ASSIGNMENT 18

1. 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*.

```
def merge_intervals(intervals):
  intervals.sort(key=lambda x: x[0])
  merged_intervals = []

  for interval in intervals:
    if not merged_intervals or interval[0] > merged_intervals[-1][1]:
      merged_intervals.append(interval)
    else:
      merged_intervals[-1][1] = max(merged_intervals[-1][1], interval[1])

  return merged_intervals
```

2. Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** 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.

```
def sortColors(nums):
  low = 0
  mid = 0
  high = len(nums) - 1

  while mid <= high:
    if nums[mid] == 0:  # Red
      nums[mid], nums[low] = nums[low], nums[mid]
      mid += 1
      low += 1
    elif nums[mid] == 2:  # Blue
      nums[mid], nums[high] = nums[high], nums[mid]
      high -= 1
    else:  # White
      mid += 1
```

3. 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.

```
def isBadVersion(version):
    # The isBadVersion API implementation is not provided
    pass

def firstBadVersion(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. 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.

```
def maximumGap(nums):
    if len(nums) < 2:
        return 0

    max_num = max(nums)
    exp = 1

    while max_num // exp > 0:
        buckets = [[] for _ in range(10)]
        for num in nums:
            digit = (num // exp) % 10
            buckets[digit].append(num)
        nums = [num for bucket in buckets for num in bucket]
        exp *= 10

    max_diff = 0
    for i in range(1, len(nums)):
        max_diff = max(max_diff, nums[i] - nums[i-1])

    return max_diff
```

5. 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.

```
def containsDuplicate(nums):
    nums.sort()
    for i in range(1, len(nums)):
        if nums[i] == nums[i-1]:
            return True
    return False
```

6. 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*.

```
def findMinArrowShots(points):
    points.sort(key=lambda x: x[1])
    min_arrows = 1
    end = points[0][1]

    for i in range(1, len(points)):
        if points[i][0] > end:
            min_arrows += 1
            end = points[i][1]

    return min_arrows
```

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

subsequence

```
def lengthOfLIS(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)
```

8. 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`.

```
def find132pattern(nums):
    n = len(nums)
    numsk = float('-inf')
    stack = []

    for i in range(n - 1, -1, -1):
        if nums[i] < numsk:
            return True
        while stack and nums[i] > stack[-1]:
            numsk = stack.pop()
        stack.append(nums[i])

    return False
```