**CODE:**

**def** sort\_colors(nums):

low **=** 0

mid **=** 0

high **=** len(nums) **-** 1

**while** mid **<=** high:

**if** nums[mid] **==** 0:

nums[low], nums[mid] **=** nums[mid], nums[low]

low **+=** 1

mid **+=** 1

**elif** nums[mid] **==** 1:

mid **+=** 1

**else**:

nums[mid], nums[high] **=** nums[high], nums[mid]

high **-=** 1

**return** nums

3. **First Bad Version Solution**

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

**Example 1:**

Input: n = 5, bad = 4

Output: 4

Explanation:

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

**Example 2:**

Input: n = 1, bad = 1

Output: 1

**Constraints:**

* 1 <= bad <= n <= 2^31 - 1

**CODE:**

**def** isBadVersion(version):

*# Placeholder function for demonstration*

*# Replace with the actual implementation*

**return** version **>=** bad

**def** first\_bad\_version(n):

left **=** 1

right **=** n

**while** left **<** right:

mid **=** left **+** (right **-** left) **//** 2

**if** isBadVersion(mid):

right **=** mid

**else**:

left **=** mid **+** 1

**return** left

4. **Maximum Gap**

Given an integer array nums, return *the maximum difference between two successive elements in its sorted form*. If the array contains less than two elements, return 0.

You must write an algorithm that runs in linear time and uses linear extra space.

**Example 1:**

Input: nums = [3,6,9,1]

Output: 3

Explanation: The sorted form of the array is [1,3,6,9], either (3,6) or (6,9) has the maximum difference 3.

**Example 2:**

Input: nums = [10]

Output: 0

Explanation: The array contains less than 2 elements, therefore return 0.

**Constraints:**

* 1 <= nums.length <= 10^5
* 0 <= nums[i] <= 10^9

**CODE:**

**def** maximumGap(nums):

**if** len(nums) **<** 2:

**return** 0

max\_num **=** max(nums)

*# Perform Radix Sort*

exp **=** 1

**while** max\_num **//** exp **>** 0:

counting\_sort(nums, exp)

exp **\*=** 10

max\_gap **=** 0

**for** i **in** range(1, len(nums)):

max\_gap **=** max(max\_gap, nums[i] **-** nums[i**-**1])

**return** max\_gap

**def** counting\_sort(nums, exp):

n **=** len(nums)

count **=** [0] **\*** 10

output **=** [0] **\*** n

**for** num **in** nums:

index **=** (num **//** exp) **%** 10

count[index] **+=** 1

**for** i **in** range(1, 10):

count[i] **+=** count[i**-**1]

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

index **=** (nums[i] **//** exp) **%** 10

output[count[index]**-**1] **=** nums[i]

count[index] **-=** 1

**for** i **in** range(n):

nums[i] **=** output[i]

5. **Contains Duplicate**

Given an integer array nums, return true if any value appears **at least twice** in the array, and return false if every element is distinct.

**Example 1:**

Input: nums = [1,2,3,1]

Output: true

**Example 2:**

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

Output: false

**Example 3:**

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

Output: true

**Constraints:**

* 1 <= nums.length <= 10^5
* 109 <= nums[i] <= 10^9

**CODE:**

**def** containsDuplicate(nums):

seen **=** set()

**for** num **in** nums:

**if** num **in** seen:

**return** **True**

seen**.**add(num)

**return** **False**

6. **Minimum Number of Arrows to Burst Balloons**

There are some spherical balloons taped onto a flat wall that represents the XY-plane. The balloons are represented as a 2D integer array points where points[i] = [xstart, xend] denotes a balloon whose **horizontal diameter** stretches between xstart and xend. You do not know the exact y-coordinates of the balloons.

Arrows can be shot up **directly vertically** (in the positive y-direction) from different points along the x-axis. A balloon with xstart and xend is **burst** by an arrow shot at x if xstart <= x <= xend. There is **no limit** to the number of arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.

Given the array points, return *the****minimum****number of arrows that must be shot to burst all balloons*.

**Example 1:**

Input: points = [[10,16],[2,8],[1,6],[7,12]]

Output: 2

Explanation: The balloons can be burst by 2 arrows:

- Shoot an arrow at x = 6, bursting the balloons [2,8] and [1,6].

- Shoot an arrow at x = 11, bursting the balloons [10,16] and [7,12].

**Example 2:**

Input: points = [[1,2],[3,4],[5,6],[7,8]]

Output: 4

Explanation: One arrow needs to be shot for each balloon for a total of 4 arrows.

**Example 3:**

Input: points = [[1,2],[2,3],[3,4],[4,5]]

Output: 2

Explanation: The balloons can be burst by 2 arrows:

- Shoot an arrow at x = 2, bursting the balloons [1,2] and [2,3].

- Shoot an arrow at x = 4, bursting the balloons [3,4] and [4,5].

**Constraints:**

* 1 <= points.length <= 10^5
* points[i].length == 2
* 231 <= xstart < xend <= 2^31 - 1

**CODE:**

**def** findMinArrowShots(points):

**if** **not** points:

**return** 0

points**.**sort(key**=lambda** x: x[1])

arrows **=** 1

curr\_end **=** points[0][1]

**for** i **in** range(1, len(points)):

**if** points[i][0] **>** curr\_end:

arrows **+=** 1

curr\_end **=** points[i][1]

**return** arrows

7. **Longest Increasing Subsequence**

Given an integer array nums, return *the length of the longest****strictly increasing***

***subsequence***

.

**Example 1:**

Input: nums = [10,9,2,5,3,7,101,18]

Output: 4

Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

**Example 2:**

Input: nums = [0,1,0,3,2,3]

Output: 4

**Example 3:**

Input: nums = [7,7,7,7,7,7,7]

Output: 1

**Constraints:**

* 1 <= nums.length <= 2500
* -10^4 <= nums[i] <= 10^4 </aside>

**CODE:**

**def** lengthOfLIS(nums):

**if** **not** nums:

**return** 0

n **=** len(nums)

dp **=** [1] **\*** n

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

**for** j **in** range(i):

**if** nums[i] **>** nums[j]:

dp[i] **=** max(dp[i], dp[j] **+** 1)

**return** max(dp)

8. **132 Pattern**

Given an array of n integers nums, a **132 pattern** is a subsequence of three integers nums[i], nums[j] and nums[k] such that i < j < k and nums[i] < nums[k] < nums[j].

Return true *if there is a****132 pattern****in* nums*, otherwise, return* false*.*

**Example 1:**

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

Output: false

Explanation: There is no 132 pattern in the sequence.

**Example 2:**

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

Output: true

Explanation: There is a 132 pattern in the sequence: [1, 4, 2].

**Example 3:**

Input: nums = [-1,3,2,0]

Output: true

Explanation: There are three 132 patterns in the sequence: [-1, 3, 2], [-1, 3, 0] and [-1, 2, 0].

**Constraints:**

* n == nums.length
* 1 <= n <= 2 \* 10^5
* -10^9 <= nums[i] <= 10^9 </aside>

**CODE:**

**def** find132pattern(nums):

stack **=** []

secondMax **=** float('-inf')

**for** i **in** range(len(nums) **-** 1, **-**1, **-**1):

**if** nums[i] **<** secondMax:

**return** **True**

**while** stack **and** nums[i] **>** stack[**-**1]:

secondMax **=** max(secondMax, stack**.**pop())

stack**.**append(nums[i])

**return** **False**