**DSA – ASSIGNMENT 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

**Solution. :-**

* Sort the intervals based on the start time.
* Initialize an empty list called merged to store the merged intervals.
* Iterate through each interval in the sorted list:
  + If the merged list is empty or the current interval does not overlap with the last interval in merged, add the current interval to merged.
  + Otherwise, merge the current interval with the last interval in merged by updating the end time of the last interval if necessary.
* Return the merged list.

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

**if not intervals:**

**return []**

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

**merged = [intervals[0]] # Initialize the merged list with the first interval**

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

**if interval[0] <= merged[-1][1]: # Check for overlap**

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

**else:**

**merged.append(interval) # No overlap, add interval to merged list**

**return merged**

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

**merged\_intervals = merge\_intervals(intervals)**

**print(merged\_intervals)**

💡 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.

**Solution. :-**

* Initialize three pointers: low to keep track of the boundary of the red section, mid to keep track of the boundary of the white section, and high to keep track of the boundary of the blue section.
* Initialize low and mid to the start of the array (index 0), and high to the end of the array (index len(nums) - 1).
* Iterate through the array using a pointer curr:
  + If nums[curr] is 0, swap it with nums[low], increment both low and mid pointers, and move curr to the next element.
  + If nums[curr] is 1, increment mid pointer, and move curr to the next element.
  + If nums[curr] is 2, swap it with nums[high], decrement the high pointer, but don't move curr yet (as the swapped element needs to be re-evaluated).
* Repeat step 3 until curr is greater than high.
* The array will be sorted in-place according to the colors.

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

**low = mid = 0**

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

**high -= 1**

**nums = [2, 0, 2, 1, 1, 0]**

**sort\_colors(nums)**

**print(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

**Solution. :-**

* Initialize two pointers, left and right, to the first and last version numbers, respectively. Set left to 1 and right to n.
* While left is less than right, do the following:
  + Calculate the midpoint: mid = left + (right - left) // 2.
  + Check if isBadVersion(mid) returns True:
    - If True, it means the first bad version is either mid or before it. So, we update right = mid.
    - If False, it means the first bad version is after mid. So, we update left = mid + 1.
* Once the binary search is complete and left and right converge, the value of left (or right) will be the first bad version.
* Return the value of left (or right).

**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**

**n = 5**

**bad = 4**

**first\_bad = first\_bad\_version(n)**

**print(first\_bad)**

💡 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

**Solution. :-**

* Check if the length of nums is less than 2. If so, return 0 since there are not enough elements to calculate the maximum gap.
* Find the maximum element, max\_num, in nums.
* Initialize a variable exp to 1, which represents the current digit place (starting with the least significant digit).
* Initialize an auxiliary array count of size 10, which will be used to count the occurrences of digits from 0 to 9.
* Iterate exp while max\_num // exp is greater than 0:
  + Reset the count array to all zeros.
  + Count the occurrences of each digit in nums at the current digit place exp and store them in the count array.
  + Update the count array by accumulating the counts. This will give us the starting indices for each digit in the sorted order.
  + Iterate over nums in reverse order:
    - Determine the digit value at the current digit place exp.
    - Decrement the count of the corresponding digit in the count array.
    - Use the decremented count as the index to place the current element in the sorted order in a temporary array.
  + Copy the elements from the temporary array back to nums.
  + Multiply exp by 10 to move to the next digit place.
* Iterate over nums to find the maximum difference between successive elements. Return the maximum difference as the maximum gap.

**def maximum\_gap(nums):**

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

**return 0**

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

**exp = 1**

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

**temp = [0] \* len(nums)**

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

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

**for num in nums:**

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

**count[digit] += 1**

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

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

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

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

**count[digit] -= 1**

**temp[count[digit]] = nums[i]**

**nums = temp[:]**

**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**

**nums = [3, 6, 9, 1]**

**max\_gap = maximum\_gap(nums)**

**print(max\_gap)**

💡 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

**Solution. :-**

* Initialize an empty hash set.
* Iterate over each element num in nums:
  + If num is already in the hash set, return True because it means we have encountered a duplicate value.
  + Otherwise, add num to the hash set.
* After the loop, if no duplicates were found, return False.

