

PASS THE PYTHON INTERVIEW

69 PYTHON CODING QUESTIONS, SOLUTIONS AND
EXPLANATIONS



LIONEL
OSAMBA

CONTENTS

Copyright

Introduction

Section 1 - Array

Section 2 - Binary

Section 3 - Dynamic Programming

Section 4 - Graph

Section 5 - Interval

Section 6 - Linked List

Section 7 - Matrix

Section 8 - String

Section 9 - Tree

Section 10 - Heap

Copyright © 2023 Lionel Osamba

All rights reserved

No part of this ebook may be reproduced, or stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without express written permission of the publisher.

INTRODUCTION

The soaring popularity of Python as the go-to language for Data Science and Machine Learning has triggered a real hunger for skilled Python pros. It's like every other company out there is on the hunt for analysts who can work their Python magic. And stepping in to meet this rising demand is a book that's packed to the brim with Python interview questions, starting from the easy stuff and ramping up to the not-so-easy. The whole point of this book? To arm folks with the knowledge they need to stride into interviews feeling cool and collected, even when faced with those head-scratching posers.

Inside this book, you'll find a treasure trove of interview questions that run the gamut of Python skills. But don't think it's just about tossing out questions and giving pat answers. Oh no, this book goes above and beyond. Each question comes with a step-by-step solution that's been meticulously put together. The aim? To make sure readers don't just get the theory, but can actually roll up their sleeves and wrestle with tough problems. What really sets this resource apart is how it takes care of beginners. The solutions are explained in a way that's crystal clear, breaking down the tough bits into manageable chunks, so you're not left scratching your head.

But what's the big picture here? This book's all about making interview prep a breeze, saving you time while supercharging your learning curve. Dive deep into the solutions and their explanations, and you'll soon be soaking in the ins and outs of Python programming for Data Science and Machine Learning.

And the icing on the cake? Landing plum positions at the big players like Walmart, Facebook, Amazon, and the whole superstar squad. Whether you're gearing up for some nerve-wracking technical interviews or just want to level up your Python game, this book's your secret weapon, paving a smooth path to climbing that career ladder.

SECTION 1

- ARRAY

Question 1

Given an array of non-negative integers ‘nums’ and a positive integer target, find two distinct numbers in the array such that their product is equal to the target.

Example 1:

Input: nums = [2, 3, 5, 10], target = 15

Output: [2, 3]

Explanation: Because $\text{nums}[2] * \text{nums}[3] == 15$, we return [2, 3].

Example 2:

Input: $\text{nums} = [6, 8, 4, 2]$, target = 12

Output: [2, 4]

Explanation: Because $\text{nums}[2] * \text{nums}[3] == 12$, we return [2, 4].

Example 3:

Input: $\text{nums} = [1, 5, 7]$, target = 35

Output: [1, 2]

Explanation: Because $\text{nums}[1] * \text{nums}[2] == 35$, we return [1, 2].

Constraints:

- $2 \leq \text{nums.length} \leq 10^4$
- $0 \leq \text{nums}[i] \leq 10^9$
- $0 < \text{target} \leq 10^9$
- Only one valid answer exists.

Solution:

```
1 class TwoNumbersProduct:
2     def find_two_numbers_with_product(self, nums, target):
3         num_index_map = {}
4
5         for index, num in enumerate(nums):
6             if num == 0 and target == 0:
7                 # Special case to handle when target is 0 and there's a 0 in the array
8                 if nums.count(0) > 1:
9                     return [index for index, n in enumerate(nums) if n == 0][:2]
10            elif target % num == 0:
11                complement = target // num
12                if complement in num_index_map:
13                    return [num_index_map[complement], index]
14                num_index_map[num] = index
15
16        return []
```

Solution Explanation:

1. We create a class called `TwoNumbersProduct` to encapsulate the logic for finding two distinct numbers in the given array whose product is equal to the target.
2. Inside the class, we define a method called `find_two_numbers_with_product(self, nums, target)`. This method takes two arguments: `nums`, which is the array of non-negative integers, and `target`, which is the positive integer we want to find as the product of two distinct numbers from the array.
3. We initialize an empty dictionary `num_index_map` to store the elements

and their corresponding indices. The purpose of this dictionary is to allow us to efficiently look up the index of a specific element during iteration.

4. We use a for loop along with enumerate to iterate through the array nums. The enumerate function gives us both the index and the element in each iteration.
5. For each element num at index index, we perform the following checks:
 - a. If num is 0 and the target is also 0, it means that the target product is 0, and we need to handle the special case when there are at least two 0's in the array. If both conditions are true, we check if the count of 0's in the array is greater than 1 using `nums.count(0) > 1`. If it is, we return the indices of the first two occurrences of 0 as the output since 0 multiplied by any number is 0. We use a list comprehension to find the indices of the first two occurrences of 0 in the array.

b. If the target is not 0 and it is divisible by num (i.e., $\text{target \% num} == 0$), it means that num could potentially be one of the numbers whose product with another number results in the target. So, we calculate the complement required to achieve the target product. The complement is calculated as $\text{complement} = \text{target} // \text{num}$. The $//$ operator performs integer division, ensuring that the result is an integer.

c. We check if the complement exists in the `num_index_map` dictionary. If it does, it means we have found two distinct elements (num and its complement) whose product is equal to the target. In this case, we return the indices of num and its complement from the `num_index_map` as the output.

6. If no valid pair of elements is found during the iteration, we return an empty list [] to indicate that there is no valid answer for the given `nums` and `target`.

7. After defining the TwoNumbersProduct class, we create an instance of the class using `tnp = TwoNumbersProduct()`.
8. We then call the method `find_two_numbers_with_product(nums, target)` on the instance `tnp` for each test case to find the two distinct numbers whose product is equal to the target.

The time complexity of this solution is $O(n)$, where n is the number of elements in the array. This is because we perform a single pass through the array during the iteration. The space complexity is also $O(n)$ due to the `num_index_map` dictionary, which can store at most n elements, where n is the number of elements in the array.

Question 2

You are given an array '**nums**' of positive integers, and you are allowed to perform at most two operations:

1. Choose any index '**i**' and increment '**num-s[i]**' by 1.
2. Choose any index '**j**' and decrement '**num-s[j]**' by 1.

Your goal is to maximize the sum of the elements in the array. Return the maximum sum you can achieve after at most two operations.

Example:

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

Output: 10

Explanation: You can increment '**nums[0]**' and '**nums[3]**' to make the array [2, 2, 3, 5], and the sum of the elements is $2 + 2 + 3 + 5 = 12$. However, you can also decrement '**nums[2]**' twice to make the array [1, 2, 1, 4], and the sum of the elements is $1 + 2 + 1 + 4 = 8$. The maximum sum achievable is 10.

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

Output: 15

Explanation: You can increment '**nums[0]**', '**nums[2]**', and '**nums[4]**' to make the array [6, 1, 4, 2, 5], and the sum of the elements is $6 + 1 + 4 + 2 + 5 = 18$. However, you can also decrement '**nums[0]**' twice and '**nums[3]**' once to make the array [3, 1, 4, 1, 4], and the sum of the elements is $3 + 1 + 4 + 1 + 4 = 13$. The maximum sum achievable is 15.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $0 \leq \text{nums}[i] \leq 10^4$

Solution:

```
1  class MaximizeSum:
2      def max_sum_after_operations(self, nums):
3          n = len(nums)
4          max_sum = float('-inf')
5
6          # Case 1: No operation
7          current_sum = sum(nums)
8          max_sum = max(max_sum, current_sum)
9
10         # Case 2: Increment one element and decrement another element
11         for i in range(n):
12             for j in range(n):
13                 # Skip if the indices are the same or adjacent
14                 if i == j or abs(i - j) == 1:
15                     continue
16                 # Increment nums[i] and decrement nums[j]
17                 current_sum = sum(nums) - nums[j] + nums[i] + 1
18                 max_sum = max(max_sum, current_sum)
19
20         return max_sum
```

Solution Explanation:

1. We define a class MaximizeSum to encapsulate the logic for finding the maximum sum of the elements in the array after at most two operations.
2. The class has a method called max_sum_after_operations(self, nums). This method takes the nums array as input

and returns the maximum sum achievable after at most two operations.

3. Inside the method, we first initialize the variable `max_sum` to negative infinity. This will be used to keep track of the maximum sum obtained after trying all possible operations.
4. We start with the first case, where we perform no operation. We calculate the sum of all elements in the `nums` array using `sum(nums)` and store it in `current_sum`. Then, we update `max_sum` to be the maximum of the current maximum and the `current_sum`.
5. Next, we consider the case where we increment one element and decrement another element. We use two nested loops to try out all possible combinations of indices `i` and `j`. We skip the combination if the indices are the same or adjacent (i.e., `i == j` or `abs(i - j) == 1`) since this would result in no change in the array.

6. For each valid combination of i and j , we increment $\text{nums}[i]$ and decrement $\text{nums}[j]$, and then calculate the new sum of the elements in the array. We subtract $\text{nums}[j]$ and add $\text{nums}[i]$ to the sum and add 1 to account for the increment operation. We store this new sum in `current_sum`.
7. We update `max_sum` to be the maximum of the current maximum and the `current_sum`.
8. Finally, after trying out all possible operations, we return the `max_sum` as the maximum sum achievable.

The time complexity of this solution is $O(n^2)$, where n is the number of elements in the `nums` array. This is because we use two nested loops to try out all combinations of indices. The space complexity is $O(1)$ since we use a constant amount of additional memory regardless of the size of the input array.

By encapsulating the logic in a class, we can reuse the `MaximizeSum` class and its method for different

test cases, making the code more modular and organized.

Question 3

Given an array '**nums**' of positive integers, find if there exists a subarray of '**nums**' such that the product of all elements in the subarray is equal to a given positive integer '**target**'.

