Q. **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.

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].

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

Q. **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.

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

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

Input: nums = [2,0,1]

Output: [0,1,2]

**Constraints:**

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

Q. **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.

Input: n = 5, bad = 4

Output: 4

Explanation:

Example 1:

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

Q. **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.

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

Q. **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.

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

Output: true

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

Output: false

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

Q. **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*.

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].

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.

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

Q. **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.

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

Output: 4

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

Output: 1

**Constraints:**

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

Q. **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*.*

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

Output: false

Explanation: There is no 132 pattern in the sequence.

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

Output: true

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

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

Answers: To solve these problems, I'll provide you with the solutions step by step.

Q1: Merge Intervals

To merge the overlapping intervals, we can follow these steps:

1. Sort the intervals based on their start time.

2. Initialize an empty list to store the merged intervals.

3. Iterate through the sorted intervals:

- If the merged list is empty or the current interval does not overlap with the previous merged interval, add the current interval to the merged list.

- Otherwise, update the end time of the previous merged interval if needed (when the end time of the current interval is greater), to merge the intervals.

4. Return the merged list.

Here's the implementation in Python:

```python

def merge\_intervals(intervals):

intervals.sort(key=lambda x: x[0]) # Sort based on start time

merged = []

for interval in intervals:

if not merged or merged[-1][1] < interval[0]:

merged.append(interval)

else:

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

return merged

```

Q2: Sort Colors

This problem is also known as the Dutch National Flag problem. We can solve it using the "two-pointer" approach.

1. Initialize three pointers: low, mid, and high.

- low points to the position where the next 0 should be placed.

- mid points to the position being currently examined.

- high points to the position where the next 2 should be placed.

2. Iterate through the array while mid <= high:

- If nums[mid] is 0, swap nums[mid] with nums[low], increment both low and mid.

- If nums[mid] is 1, no swapping is needed, just increment mid.

- If nums[mid] is 2, swap nums[mid] with nums[high], decrement high.

3. Repeat step 2 until mid <= high.

4. The array will be sorted after this process.

Here's the implementation in Python:

```python

def sort\_colors(nums):

low, mid, high = 0, 0, len(nums) - 1

while mid <= high:

if nums[mid] == 0:

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

low += 1

mid += 1

elif nums[mid] == 1:

mid += 1

else: # nums[mid] == 2

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

high -= 1

```

Q3: First Bad Version Solution

To find the first bad version efficiently, we can use a binary search approach.

1. Initialize two pointers, `left` and `right`, where `left = 1` and `right = n`.

2. While `left < right`, do:

- Calculate the mid version: `mid = left + (right - left) // 2`.

- If `isBadVersion(mid)` is `True`, set `right = mid` to search for the first bad version in the left half.

- Otherwise, set `left = mid + 1` to search for the first bad version in the right half.

3. After the binary search, `left` will be pointing to the first bad version.

Here's the implementation in Python:

```python

def first\_bad\_version(n):

left, right = 1, n

while left < right:

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

if isBadVersion(mid):

right = mid

else:

left = mid + 1

return left

```

Q4: Maximum Gap

To find the maximum gap between successive elements, we can use the radix sort algorithm.

1. Find the maximum element, `max\_num`, in the array.

2. Initialize the `exp` (exponent) variable as 1.

3. Create a temporary array, `output`, and a counting array, `count`, of size 10.

4. Until `max\_num // exp` becomes 0:

- Initialize the `count` array with zeros.

- Count the occurrences of each digit at the current exponent in the input array and store the count in the `count` array.

- Calculate the cumulative count by adding up the previous counts in the `count` array.

- Build the `output` array by placing the elements from the input array based on the count and digit values.

- Copy the elements from the `output` array to the input array.

- Multiply the `exp` variable by 10 to move to the next digit.

5. After the radix sort, iterate through the sorted array and find the maximum gap between successive elements.

Here's the implementation in Python:

```python

def maximum\_gap(nums):

if len(nums) < 2:

return 0

max\_num = max(nums)

exp = 1

n = len(nums)

output = [0] \* n

count = [0] \* 10

while max\_num // exp > 0:

count = [0] \* 10

for i in range(n):

count[(nums[i] // exp) % 10] += 1

for i in range(1, 10):

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

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

digit = (nums[i] // exp) % 10

output[count[digit] - 1] = nums[i]

count[digit] -= 1

nums = output.copy()

exp \*= 10

max\_gap = 0

for i in range(1, n):

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

return max\_gap

```

Q5: Contains Duplicate

To determine if any value appears at least twice in the array, we can use a hash set.

1. Initialize an empty set, `seen`.

2. Iterate through the array:

- If the current element is already in the `seen` set, return `True` (duplicate found).

- Otherwise, add the current element to the `seen` set.

3. If the loop completes without finding any duplicates, return `False`.

Here's the implementation in Python:

```python

def contains\_duplicate(nums):

seen = set()

for num in nums:

if num in seen:

return True

seen.add(num)

return False

```

Q6: Minimum Number of Arrows to Burst Balloons

To find the minimum number of arrows to burst all balloons, we can use a greedy algorithm.

1. Sort the balloons based on their end coordinate.

2. Initialize a variable, `end`, as negative infinity to track the end coordinate of the previous balloon.

3. Initialize a variable, `arrows`, as 0 to count the number of arrows needed.

4. Iterate through the sorted balloons:

- If the current balloon's start coordinate is greater than `end`, it means a new arrow is needed. Increment `arrows` and update `end` to the current balloon's end coordinate.

5. After the iteration, `arrows` will represent the minimum number of

arrows needed to burst all balloons.

Here's the implementation in Python:

```python

def find\_min\_arrows(points):

if not points:

return 0

points.sort(key=lambda x: x[1]) # Sort based on end coordinate

arrows = 1

end = points[0][1]

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

if points[i][0] > end:

arrows += 1

end = points[i][1]

return arrows

```

Q7: Longest Increasing Subsequence

To find the length of the longest strictly increasing subsequence, we can use the dynamic programming approach.

1. Create a list, `dp`, of length `n` initialized with 1s. `dp[i]` represents the length of the longest increasing subsequence ending at index `i`.

2. Iterate through the array from left to right:

- For each element at index `i`, iterate through all the elements before it from 0 to `i-1`.

- If `nums[i]` is greater than `nums[j]`, update `dp[i]` as `max(dp[i], dp[j] + 1)`. This means the longest increasing subsequence ending at index `i` can be extended by 1 if the element at index `j` is smaller than the element at index `i`.

3. After the iteration, the maximum value in the `dp` list represents the length of the longest increasing subsequence.

Here's the implementation in Python:

```python

def length\_of\_lis(nums):

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)

```

Q8: 132 Pattern

To determine if there is a 132 pattern in the array, we can use a stack.

1. Initialize an empty stack, `stack`, and a variable, `s3`, as negative infinity to track the third element of the 132 pattern.

2. Iterate through the array from right to left:

- If the current element is greater than `s3`, return `True` (132 pattern found).

- While the stack is not empty and the top element of the stack is less than the current element, update `s3` to the top element of the stack.

- Push the current element onto the stack.

3. If the loop completes without finding a 132 pattern, return `False`.

Here's the implementation in Python:

```python

def find132pattern(nums):

stack = []

s3 = float('-inf')

for num in reversed(nums):

if num > s3:

return True

while stack and stack[-1] < num:

s3 = stack.pop()

stack.append(num)

return False

```

I hope these explanations and implementations help you solve the given problems. Let me know if you have any further questions!