**def contains\_duplicate(nums):**

**seen = set()**

**for num in nums:**

**if num in seen:**

**return True**

**seen.add(num)**

**return False**

**nums = [1, 2, 3, 1]**

**result = contains\_duplicate(nums)**

**print(result)**

💡 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

**Solution. :-**

* Sort the points array based on the end coordinates of the balloons in ascending order.
* Initialize a variable arrowPos to store the position of the current arrow. Initialize it with the end coordinate of the first balloon.
* Initialize a variable arrowCount to keep track of the number of arrows shot. Set it to 1 since we already shot an arrow at the first balloon.
* Iterate over the balloons starting from the second one:
  + If the current balloon's start coordinate is greater than arrowPos, it means the balloon is not overlapping with the previous ones. In this case, increment arrowCount by 1 and update arrowPos with the current balloon's end coordinate.
  + If the current balloon is overlapping with the previous ones, update arrowPos with the minimum of arrowPos and the current balloon's end coordinate.
* Return arrowCount as the minimum number of arrows required.

**def find\_min\_arrows(points):**

**if not points:**

**return 0**

**# Sort the points based on end coordinates**

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

**arrowPos = points[0][1]**

**arrowCount = 1**

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

**if points[i][0] > arrowPos:**

**arrowCount += 1**

**arrowPos = points[i][1]**

**else:**

**arrowPos = min(arrowPos, points[i][1])**

**return arrowCount**

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

**result = find\_min\_arrows(points)**

**print(result)**

💡 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

**Solution. :-**

* Create an array dp of the same length as nums and initialize all its elements to 1. This array will store the lengths of the longest increasing subsequences ending at each index.
* Initialize a variable maxLen to 1. This variable will store the length of the longest increasing subsequence.
* Iterate over the elements of nums starting from the second element:
  + For each element, iterate over all the previous elements (from index 0 to the current index) to check if there is a subsequence ending at the current index that can be extended by including the current element. If nums[j] < nums[i], it means that we can extend the subsequence ending at index j by including the element at index i.
  + If the length of the subsequence ending at index j plus 1 is greater than the length of the subsequence ending at index i, update dp[i] with the length of the subsequence ending at index j plus 1.
  + Also, update maxLen with the maximum value between maxLen and dp[i].
* Return maxLen as the length of the longest increasing subsequence.

**def length\_of\_lis(nums):**

**if not nums:**

**return 0**

**n = len(nums)**

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

**maxLen = 1**

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

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

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

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

**maxLen = max(maxLen, dp[i])**

**return maxLen**

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

**result = length\_of\_lis(nums)**

**print(result)**

💡 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

**Solution. :-**

* Create an empty stack and initialize a variable s3 to negative infinity. The stack will store the candidate values for nums[j].
* Iterate over the elements of nums in reverse order (starting from the last element):
  + If the current element nums[i] is greater than s3, it means we have found a valid 132 pattern because nums[i] > nums[k] > nums[j]. Return True.
  + While the stack is not empty and the top element of the stack stack[-1] is less than the current element nums[i], update s3 with the top element of the stack and pop it from the stack. This is because nums[i] can potentially be nums[k] for future patterns.
  + Push the current element nums[i] onto the stack.
* If we reach the end of the array without finding a valid 132 pattern, return False.

**def find132pattern(nums):**

**n = len(nums)**

**stack = []**

**s3 = float('-inf')**

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

**if nums[i] < s3:**

**return True**

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

**s3 = stack.pop()**

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

**return False**

**nums = [1, 2, 3, 4]**

**result = find132pattern(nums)**

**print(result)**