Example:

Input: nums = [2, 3, 5, 10, 1], target = 30

Output: True

Explanation: The subarray [2, 3, 5] has a product equal to the target 30.

Input: nums = [6, 8, 4, 2], target = 50

Output: False

Explanation: There is no subarray whose product is equal to the target 50.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^5$
- $1 \leq \text{target} \leq 10^9$

Solution:

```
1  class SubarrayProduct:
2      def check_subarray_product(self, nums, target):
3          n = len(nums)
4          left = 0
5          product = 1
6
7          for right in range(n):
8              product *= nums[right]
9
10         while product >= target and left <= right:
11             if product == target:
12                 return True
13             product //= nums[left]
14             left += 1
15
16     return False
```

Solution Explanation:

1. We define a class SubarrayProduct to encapsulate the logic for finding if there exists a subarray of nums such that the product of all elements in the subarray is equal to the given target.
2. The class has a method called `check_subarray_product(self, nums, target)`. This method takes the `nums` array and the target as inputs and returns `True` if such a subarray exists; otherwise, it returns `False`.
3. Inside the method, we initialize two pointers `left` and `right` to keep track of the

sliding window. We also initialize a variable product to store the product of the elements within the current window.

4. We start with an empty window, so left is initialized to 0 and product is initialized to 1.
5. We use a for loop with the variable right ranging from 0 to n-1, where n is the length of the nums array.
6. In each iteration, we update the product by multiplying it with the element at the current right index, i.e., product *= nums[right].
7. We then check if the product is greater than or equal to the target. If it is, it means the current window contains a subarray whose product is greater than or equal to the target. In this case, we enter a while loop.
8. The while loop is used to adjust the window size by incrementing the left pointer and dividing the product by the element

at the left index until the product becomes less than the target or until the left pointer reaches the right pointer.

9. During each iteration of the while loop, we check if the product is equal to the target. If it is, we have found a subarray whose product is equal to the target, so we return True.
10. If we exit the for loop without finding a subarray with the desired product, it means no such subarray exists, and we return False.

The time complexity of this solution is $O(n)$, where n is the number of elements in the `nums` array. This is because we use a single pass through the array with both the left and right pointers. The space complexity is $O(1)$ since we use a constant amount of additional memory regardless of the size of the input array.

Question 4

You are given an array '**nums**' of positive integers.
For each element '**nums[i]**', find the nearest element

on its left and right side that is strictly greater than '**nums[i]**'. If there is no such element, consider it as -1. Return the sum of these nearest greater elements for each element in the array.

Example:

Input: `nums = [3, 6, 8, 2, 7, 5]`

Output: 19

Explanation: For each element in the array, the nearest greater elements on its left and right side are as follows:

For element 3: Nearest greater elements are -1 on the left and 6 on the right.

For element 6: Nearest greater elements are 3 on the left and 8 on the right.

For element 8: Nearest greater elements are 6 on the left and -1 on the right.

For element 2: Nearest greater elements are -1 on the left and 7 on the right.

For element 7: Nearest greater elements are 2 on the left and -1 on the right.

For element 5: Nearest greater elements are 2 on the left and -1 on the right.

The sum of nearest greater elements for each element is 19.

Constraints:

- $2 \leq \text{nums.length} \leq 10^5$
- $-30 \leq \text{nums}[i] \leq 30$
- The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

Solution:

```
1  class NearestGreaterElements:
2      def find_sum_of_nearest_greater_elements(self, nums):
3          n = len(nums)
4          stack = []
5          left_greaters = [-1] * n
6          right_greaters = [-1] * n
7          sum_greaters = 0
8
9          # Calculate the nearest greater elements on the left side
10         for i in range(n):
11             while stack and nums[stack[-1]] <= nums[i]:
12                 stack.pop()
13             if stack:
14                 left_greaters[i] = nums[stack[-1]]
15             stack.append(i)
16
17         # Clear the stack and calculate the nearest greater elements on the right side
18         stack.clear()
19         for i in range(n - 1, -1, -1):
20             while stack and nums[stack[-1]] <= nums[i]:
21                 stack.pop()
22             if stack:
23                 right_greaters[i] = nums[stack[-1]]
24             stack.append(i)
25
26         # Calculate the sum of the nearest greater elements for each element
27         for i in range(n):
28             sum_greaters += left_greaters[i] + right_greaters[i]
29
30     return sum_greaters
```

Solution Explanation:

1. We define a class `NearestGreaterElements` to encapsulate the logic for finding the sum of nearest greater elements for each element in the `nums` array.
2. The class has a method called `find_sum_of_nearest_greater_elements(self, nums)`. This method takes the `nums` array as input and returns the sum of nearest greater elements for each element.
3. Inside the method, we first initialize some variables and lists. `n` is the length of the `nums` array. `stack` is used to store the indices of elements in a decreasing order

of values. `left_greaters` and `right_greaters` are lists to store the nearest greater elements on the left and right side of each element, respectively. `sum_greaters` is used to store the sum of nearest greater elements for each element.

4. We calculate the nearest greater elements on the left side of each element using a loop that traverses the `nums` array from left to right. For each element, we compare it with the top of the stack (the last element added). If the element in the stack is smaller than or equal to the current element, we remove it from the stack as it cannot be the nearest greater element on the left side of the current element. We keep popping elements until the stack is empty or the top element is greater than the current element. After finding the nearest greater element on the left side, we store it in the `left_greaters` list.

5. Next, we clear the stack and calculate the nearest greater elements on the right side of each element using a loop that traverses the nums array from right to left. The process is similar to the one explained in step 4 but in the opposite direction. After finding the nearest greater element on the right side, we store it in the right_greaters list.
6. Finally, we calculate the sum of nearest greater elements for each element by summing up the elements from the left_greaters and right_greaters lists at each index.
7. We return the sum_greaters as the output.

The time complexity of this solution is $O(n)$, where n is the number of elements in the nums array. We traverse the array twice, once from left to right and once from right to left. The space complexity is $O(n)$ as we use additional lists (left_greaters and right_

greater) to store the nearest greater elements for each element.

Question 5

You are given an array '**nums**' of integers. Find the length of the longest subarray in which the difference between the maximum and minimum element is less than or equal to a given integer '**k**'. Return the length of the longest subarray.

Example:

Input: **nums** = [2, 6, 1, 4, 9, 3], **k** = 3

Output: 5

Explanation: The longest subarray in which the difference between the maximum and minimum element is less than or equal to 3 is [2, 6, 1, 4, 3], and its length is 5.

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

Output: 5

Explanation: The longest subarray in which the difference between the maximum and minimum element is less than or equal to 0 is [1, 1, 1, 1, 1], and its length is 5.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- $0 \leq k \leq 10^4$

Solution:

```
3  class LongestSubarrayWithDifference:
4      def longest_subarray_length(self, nums, k):
5          n = len(nums)
6          max_queue = deque()
7          min_queue = deque()
8          left = 0
9          longest_len = 0
10
11         for right in range(n):
12             # Remove elements outside the current window from both queues
13             while max_queue and nums[right] > max_queue[-1]:
14                 max_queue.pop()
15             while min_queue and nums[right] < min_queue[-1]:
16                 min_queue.pop()
17
18             # Add the current element to both queues
19             max_queue.append(nums[right])
20             min_queue.append(nums[right])
21
22             # Adjust the window size
23             while max_queue[0] - min_queue[0] > k:
24                 if nums[left] == max_queue[0]:
25                     max_queue.popleft()
26                 if nums[left] == min_queue[0]:
27                     min_queue.popleft()
28                 left += 1
29
30             # Update the longest subarray length
31             longest_len = max(longest_len, right - left + 1)
32
33         return longest_len
```

Solution Explanation:

1. We define a class `LongestSubarrayWithDifference` to encapsulate the logic for finding the length of the longest subarray in which the difference between the maximum and minimum element is less than or equal to k .
2. The class has a method called `longest_subarray_length(self, nums, k)`. This method takes the `nums` array and the value k as inputs and returns the length of the longest subarray.
3. Inside the method, we initialize some variables and two deques `max_queue` and `min_queue` to store the indices of elements in the `nums` array. The `max_queue` deque will store elements in decreasing order of values, and the `min_queue` deque will store elements in increasing order of values. `left` is the left pointer of the sliding

window, and longest_len is used to store the length of the longest subarray.

4. We use a loop with the variable right ranging from 0 to n-1, where n is the length of the nums array.
5. In each iteration, we adjust the max_queue and min_queue to remove elements outside the current window. We keep popping elements from the right side of the queues until the rightmost element is greater than or equal to the current element for max_queue and until the rightmost element is less than or equal to the current element for min_queue. This ensures that both queues always store the maximum and minimum elements within the current window.
6. After adjusting the queues, we add the current element to both max_queue and min_queue since it is part of the current window.

7. We then adjust the window size by moving the left pointer (left) and removing elements from both queues until the difference between the maximum and minimum elements within the window is less than or equal to k. This ensures that the window contains the longest subarray that meets the condition.
8. After adjusting the window, we update the longest_len to be the maximum of the current longest length and the length of the current subarray (right - left + 1).
9. After the loop, we return the longest_len as the output.

The time complexity of this solution is $O(n)$, where n is the number of elements in the nums array. We traverse the array once with the right pointer and move the left pointer at most n times, making the overall time complexity linear. The space complexity is $O(k)$, as the maximum number of elements we store

in the queues at any point in time is $k+1$, where k is the given integer.

Question 6

You are given an array ‘nums’ of positive integers. Find the maximum sum of a subarray in the array. A subarray is a contiguous part of the array. Return the maximum sum.

Example:

Input: nums = [4, -3, 5, -2, -1, 2, 6, -2]

Output: 11

Explanation: The subarray with the maximum sum is [5, -2, -1, 2, 6], and its sum is 11.

Input: nums = [1, 2, 3, -2, 5]

Output: 9

Explanation: The subarray with the maximum sum is [1, 2, 3], and its sum is 9.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

Solution:

```
1  class MaxSubarraySum:
2      def max_subarray_sum(self, nums):
3          max_sum = nums[0] # Initialize max_sum with the first element
4          current_sum = nums[0] # Initialize current_sum with the first element
5
6          for num in nums[1:]:
7              # Compare the current element with the sum
8              # of current element and previous subarray sum
9              current_sum = max(num, current_sum + num)
10
11              # Update max_sum if the current subarray sum is greater
12              max_sum = max(max_sum, current_sum)
13
14      return max_sum
```

Solution Explanation:

1. We define a class MaxSubarraySum to encapsulate the logic for finding the max-

imum sum of a subarray in the nums array.

2. The class has a method called `max_subarray_sum(self, nums)`. This method takes the `nums` array as input and returns the maximum sum.
3. Inside the method, we initialize two variables: `max_sum` and `current_sum`. Both are initialized with the first element of the `nums` array.
4. We use a loop to iterate through the `nums` array starting from the second element (index 1).
5. For each element, we calculate the maximum of two values: the current element (`num`) and the sum of the current element and the previous subarray sum (`current_sum + num`). This step allows us to decide whether to extend the current subarray or start a new subarray from the current element.

6. We update `current_sum` to be the maximum of the two values calculated in the previous step. This ensures that `current_sum` always stores the maximum sum of subarrays ending at each index.
7. We update `max_sum` to be the maximum of the current value of `max_sum` and `current_sum`. This way, `max_sum` always stores the maximum sum of all subarrays considered so far.
8. After the loop, we return `max_sum` as the output, which represents the maximum sum of a subarray in the `nums` array.

The time complexity of this solution is $O(n)$, where n is the number of elements in the `nums` array. We traverse the array once, updating `current_sum` and `max_sum` at each step. The space complexity is $O(1)$ as we are using a constant amount of additional space to store the variables `max_sum` and `current_sum`.

Question 7

You are given an array '**nums**' of integers that represents the number of people at various positions. Each element in the array represents the number of

people at a specific position, and the positions are arranged in ascending order. The array '**nums**' is rotated between 1 and n times, where n is the length of the array. Find the position with the maximum number of people and return its index.

Example:

Input: nums = [5, 6, 3, 4]

Output: 1

Explanation: The original array was [3, 4, 5, 6] rotated 2 times. The position with the maximum number of people is at index 1, and the number of people at that position is 6.

Input: nums = [7, 2, 3, 5, 6]

Output: 0

Explanation: The original array was [2, 3, 5, 6, 7] rotated 1 time. The position with the maximum number of people is at index 0, and the number of people at that position is 7.

Constraints:

- $1 \leq \text{nums.length} \leq 5000$
- $-5000 \leq \text{nums}[i] \leq 5000$
- All the integers of '**nums**' are unique.
- **nums** is sorted and rotated between 1 and n times.

Solution:

```
1 class MaxPeoplePosition:  
2     def find_max_people_position(self, nums):  
3         pivot = next((i for i in range(1, len(nums)) if nums[i] < nums[i - 1]), 0)  
4         target = max(nums[pivot], nums[pivot - 1])  
5         return nums.index(target) if target in nums else -1
```

Solution Explanation:

1. We define a class `MaxPeoplePosition` with a method called `find_max_people_position(self, nums)`.

2. Inside the method, we use a generator expression and the next function to find the pivot point (pivot) where the rotation happened. The pivot point is the index where the next element is smaller than the current element.
3. We then find the target element, which is the maximum between the pivot and the element before the pivot.
4. If the target element is present in the nums array, we return its index using the index method. Otherwise, we return -1 to indicate that the maximum number of people position is not found in the array.

The time complexity and space complexity remain the same as $O(\log n)$ and $O(1)$, respectively.

Question 8

You are given a rotated sorted array `nums` with distinct integers. Prior to being passed to your function, the array `nums` is possibly rotated at an unknown

pivot index k ($0 \leq k < \text{nums.length}$). You are also given an integer target. Write a function that finds the target element in the array nums.

If the target is found, return its index; otherwise, return -1.

Your algorithm must run in $O(\log n)$ time complexity.

Example:

Input: nums = [7, 10, 15, 1, 3, 5], target = 3

Output: 4

Explanation: The original sorted array was [1, 3, 5, 7, 10, 15] rotated at pivot index 3. The target element 3 is found at index 4 in the rotated array.

Input: nums = [4, 6, 8, 9, 1, 3], target = 8

Output: 2

Explanation: The original sorted array was [1, 3, 4, 6, 8, 9] rotated at pivot index 4. The target element 8 is found at index 2 in the rotated array.

Input: nums = [5, 6, 7, 8, 9, 10, 2, 3], target = 7

Output: 2

Explanation: The original sorted array was [2, 3, 5, 6, 7, 8, 9, 10] rotated at pivot index 2. The target element 7 is found at index 2 in the rotated array.

Constraints:

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- All values of nums are unique.
- nums is an ascending array that is possibly rotated.
- $-10^4 \leq \text{target} \leq 10^4$

Solution:

```
1  class RotatedSortedArray:
2      def search(self, nums, target):
3          left, right = 0, len(nums) - 1
4
5          while left <= right:
6              mid = left + (right - left) // 2
7              if nums[mid] == target:
8                  return mid
9
10             if nums[left] <= nums[mid]:
11                 if nums[left] <= target < nums[mid]:
12                     right = mid - 1
13                 else:
14                     left = mid + 1
15             else:
16                 if nums[mid] < target <= nums[right]:
17                     left = mid + 1
18                 else:
19                     right = mid - 1
20
21         return -1
```

Solution Explanation:

1. We use a single binary search loop instead of two separate loops for each part of the array.
2. We compare the target element with the leftmost and middle element to determine which part of the array is sorted.
3. Based on whether the target lies in the sorted part of the array or not, we update the left and right pointers accordingly.
4. If the target element is found, we return its index; otherwise, we return -1 to indicate that the target is not in the array.

The time complexity of this shorter solution is still $O(\log n)$, and the space complexity is $O(1)$.

Question 9

You are given an integer array `nums`. Your task is to find all the triplets `[nums[i], nums[j], nums[k]]` such

that $i \neq j \neq k$, and $\text{nums}[i] * \text{nums}[j] * \text{nums}[k] == 0$. Each triplet should be returned as a list in the output.

Example:

Input: $\text{nums} = [1, 2, 0, 3, 0, 4]$

Output: $[[1, 2, 0], [1, 0, 3], [2, 0, 3], [0, 3, 0], [0, 0, 4]]$

Explanation: There are five triplets that satisfy the condition:

1. $\text{nums}[0] * \text{nums}[1] * \text{nums}[2] = 1 * 2 * 0 = 0$
2. $\text{nums}[0] * \text{nums}[1] * \text{nums}[4] = 1 * 2 * 0 = 0$
3. $\text{nums}[0] * \text{nums}[3] * \text{nums}[4] = 1 * 3 * 0 = 0$
4. $\text{nums}[2] * \text{nums}[3] * \text{nums}[4] = 0 * 3 * 0 = 0$
5. $\text{nums}[2] * \text{nums}[4] * \text{nums}[5] = 0 * 0 * 4 = 0$

Input: $\text{nums} = [0, 0, 0]$

Output: [[0, 0, 0]]

Explanation: There is one triplet that satisfies the condition:

1. $\text{nums}[0] * \text{nums}[1] * \text{nums}[2] = 0 * 0 * 0 = 0$

Input: $\text{nums} = [1, 2, 3, 4]$

Output: []

Explanation: There are no triplets that satisfy the condition.

Constraints:

- $3 \leq \text{nums.length} \leq 3000$
- $-10^5 \leq \text{nums}[i] \leq 10^5$

Solution:

```
1  class Solution:
2      def threeSumZeroProduct(self, nums):
3          nums.sort()
4          result = set()
5
6          for i in range(len(nums) - 2):
7              if i > 0 and nums[i] == nums[i - 1]:
8                  continue
9
10         left, right = i + 1, len(nums) - 1
11
12         while left < right:
13             product = nums[i] * nums[left] * nums[right]
14             if product == 0:
15                 result.add((nums[i], nums[left], nums[right]))
16                 # Move left and right pointers to find other solutions
17                 left += 1
18                 right -= 1
19                 # Skip duplicates
20                 while left < right and nums[left] == nums[left - 1]:
21                     left += 1
22                 while left < right and nums[right] == nums[right + 1]:
23                     right -= 1
24             elif product < 0:
25                 left += 1
26             else:
27                 right -= 1
28
29         return list(result)
```

Solution Explanation:

1. Sort the array: The first step is to sort the input array `nums` in ascending order. Sorting the array helps in optimizing the solution and finding the triplets efficiently.
2. Initialize an empty set to store the unique triplets: We will use a set to store the triplets to ensure that there are no duplicate triplets in the result. Sets in Python automatically handle duplicates and ensure uniqueness.
3. Iterate through the array: We will use a loop to iterate through the sorted array `nums` from the first element to the second last element (index `len(nums) - 2`). The reason we stop at the second last element is because we need at least three elements to form a triplet (i.e., i, j , and k where $i \neq j \neq k$).

4. Skip duplicates: Since the array is sorted, if $\text{nums}[i]$ is the same as $\text{nums}[i-1]$, then we will get duplicate triplets, so we skip to the next iteration.
5. Set left and right pointers: We set two pointers left and right to the elements right after i and at the end of the array, respectively. The left pointer points to the element immediately after i , and the right pointer points to the last element of the array.
6. Use Two-pointer technique: We will use a two-pointer technique to find pairs of elements whose product is zero. The idea is to check the product of the current element $\text{nums}[i]$ with the elements pointed by the left and right pointers.
7. Calculate the product: We calculate the product of the three elements $\text{nums}[i]$, $\text{nums}[\text{left}]$, and $\text{nums}[\text{right}]$ and store it in the variable product.

8. Check if the product is zero: If the product is zero, it means we have found a triplet that satisfies the condition $\text{nums}[i] * \text{nums}[\text{left}] * \text{nums}[\text{right}] == 0$. We add this triplet to the result set.
9. Move pointers: After finding a triplet, we need to move the left pointer to the right to explore other possibilities for left, and move the right pointer to the left to explore other possibilities for right. Additionally, we skip any duplicate elements while moving the pointers.
10. Adjust pointers if the product is negative or positive: If the product is negative, it means the current triplet has a negative value, and we need a larger value to make the product zero. So, we increment the left pointer. If the product is positive, it means the current triplet has a positive value, and we need a smaller value to make the product zero. So, we decrement the right pointer.

11. Continue until the pointers meet: We repeat steps 7 to 10 until the left and right pointers meet or cross each other.
12. Convert the set to a list and return: Finally, we convert the set of triplets to a list and return it as the output.

Overall, the solution uses a two-pointer technique to efficiently find all unique triplets in the array whose product is zero. The time complexity of the solution is $O(n^2)$, where n is the number of elements in the array. Sorting the array takes $O(n \log n)$ time, and finding the triplets using two pointers takes $O(n^2)$ time. The space complexity is $O(n)$ due to the use of the result set to store the triplets.

Question 10

You are given an integer array `nums` of length `n`. The array contains some positive and negative integers. Your task is to find the maximum absolute difference between two elements in the array.

ence between the sum of any two non-overlapping subarrays of the array.

A subarray of the given array is any contiguous part of the array. For example, a subarray of [4, 5, 6] could be [4], [5], [6], [4, 5], [5, 6], or [4, 5, 6].

The absolute difference between two subarrays is defined as the absolute difference between the sum of the elements in the two subarrays.

Return the maximum absolute difference.

Example:

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

Output: 8

Explanation: The maximum absolute difference is achieved by choosing subarrays [1, 2] and [-3, 4]. The sum of [1, 2] is 3, and the sum of [-3, 4] is 1. The absolute difference is $|3 - 1| = 2$, which is the maximum absolute difference.

Input: nums = [4, -2, 5, 3]

Output: 12

Explanation: The maximum absolute difference is achieved by choosing subarrays [4, -2] and [5, 3]. The sum of [4, -2] is 2, and the sum of [5, 3] is 8. The absolute difference is $|2 - 8| = 6$, which is the maximum absolute difference.

Constraints:

- The length of the array nums is between 2 and 10^5 .
- Each element in nums is between -10^4 and 10^4 .

Solution:

```
1  class Solution:
2      def maxAbsoluteDifference(self, nums):
3          n = len(nums)
4          left_dp = [0] * n
5          right_dp = [0] * n
6
7          # Compute left_dp array
8          current_sum = 0
9          for i in range(n):
10              current_sum += nums[i]
11              left_dp[i] = max(current_sum, left_dp[i-1] if i > 0 else 0)
12              if current_sum < 0:
13                  current_sum = 0
14
15          # Compute right_dp array
16          current_sum = 0
17          for i in range(n-1, -1, -1):
18              current_sum += nums[i]
19              right_dp[i] = max(current_sum, right_dp[i+1] if i < n-1 else 0)
20              if current_sum < 0:
21                  current_sum = 0
22
23          # Calculate the maximum absolute difference
24          max_diff = 0
25          for i in range(1, n):
26              max_diff = max(max_diff, abs(left_dp[i-1] - right_dp[i]))
27
28          return max_diff
```

Solution Explanation:

1. We initialize two arrays, `left_dp` and `right_dp`, with length n to store the maximum sum of subarrays ending at and starting from each index, respectively.
2. We compute the `left_dp` array by iterating through the array and keeping track of the current sum. If the current sum becomes negative, we reset it to 0 because we don't want to consider negative sums as they would reduce the overall sum of a subarray. We update the `left_dp[i]` with the

maximum of the current_sum and left_dp[i-1].

3. We compute the right_dp array in a similar way, but we iterate through the array in reverse order to find the maximum sum of subarrays starting at each index.
4. Finally, we iterate through the array (excluding the first element) and calculate the maximum absolute difference between the sums of the two non-overlapping subarrays using left_dp and right_dp arrays. We keep track of the maximum absolute difference in the variable max_diff and return it as the result.

The time complexity of this solution is $O(n)$ because we make three passes through the input array, and the space complexity is also $O(n)$ to store the left_dp and right_dp arrays.

SECTION 2 -

BINARY

Question 11

Provide a solution to calculate the absolute difference between two integers, a and b , using only bitwise operators and basic arithmetic operations (addition, subtraction, multiplication, division). Include examples and ensure that the solution remains within the given constraints.

Example 1:

Input: a = 5, b = 3

Output: 2

Example 2:

Input: a = -7, b = 4

Output: 11

Constraints:

- $-1000 \leq a, b \leq 1000$

Solution:

```
1  class AbsoluteDifferenceCalculator:  
2      def calculate_absolute_difference(self, a, b):  
3          mask = 0xFFFFFFFF # 32-bit mask to handle negative numbers  
4  
5          while b:  
6              a, b = (a ^ b) & mask, ((a & b) << 1) & mask  
7  
8          return a if a <= 0x7FFFFFFF else ~(a ^ mask)  
9
```

Solution Explanation:

1. We create a class named `AbsoluteDifferenceCalculator` to encapsulate the logic.

2. Inside the class, the method `calculate_absolute_difference` is defined to perform the absolute difference calculation.
3. We use bitwise operators to calculate the difference between `a` and `b` while considering overflow cases (using a 32-bit mask).
4. The loop continues until `b` becomes zero.
5. In the end, we check if the result `a` is a negative number (i.e., its most significant bit is set). If so, we calculate the two's complement to obtain the correct absolute difference value.

Question 12

Create a function that takes a positive integer num and returns the number of prime numbers within

the range from 1 to num, inclusive.

Note:

In some programming languages, such as Python, you can use the `isprime()` function from certain libraries to check for prime numbers.

In languages without a built-in prime-checking function, implement a suitable algorithm to determine if a number is prime.

Example 1:

Input: num = 10

Output: 4

Explanation: There are four prime numbers within the range 1 to 10: 2, 3, 5, and 7.

Example 2:

Input: num = 20

Output: 8

Explanation: There are eight prime numbers within the range 1 to 20: 2, 3, 5, 7, 11, 13, 17, and 19.

Example 3:

Input: num = 30

Output: 10

Explanation: There are ten prime numbers within the range 1 to 30: 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29.

Constraints:

- num is a positive integer greater than or equal to 2.

Solution

```
1  class PrimeCounter:
2      def count_primes(self, num):
3          if num < 2:
4              return 0
5
6          is_prime = [True] * (num + 1)
7          is_prime[0] = is_prime[1] = False
8
9          for i in range(2, int(num ** 0.5) + 1):
10             if is_prime[i]:
11                 for j in range(i * i, num + 1, i):
12                     is_prime[j] = False
13
14         return sum(is_prime)
```

Solution Explanation:

1. We create a class named PrimeCounter to encapsulate the prime-counting logic.
2. Inside the class, the method count_primes is defined to calculate the number of prime numbers within the given range.
3. We initialize a list is_prime of size num + 1, where each element represents whether the corresponding number is prime.
4. We set is_prime[0] and is_prime[1] to False since 0 and 1 are not prime.
5. We use the Sieve of Eratosthenes algorithm to mark non-prime numbers in the is_prime list. Starting from 2, we mark all multiples of each prime number as non-prime.
6. The loop iterates until “i” reaches the square root of num, as any larger factor of

num must be a multiple of a smaller factor already marked as non-prime.

- Finally, we sum up the True values in the is_prime list to get the count of prime numbers within the given range.

The provided solution uses the Sieve of Eratosthenes algorithm to efficiently count prime numbers within the given range while adhering to the specified constraints.

Question 13

Given a non-negative integer n, create a function that generates an array of length $n + 1$, where each element $\text{ans}[i]$ represents the count of unique digits in the decimal representation of i.

Example 1:

Input: $n = 4$

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

Explanation:

0 has 1 unique digit: 0

1 has 1 unique digit: 1

2 has 1 unique digit: 2

3 has 2 unique digits: 3, 2

4 has 2 unique digits: 4, 2

Example 2:

Input: $n = 7$

Output: [1, 1, 1, 2, 2, 3, 3, 4]

Explanation:

0 has 1 unique digit: 0

1 has 1 unique digit: 1

2 has 1 unique digit: 2

3 has 2 unique digits: 3, 2

4 has 2 unique digits: 4, 2

5 has 3 unique digits: 5, 3, 2

6 has 3 unique digits: 6, 3, 2

7 has 4 unique digits: 7, 6, 3, 2

Constraints:

- $0 \leq n \leq 10^5$

Solution:

```
1  class UniqueDigitCounter:
2      def count_unique_digits(self, n):
3          def count_digits(num):
4              digit_set = set()
5              while num > 0:
6                  digit_set.add(num % 10)
7                  num //= 10
8              return len(digit_set)
9
10     ans = [0] * (n + 1)
11
12     for i in range(n + 1):
13         ans[i] = count_digits(i)
14
15     return ans
```

Solution Explanation:

1. We create a class named UniqueDigitCounter to encapsulate the unique digit counting logic.
2. Inside the class, the method count_unique_digits is defined to calculate the array of unique digit counts.
3. The nested function count_digits is used to count the unique digits in a given number. We use a set to keep track of unique digits.

4. We initialize an array `ans` of length $n + 1$ to store the unique digit counts for each number from 0 to n .
5. We iterate through each number i from 0 to n and calculate its unique digit count using the `count_digits` function.
6. Finally, we return the `ans` array containing the unique digit counts for each number in the given range.

The provided solution uses a class-based approach to generate an array of unique digit counts for the decimal representation of each number within the given range.

Question 14

You are given an array `nums` containing n unique numbers in the range $[1, n + 1]$. However, one number from this range is missing in the array. Write a function to find and return the missing number.

Example 1:

Input: `nums = [3, 7, 1, 5]`

Output: 2

Explanation: The array contains numbers from 1 to 8, and 2 is missing.

Example 2:

Input: nums = [1, 4, 6, 3, 5]

Output: 2

Explanation: The array contains numbers from 1 to 7, and 2 is missing.

Example 3:

Input: nums = [10, 8, 7, 2, 4, 1, 3, 9]

Output: 5

Explanation: The array contains numbers from 1 to 11, and 5 is missing.

Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 10^4$
- $1 \leq \text{nums}[i] \leq n + 1$
- All the numbers of nums are unique.

Solution:

```
1 class MissingNumberFinder:
2     def find_missing_number(self, nums):
3         n = len(nums)
4         total_sum = (n + 1) * (n + 2) // 2 # Sum of numbers from 1 to n+1
5
6         array_sum = sum(nums)
7         missing_number = total_sum - array_sum
8
9     return missing_number
```

Solution Explanation:

1. We create a class named MissingNumberFinder to encapsulate the missing number finding logic.

2. Inside the class, the method `find_missing_number` is defined to calculate and return the missing number in the array.
3. We calculate the total sum of numbers from 1 to $n + 1$ using the formula $(n + 1) * (n + 2) // 2$.
4. We calculate the sum of the elements in the given array using the `sum()` function.
5. The missing number is the difference between the total sum and the array sum.
6. Finally, we return the calculated missing number.

Question 15

Reverse the digits of a given 32-bit signed integer and return the resulting integer.

Note:

In languages like Python, you can use the built-in `int()` and `str()` functions to handle the integer-to-string and string-to-integer conversions.

Example 1:

Input: `num = 12345`

Output: 54321

Explanation: The input integer 12345 is reversed to 54321.

Example 2:

Input: `num = -9876`

Output: -6789

Explanation: The input integer -9876 is reversed to -6789.

Example 3:

Input: num = 1200

Output: 21

Explanation: The input integer 1200 is reversed to 21.

Constraints:

- $-2^{31} \leq \text{num} \leq 2^{31} - 1$

Solution:

```
1  class IntegerReverser:
2      def reverse_integer(self, num):
3          INT_MAX = 2**31 - 1
4          INT_MIN = -2**31
5
6          sign = 1 if num >= 0 else -1
7          num = abs(num)
8
9          reversed_num = 0
10         while num > 0:
11             digit = num % 10
12             if reversed_num > (INT_MAX - digit) // 10:
13                 return 0 # Overflow check
14             reversed_num = reversed_num * 10 + digit
15             num //= 10
16
17         return reversed_num * sign
```

Solution Explanation:

1. We create a class named IntegerReverser to encapsulate the integer reversing logic.
2. Inside the class, the method `reverse_integer` is defined to reverse the digits of the given integer and return the resulting integer.
3. We define `INT_MAX` and `INT_MIN` constants to represent the maximum and minimum 32-bit signed integer values.
4. We determine the sign of the input integer and work with its absolute value.
5. Using a while loop, we repeatedly extract the last digit of the input integer using the modulus operator and add it to the reversed number. We also perform an overflow check to ensure the reversed number does not exceed the 32-bit integer limits.
6. Finally, we return the reversed number with the original sign.

The provided solution efficiently reverses the digits of a 32-bit signed integer while considering overflow conditions.

SECTION 3

- DYNAMIC PROGRAMMING

Question 16

You are designing a robot to move across a grid. The robot can only move in two directions: right or down. Given a grid of dimensions $m \times n$, how many distinct paths can the robot take to reach the bottom-right corner from the top-left corner?

Example 1:

Input: $m = 3, n = 2$

Output: 3

Explanation: There are three distinct paths to reach the bottom-right corner:

Right -> Down -> Down

Down -> Down -> Right

Down -> Right -> Down

Example 2:

Input: $m = 7, n = 3$

Output: 28

Constraints:

- $1 \leq m, n \leq 100$

Solution:

```
1  class RobotGridPaths:
2      def unique_paths(self, m, n):
3          dp = [[0] * n for _ in range(m)]
4
5          for i in range(m):
6              dp[i][0] = 1
7
8          for j in range(n):
9              dp[0][j] = 1
10
11         for i in range(1, m):
12             for j in range(1, n):
13                 dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
14
15         return dp[m - 1][n - 1]
```

Solution Explanation:

1. We create a class named RobotGridPaths to encapsulate the grid path counting logic.
2. Inside the class, the method unique_paths is defined to calculate and return the number of distinct paths.
3. We use a dynamic programming approach to solve this problem. We create a 2D array dp of size m x n to store the count of paths to each cell.
4. We initialize the first row and first column of the dp array to 1, as there is only one way to reach any cell in the first row or first column (either moving right or moving down).
5. We then iterate over the remaining cells in the grid, and for each cell (i, j) , we calculate the number of paths to that cell as the sum of paths from the cell above $(i - 1, j)$ and the cell to the left $(i, j - 1)$.

6. The value at $dp[m - 1][n - 1]$ represents the total number of distinct paths to reach the bottom-right corner of the grid.

The provided solution efficiently calculates the number of distinct paths a robot can take to reach the bottom-right corner of a grid while adhering to the specified constraints.

Question 17

You are designing an automated toll collection system for a highway. The system needs to determine the optimal way to provide change to the drivers. You have a set of coin denominations available to give as change.

Write a function that takes a list of coin denominations and an amount of money as input, and returns the total number of distinct ways the change can be provided using the given denominations. If the amount cannot be made up by any combination of the coins, return -1.

Example 1:

Input: denominations = [1, 5, 10], amount = 6

Output: 3

Explanation: There are three distinct ways to provide change for 6 units: [1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 5], and [1, 5].

Example 2:

Input: denominations = [2, 3, 7], amount = 10

Output: 2

Explanation: There are two distinct ways to provide change for 10 units: [2, 2, 2, 2, 2] and [3, 3, 2, 2].

Example 3:

Input: denominations = [1, 2, 3, 4], amount = 8

Output: 8

Explanation: There are eight distinct ways to provide change for 8 units: [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 2, 2], [1, 1, 1, 2, 2], [1, 1, 2, 2, 2], [2, 2, 2, 2], [1, 1, 1, 1, 1, 4], and [1, 1, 2, 4].

Constraints:

- $1 \leq \text{denominations.length} \leq 12$
- $1 \leq \text{denominations}[i] \leq 100$
- $0 \leq \text{amount} \leq 1000$

Solution:

```
1  class CoinChangeWays:
2      def coin_change(self, denominations, amount):
3          dp = [0] * (amount + 1)
4          dp[0] = 1
5
6          for coin in denominations:
7              for i in range(coin, amount + 1):
8                  dp[i] += dp[i - coin]
9
10         return dp[amount] if dp[amount] != 0 else -1
```

Solution Explanation:

1. We create a class named ‘CoinChangeWays’ to encapsulate the coin change counting logic.
2. Inside the class, the method `coin_change` is defined to calculate and return the total number of distinct ways.
3. We use a dynamic programming approach to solve this problem. We create an array `dp` of size `amount + 1` to store the count of ways for each amount.
4. We initialize `dp[0]` to 1, as there is one way to provide no change (using no coins).
5. For each coin in the denominations, we iterate over the `dp` array and update the counts based on the current coin. We start from the value of the coin since using a coin of smaller value cannot make up the current amount.

6. The final value at $dp[amount]$ represents the total number of distinct ways to provide change for the given amount.

The provided solution efficiently calculates the total number of distinct ways to provide change while adhering to the specified constraints.

Question 18

Given an array of integers arr, find the length of the longest subsequence in which the absolute difference between adjacent elements is greater than or equal to a given positive integer k.

Example 1:

Input: arr = [3, 8, 2, 15, 7], k = 5

Output: 3

Explanation: The longest subsequence with absolute difference ≥ 5 is [3, 8, 15].

Example 2:

Input: arr = [1, 4, 6, 10, 15], k = 3

Output: 2

Explanation: The longest subsequence with absolute difference ≥ 3 is [1, 4].

Example 3:

Input: arr = [5, 5, 5, 5, 5], k = 2

Output: 1

Explanation: The longest subsequence with absolute difference ≥ 2 is [5].

Constraints:

- $1 \leq \text{arr.length} \leq 2500$
- $-10^4 \leq \text{arr}[i] \leq 10^4$
- $1 \leq k \leq 10^4$

Solution:

```
1  class LongestSubsequence:
2      def longest_subsequence(self, arr, k):
3          n = len(arr)
4          dp = [1] * n # Initialize all elements with 1
5
6          for i in range(1, n):
7              for j in range(i):
8                  if abs(arr[i] - arr[j]) >= k:
9                      dp[i] = max(dp[i], dp[j] + 1)
10
11     return max(dp)
```

Solution Explanation:

1. We create a class named LongestSubsequence to encapsulate the longest subsequence calculation logic.
2. Inside the class, the method longest_subsequence is defined to calculate and return the length of the longest subsequence.
3. We use dynamic programming to solve this problem. We create an array dp of the same size as the input array arr and initialize all elements to 1, as any element itself is a valid subsequence of length 1.
4. We iterate through the elements of the input array, and for each element arr[i], we compare it with all elements before it. If the absolute difference between arr[i] and arr[j] is greater than or equal to k, then we update dp[i] to be the maximum of its current value and dp[j] + 1, where

$dp[j]$ represents the length of the subsequence ending at index j .

- Finally, we return the maximum value in the dp array, which represents the length of the longest subsequence with the given condition.

Question 19

Given two strings $word1$ and $word2$, find the length of the shortest string that contains both $word1$ and $word2$ as subsequences. If no such string exists, return -1.

A subsequence of a string is a new string generated from the original string with some characters (can

be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".

Example 1:

Input: word1 = "abcde", word2 = "ace"

Output: 5

Explanation: The shortest string that contains both "abcde" and "ace" as subsequences is "abcdae", and its length is 5.

Example 2:

Input: word1 = "abc", word2 = "abc"

Output: 3

Explanation: The shortest string that contains both "abc" and "abc" as subsequences is "abc", and its length is 3.

Example 3:

Input: word1 = "abc", word2 = "def"

Output: -1

Explanation: There is no such string that contains both "abc" and "def" as subsequences, so the result is -1.

Constraints:

- $1 \leq \text{word1.length}, \text{word2.length} \leq 1000$
- word1 and word2 consist of only lowercase English characters.

Solution:

```
1  class ShortestCommonSupersequence:
2      def shortest_common_supersequence(self, word1, word2):
3          m, n = len(word1), len(word2)
4          dp = [[0] * (n + 1) for _ in range(m + 1)]
5
6          for i in range(m + 1):
7              for j in range(n + 1):
8                  if i == 0:
9                      dp[i][j] = j
10                 elif j == 0:
11                     dp[i][j] = i
12                 elif word1[i - 1] == word2[j - 1]:
13                     dp[i][j] = dp[i - 1][j - 1] + 1
14                 else:
15                     dp[i][j] = min(dp[i - 1][j], dp[i][j - 1]) + 1
16
17         return dp[m][n]
```

Solution Explanation:

1. We create a class named `ShortestCommonSupersequence` to encapsulate the shortest common supersequence calculation logic.
2. Inside the class, the method `shortest_common_supersequence` is defined to calculate and return the length of the shortest common supersequence.
3. We use dynamic programming to solve this problem. We create a 2D array `dp` of size $(m+1) \times (n+1)$, where m is the length of `word1` and n is the length of `word2`.

4. We initialize the first row and first column of the dp array to represent the lengths of subsequences of empty strings and the corresponding word.
5. We then iterate through the remaining cells of the dp array. If the characters at the current positions in both word1 and word2 are equal, we increment the value in the dp array by 1. Otherwise, we take the minimum of the values in the cell above and the cell to the left, and add 1 to it.
6. The value in the bottom-right cell of the dp array represents the length of the shortest common supersequence.

Question 20

Given a string text and a list of strings patternDict, return true if text can be segmented into a sequence of one or more strings from the patternDict. Note that the same string in the dictionary may be reused multiple times in the segmentation.

Example 1:

Input: text = "programming", patternDict = ["pro", "gram", "ming"]

Output: true

Explanation: Return true because "programming" can be segmented as "pro gram ming".

Example 2:

Input: text = "appledogapple", patternDict = ["apple", "dog"]

Output: true

Explanation: Return true because "appledogapple" can be segmented as "apple dog apple".

Note that you are allowed to reuse a dictionary string.

Example 3:

Input: text = "treehouse", patternDict = ["tree", "house"]

Output: true

Explanation: Return true because "treehouse" can be segmented as "tree house".

Constraints:

- $1 \leq \text{text.length} \leq 300$

- $1 \leq \text{patternDict.length} \leq 1000$
- $1 \leq \text{patternDict[i].length} \leq 20$
- text and patternDict[i] consist of only lower-case English letters.
- All the strings of patternDict are unique.

Solution:

```
1  class WordSegmentation:
2      def word_break(self, text, patternDict):
3          n = len(text)
4          dp = [False] * (n + 1)
5          dp[0] = True
6
7          for i in range(1, n + 1):
8              for j in range(i):
9                  if dp[j] and text[j:i] in patternDict:
10                      dp[i] = True
11                      break
12
13      return dp[n]
```

Solution Explanation:

1. We create a class named WordSegmentation to encapsulate the word segmentation logic.
2. Inside the class, the method word_break is defined to check if the given text can be segmented using the words in the patternDict.
3. We use dynamic programming to solve this problem. We create an array dp of size $(n+1)$, where n is the length of the text. Each element $dp[i]$ represents whether the substring $text[0:i]$ can be segmented using words from the patternDict.
4. We initialize $dp[0]$ to True, as an empty string can always be segmented.
5. We iterate through each index i from 1 to n, and for each index, we iterate through each index j from 0 to i-1. If $dp[j]$ is True (meaning the substring $text[0:j]$ can be segmented) and the substring $text[j:i]$ is

present in the patternDict, we set $dp[i]$ to True and break out of the inner loop.

- Finally, we return $dp[n]$, which indicates whether the entire text can be segmented.

Question 21

You are given an array of distinct integers ‘numbers’ and a target integer ‘target’. Return the maximum number of non-overlapping subsequences of ‘numbers’ that sum up to exactly ‘target’.

Example 1:

Input: numbers = [2, 4, 6, 8, 10], target = 12

Output: 3

Explanation: The possible non-overlapping subsequences are: [2, 10], [4, 8], [6, 6].

Example 2:

Input: numbers = [3, 5, 7, 9], target = 12

Output: 2

Explanation: The possible non-overlapping subsequences are: [3, 9], [5, 7].

Example 3:

Input: numbers = [1, 2, 3, 4, 5], target = 8

Output: 4

Explanation: The possible non-overlapping subsequences are: [1, 2, 5], [1, 3, 4], [2, 3, 3], [1, 2, 2, 3].

Constraints:

- $1 \leq \text{numbers.length} \leq 200$
- $1 \leq \text{numbers}[i] \leq 1000$
- All the elements of numbers are unique.
- $1 \leq \text{target} \leq 10000$

Solution:

```
1  class MaxNonOverlappingSubsequences:
2      def maxNonOverlapping(self, numbers, target):
3          dp = [0] * (len(numbers) + 1)
4          prefix_sum = {0: 0}
5          last_index = {-1: -1}
6          current_sum = 0
7
8          for i, num in enumerate(numbers):
9              current_sum += num
10             dp[i + 1] = dp[i]
11
12             if current_sum - target in prefix_sum:
13                 prev_index = prefix_sum[current_sum - target]
14                 dp[i + 1] = max(dp[i + 1], dp[prev_index] + 1)
15
16             prefix_sum[current_sum] = i
17             last_index[i] = i
18
19             if i > 0:
20                 last_index[i] = max(last_index[i], last_index[i - 1])
21
22     return dp[len(numbers)]
```

Solution Explanation:

1. We create a class named MaxNonOverlappingSubsequences to encapsulate the solution.
2. Inside the class, the method maxNonOverlapping takes the list of numbers and the target as input and returns the maximum number of non-overlapping subsequences that sum up to exactly the target.
3. We use dynamic programming to solve this problem. We initialize a list dp of size $(\text{len}(\text{numbers}) + 1)$ to keep track of the maximum number of subsequences at each index.
4. We also create a prefix_sum dictionary to store the cumulative sum up to each index and a last_index dictionary to keep track of the last index where a particular cumulative sum was seen.

5. We iterate through the numbers using a loop and update the prefix_sum and last_index dictionaries accordingly.
6. At each index i , we check if there exists a previous index $prev_index$ such that the difference between the current cumulative sum and the target is seen at $prefix_sum[current_sum - target]$.
7. If such an index exists, we update $dp[i + 1]$ to be the maximum of $dp[i]$ and $dp[prev_index] + 1$.
8. Finally, we return $dp[len(numbers)]$, which represents the maximum number of non-overlapping subsequences that sum up to the target.

Question 22

You are a treasure hunter exploring a series of vaults arranged in a linear pattern. Each vault contains a certain amount of treasure, but there's a twist – adjacent vaults are connected by a security system that will trigger an alarm if two adjacent vaults are entered on the same night. Your goal is to maximize the total amount of treasure you can obtain without triggering the security system.

Given an array ‘vaults’ representing the amount of treasure in each vault, return the maximum amount of treasure you can obtain tonight without setting off the security system.

Example 1:

Input: vaults = [5, 10, 2, 7, 15]

Output: 32

Explanation: You can enter vaults 1, 3, and 5 to collect the treasures: $10 + 2 + 15 = 27$.

Example 2:

Input: vaults = [3, 8, 4, 1, 9]

Output: 18

Explanation: You can enter vaults 2 and 5 to collect the treasures: $8 + 9 = 17$.

Example 3:

Input: vaults = [6, 2, 1, 9, 4, 5]

Output: 16

Explanation: You can enter vaults 1, 4, and 6 to collect the treasures: $6 + 9 + 1 = 16$.

Constraints:

- $1 \leq \text{vauls.length} \leq 100$
- $0 \leq \text{vauls}[i] \leq 50$

Solution:

```
1  class MaxTreasureHunter:
2      def maxTreasure(self, vaults):
3          if not vaults:
4              return 0
5
6          n = len(vaults)
7          dp = [0] * n
8          dp[0] = vaults[0]
9
10         for i in range(1, n):
11             dp[i] = max(dp[i-1], (dp[i-2] if i >= 2 else 0) + vaults[i])
12
13         return dp[-1]
```

Solution Explanation:

1. We create a class named MaxTreasure-Hunter to encapsulate the solution.
2. Inside the class, the method maxTreasure takes the list of vaults as input and returns the maximum amount of treasure that can be obtained without setting off the security system.
3. We use dynamic programming to solve this problem. We initialize a list dp of size n to keep track of the maximum amount of treasure that can be obtained up to each vault.

4. We iterate through the vaults using a loop and update the dp array accordingly. At each vault i , we calculate the maximum amount of treasure that can be obtained either by skipping the current vault ($dp[i-1]$) or by including the current vault along with the vault two steps back ($dp[i-2] + vaults[i]$).
5. We return $dp[-1]$, which represents the maximum amount of treasure that can be obtained without triggering the security system.

Question 23

You are a professional robber planning to loot houses

in a circular neighborhood. Each house has a certain amount of money stashed. All houses in this circular neighborhood are arranged in a circle, forming a loop. That means the first house is adjacent to both the last house and the second house. Meanwhile, adjacent houses have a security system connected, and it will automatically contact the police if two adjacent houses were looted on the same night.

Given an integer array `loot` representing the amount of money in each house, return the maximum amount of money you can loot tonight without alerting the police.

Example 1:

Input: `loot` = [3, 5, 2, 10, 6]

Output: 15

Explanation: You can loot house 2 (money = 5), house 4 (money = 10), and house 1 (money = 3). Total loot = $5 + 10 + 3 = 15$.

Example 2:

Input: loot = [8, 4, 2, 7, 6]

Output: 14

Explanation: You can loot house 1 (money = 8) and house 4 (money = 7). Total loot = $8 + 7 = 14$.

Example 3:

Input: loot = [5, 3, 1, 6, 4]

Output: 10

Explanation: You can loot house 1 (money = 5) and house 4 (money = 6). Total loot = $5 + 6 = 10$.

Constraints:

- $1 \leq \text{loot.length} \leq 100$
- $0 \leq \text{loot}[i] \leq 1000$

Solution:

```
1  class Robber:
2      def rob(self, loot):
3          def simple_rob(nums):
4              prev1 = prev2 = 0
5              for num in nums:
6                  temp = prev1
7                  prev1 = max(prev2 + num, prev1)
8                  prev2 = temp
9              return prev1
10
11     if len(loot) == 1:
12         return loot[0]
13
14     return max(simple_rob(loot[:-1]), simple_rob(loot[1:]))
```

Solution Explanation:

1. We define a class Robber to encapsulate the solution logic.
2. We define a helper function simple_rob(nums) to calculate the maximum loot amount using the standard "House Robber" approach. It takes a list of house loot as input and returns the maximum loot.
3. In the rob method, if there's only one house, we can directly return its loot.
4. We calculate the maximum loot by considering two cases: robbing houses from the first to the second-to-last and robbing houses from the second to the last. This is to handle the circular arrangement of houses.
5. We return the maximum loot among the two cases.

The key insight is that we consider two separate cases (excluding the first or the last house) and choose the one with the maximum loot. This way, we handle the circular arrangement of houses effectively and avoid adjacent houses in the loot.

Question 24

You are given a string containing a sequence of digits. Each digit corresponds to a certain character mapping as follows:

'0' -> "A"

'1' -> "B"

'2' -> "C"

...

'9' -> "J"

To encode a message, you need to find all possible ways of replacing the digits with their corresponding character mappings. For example, the sequence "101" can be encoded in the following ways:

"ABA" with the mapping (1 0 1)

"KBA" with the mapping (10 1)

"AJB" with the mapping (1 0 10)

Given a string seq containing only digits, return the total number of distinct ways to encode the message.

Example:

Input: seq = "101"

Output: 3

Explanation: The sequence "101" can be encoded in 3 different ways as shown above.

Input: seq = "222"

Output: 5

Explanation: The sequence "222" can be encoded in 5 different ways: "CCC", "BBB", "CJB", "BJC", "BKB".

Input: seq = "063"

Output: 0

Explanation: The sequence "063" cannot be mapped to a valid character encoding due to the leading zero ("6" is different from "06").

Constraints:

- $1 \leq \text{seq.length} \leq 100$
- seq contains only digits and may contain leading zero(s).

Solution:

```
1  class Solution:
2      def numDecodings(self, seq: str) -> int:
3          if not seq or seq[0] == '0':
4              return 0
5
6          n = len(seq)
7          dp = [0] * (n + 1)
8          dp[0], dp[1] = 1, 1
9
10         for i in range(2, n + 1):
11             one_digit = int(seq[i - 1])
12             two_digits = int(seq[i - 2:i])
13
14             if one_digit >= 1:
15                 dp[i] += dp[i - 1]
16
17             if 10 <= two_digits <= 26:
18                 dp[i] += dp[i - 2]
19
20         return dp[n]
```

Solution Explanation:

1. We use dynamic programming to solve this problem. We create a dp array where $dp[i]$ represents the number of ways to decode the substring $seq[0:i]$.
2. We initialize $dp[0]$ and $dp[1]$ to 1, as there's only one way to decode a single-digit number.
3. We iterate through the string starting from the third character (index 2) to the end. For each index i , we consider two possibilities: decoding the current digit as a single-digit number and decoding the current digit along with the previous digit as a two-digit number.

4. If the current digit is between 1 and 9 (inclusive), we add $dp[i - 1]$ to $dp[i]$ since we can decode the current digit as a single-digit number.
5. If the two-digit number formed by the previous and current digits is between 10 and 26 (inclusive), we add $dp[i - 2]$ to $dp[i]$ since we can decode the two digits together.
6. Finally, the value of $dp[n]$ represents the total number of ways to decode the entire string.

The solution uses dynamic programming to efficiently count the number of distinct ways to encode the given sequence while considering the character mappings and the constraints provided in the problem statement.

Question 25

You are designing a game board with a grid of dimensions $m \times n$. A player starts at the top-left corner (i.e., $\text{grid}[0][0]$) and aims to reach the bottom-right corner (i.e., $\text{grid}[m - 1][n - 1]$). The player can only move either down or right at any point in time. Each cell of the grid contains a score, and the player aims to maximize their total score as they navigate from the top-left corner to the bottom-right corner.

Given the two integers m and n , along with the grid containing scores for each cell, return the maximum possible total score the player can achieve while moving from the top-left corner to the bottom-right corner.

The test cases are generated so that the answer will be within a reasonable range.

Example:

Input:

$m = 4, n = 3$

```
grid = [  
    [3, 2, 4],  
    [1, 8, 5],  
    [9, 1, 2],  
    [2, 3, 7]  
]
```

Output:

Maximum total score: 32

Explanation: The player can take the following path to maximize their score: (0,0) -> (1,0) -> (1,1) -> (1,2) -> (2,2) -> (3,2). The total score would be $3 + 1 + 8 + 5 + 2 + 7 = 26$.

Constraints:

- $1 \leq m, n \leq 10$
- $1 \leq \text{grid}[i][j] \leq 100$

Solution:

```
1  class GameBoard:
2      def maxScore(self, m, n, grid):
3          # Create a 2D DP array to store the maximum score at each cell
4          dp = [[0] * n for _ in range(m)]
5
6          # Initialize the top-left cell's score
7          dp[0][0] = grid[0][0]
8
9          # Initialize the first column's scores
10         for i in range(1, m):
11             dp[i][0] = dp[i - 1][0] + grid[i][0]
12
13         # Initialize the first row's scores
14         for j in range(1, n):
15             dp[0][j] = dp[0][j - 1] + grid[0][j]
16
17         # Fill in the DP array using the recurrence relation
18         for i in range(1, m):
19             for j in range(1, n):
20                 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + grid[i][j]
21
22         # The maximum score will be in the bottom-right corner
23         return dp[m - 1][n - 1]
```

Solution Explanation:

1. We create a class GameBoard with a method maxScore that takes the dimensions m and n, along with the grid of scores as input.
2. We create a 2D DP array dp of size m x n to store the maximum score that can be achieved at each cell.
3. We initialize the top-left cell's score as the score of the top-left cell in the grid.
4. We then initialize the scores for the first column and first row of the DP array. For each cell, we calculate its score based on

the score of the adjacent cells (top and left) plus the current cell's score from the grid.

5. We fill in the rest of the DP array using the recurrence relation: $dp[i][j] = \max(dp[i - 1][j], dp[i][j - 1]) + grid[i][j]$.
6. Finally, the maximum score will be in the bottom-right corner of the DP array, which we return as the result.

Question 26

You are given an integer array ‘steps’ representing the number of steps you can jump at each position. You are initially positioned at the array’s first index,

and each element in the array represents the maximum number of steps you can take from that position. Additionally, each step has a corresponding energy cost associated with it.

You start with an initial energy level of 'startEnergy'. At each step, you can choose to jump to the next position with the given number of steps, but it will cost you the energy equal to the step's energy cost. If your energy drops to zero or below, you cannot continue the journey.

Return true if you can reach the last index with a positive energy level, or false otherwise.

Write a function 'can_reach_end' that takes the following parameters:

- A list of integers ‘steps’ representing the number of steps you can jump at each position.
- A list of integers ‘energyCost’ representing the energy cost at each position.
- An integer ‘startEnergy’ representing the initial energy level.

Your function should return a boolean value: ‘True’ if you can reach the last index with a positive energy level, or ‘False’ otherwise.

Constraints:

- $1 \leq \text{Length of steps} \leq 10^4$
- $1 \leq \text{Length of energyCost} \leq 10^4$
- $0 \leq \text{Each element of steps, energyCost} \leq 10^5$
- $1 \leq \text{startEnergy} \leq 10^5$

Note: In this variation of the problem, you need to consider not only the number of steps you can take but also the energy cost associated with each step. You need to determine whether you can reach the

last index while maintaining a positive energy level throughout the journey.

Solution:

```
1  class JumpGameWithEnergy:
2      @staticmethod
3      def can_reach_end(steps, energy_cost, start_energy):
4          n = len(steps)
5          dp = [0] * n # Create a list to store the maximum energy at each position
6
7          # Initialize the first position's energy with the start energy
8          dp[0] = start_energy
9
10         for i in range(1, n):
11             # Calculate the energy at the current position
12             dp[i] = max(dp[i - 1] - energy_cost[i - 1], 0) + steps[i - 1]
13
14             # If the energy at the current position is non-negative, continue
15             if dp[i] < 0:
16                 return False
17
18         # Check if the final energy at the last position is positive
19         return dp[-1] >= 0
```

Solution Explanation:

1. We define a static method `can_reach_end` within the `JumpGameWithEnergy` class that takes three parameters: `steps`, `energy_cost`, and `start_energy`.

2. We initialize n as the length of the steps list.
3. We create a list dp of size n to store the maximum energy at each position. $dp[i]$ will represent the maximum energy at position i.
4. We initialize the first position's energy ($dp[0]$) with the start_energy.
5. We iterate from the second position (index 1) to the last position (index $n - 1$).
6. For each position i, we calculate the energy at that position by subtracting the energy cost of the previous position and adding the steps taken from the previous position. The formula $dp[i] = \max(dp[i - 1] - energy_cost[i - 1], 0) + steps[i - 1]$ ensures that the energy doesn't drop below zero.
7. If the energy at the current position becomes negative, it means the player cannot continue the journey, so we return False.

- After iterating through all positions, we check if the final energy at the last position ($dp[-1]$) is non-negative. If it is, it means the player can reach the last index with a positive energy level, so we return True.

This solution uses dynamic programming to keep track of the maximum energy at each position and ensures that the energy remains non-negative throughout the journey.

SECTION 4 -

GRAPH

Question 27

You are given a reference to a node in a directed graph, where each node represents a task and has a value (int) and a list (List[Node]) of its dependencies. You need to create a copy of the graph while maintaining the dependencies.

Implement a function `clone_dependency_graph` that takes the following parameters:

A list of nodes `nodes`, where each node is represented by a tuple `(val, dependencies)`:

`val` is the value (int) of the node/task.

`dependencies` is a list of integers representing the indices of nodes that the current node depends on.

Your function should return a new list of nodes representing the copied graph with the same dependencies.

Test case format:

For simplicity, each node's value is the same as its index (0-indexed). The graph is represented in the test case using the `nodes` list.

Example 1:

Input: `nodes = [(0, [1, 2]), (1, [2]), (2, [2])]`

Output: `[(0, [1, 2]), (1, [2]), (2, [2])]`

Explanation: There are 3 nodes in the graph.

Node 0's dependencies are nodes 1 and 2.

Node 1's dependency is node 2.

Node 2's dependency is itself.

Example 2:

Input: nodes = [(0, []), (1, []), (2, [])]

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

Explanation: The graph consists of 3 isolated nodes, each without dependencies.

Example 3:

Input: nodes = []

Output: []

Explanation: This is an empty graph with no nodes.

Constraints:

- The number of nodes in the graph is in the range [0, 100].
- $0 \leq \text{val} \leq 100$
- val is unique for each node.
- There are no repeated edges.
- The graph may not be connected, and nodes might not be reachable from each other.

Solution:

```
1  class Node:
2      def __init__(self, val, neighbors=None):
3          self.val = val
4          self.neighbors = neighbors if neighbors is not None else []
5
6  def clone_dependency_graph(nodes):
7      copied_nodes = {}
8
9  def dfs(node):
10     if node in copied_nodes:
11         return copied_nodes[node]
12
13     copied_node = Node(node.val)
14     copied_nodes[node] = copied_node
15     copied_node.neighbors = [
16         dfs(neighbor) for neighbor in node.neighbors
17     ]
18
19     return copied_node
20
21     return [(dfs(node), [
22         copied_nodes[dep] for dep in dependencies
23     ]) for node, dependencies in nodes]
```

Solution Explanation:

1. We define a `Node` class to represent nodes in the graph.
2. The `clone_dependency_graph` function takes a list of nodes as input and returns a list of copied nodes with their dependencies.
3. We use a recursive DFS approach to create copies of nodes and their dependencies while maintaining the dependencies.
4. The `copied_nodes` dictionary keeps track of original nodes and their corresponding copies.
5. The `dfs` function is a helper function that creates copies of nodes and their dependencies recursively.
6. The return statement at the end of the `dfs` function ensures that the copy of the current node and its neighbors is returned.
7. The final return statement constructs the list of copied nodes along with their

copied dependencies using a list comprehension.

Question 28

You are a software architect working on a project that involves developing multiple software modules. Each module has a list of dependencies that need to be satisfied before the module can be developed. You are given a list of module dependencies in the form

of pairs [module_a, module_b], which indicates that module module_a depends on module module_b.

Write a function can_develop_modules that takes the following parameters:

An integer totalModules representing the total number of software modules.

A list of pairs moduleDependencies, where each pair [module_a, module_b] indicates that module_a depends on module_b.

Your function should return True if it's possible to develop all the modules in such a way that all dependencies are satisfied, or False otherwise.

Examples:

Input:

totalModules = 4

moduleDependencies = [[1, 0], [2, 1], [3, 2]]

Output: True

Explanation:

Module 0 has no dependencies, so it can be developed.

Module 1 depends on module 0, so module 0 needs to be developed before module 1.

Module 2 depends on module 1, so module 1 needs to be developed before module 2.

Module 3 depends on module 2, so module 2 needs to be developed before module 3.

Therefore, all modules can be developed in the given order: 0 -> 1 -> 2 -> 3.

Input:

totalModules = 4

moduleDependencies = [[1, 0], [0, 2], [2, 1]]

Output: False

Explanation:

Module 0 depends on module 2, so module 2 needs to be developed before module 0.

Module 1 depends on module 0, so module 0 needs to be developed before module 1.

Module 2 depends on module 1, so module 1 needs to be developed before module 2.

This creates a circular dependency that cannot be resolved, so it's not possible to develop all modules.

Constraints:

- $1 \leq \text{totalModules} \leq 2000$
- $0 \leq \text{moduleDependencies.length} \leq 5000$
- $\text{moduleDependencies}[i].length == 2$
- $0 \leq \text{module_a}, \text{module_b} < \text{totalModules}$

- All the pairs `moduleDependencies[i]` are unique.

Solution:

```
1  class Solution:
2      def can_develop_modules(self, totalModules, moduleDependencies):
3          # Create an adjacency list to represent the dependencies
4          adjacency_list = [[] for _ in range(totalModules)]
5          for a, b in moduleDependencies:
6              adjacency_list[b].append(a)
7
8          # Function to check if a module has a cycle using DFS
9          def has_cycle(module, visited):
10              if visited[module] == 1:
11                  return True
12              visited[module] = 1
13              return any(has_cycle(dependency, visited) for dependency in adjacency_list[module])
14
15          # Check if there are any circular dependencies
16          return not any(has_cycle(module, [0] * totalModules) for module in range(totalModules))
```

Solution Explanation:

1. We create a class Solution to encapsulate the solution logic.

2. We define the `can_develop_modules` method that takes `totalModules` and `moduleDependencies` as inputs.
3. We create an `adjacency_list` to represent the dependencies between modules. For each pair $[a, b]$ in `moduleDependencies`, we add module a to the adjacency list of module b .
4. We define the `has_cycle` function to check if a module has a cycle using Depth-First Search (DFS). If a module is already visited (indicated by `visited[module] == 1`), it means we have encountered a cycle.
5. In the main part of the function, we iterate through each module and check if it has a cycle using the `has_cycle` function. If any module has a cycle, we return `False` (since it's not possible to develop all modules). Otherwise, we return `True`.
6. We create an instance of the `Solution` class and use it to test the function with the provided test cases.

Question 29

You are exploring a region with a unique landscape that includes mountains, valleys, and rivers. The landscape is represented as an $m \times n$ grid, where each cell contains an integer representing its elevation.

Your task is to identify specific cells where water can flow from higher elevations to lower elevations in both the north-south and east-west directions. A cell is considered to be a candidate if water can flow from it to both the northern ocean and the southern ocean, as well as from it to both the eastern ocean and the western ocean.

Write a function `find_water_flow_cells` that takes the following parameters:

An integer m representing the number of rows in the grid.

An integer n representing the number of columns in the grid.

A 2D list of integers `elevations`, where `elevations[r][c]` represents the elevation of the cell at coordinate (r, c) .

Your function should return a 2D list of cell coordinates ‘`result`’, where each element $[ri, ci]$ denotes a cell from which water can flow to all four oceans.

Examples:

Input:

$m = 5$

$n = 5$

```
elevations = [  
    [5, 6, 7, 8, 9],  
    [4, 5, 6, 7, 8],  
    [3, 4, 5, 6, 7],  
    [2, 3, 4, 5, 6],  
    [1, 2, 3, 4, 5]  
]
```

Output:

```
result = [[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]]
```

Explanation: The cells at these coordinates can flow water to all four oceans.

Input:

$m = 1$

$n = 1$

`elevations = [[10]]`

Output:

`result = [[0, 0]]`

Explanation: The only cell in the grid can flow water to all four oceans.

Constraints:

- $1 \leq m, n \leq 200$
- $0 \leq \text{elevations}[r][c] \leq 10^5$

Solution:

```
1  class WaterFlowCells:
2      def find_water_flow_cells(self, m, n, elevations):
3          if not elevations or m <= 0 or n <= 0:
4              return []
5
6          directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
7          result = []
8
9          def dfs(r, c, ocean_mask):
10             ocean_mask |= 1 << ocean
11             visited[r][c] |= ocean_mask
12
13             for dr, dc in directions:
14                 nr, nc = r + dr, c + dc
15                 if 0 <= nr < m and 0 <= nc < n and elevations[nr][nc] >= elevations[r][c]:
16                     if not (visited[nr][nc] & ocean_mask):
17                         dfs(nr, nc, ocean_mask)
18
19             for r in range(m):
20                 for c in range(n):
21                     visited = [[0 for _ in range(n)] for _ in range(m)]
22                     for ocean in range(4):
23                         dfs(r, c, 0)
24                         if visited[r][c] == 15: # 0b1111, indicating flow to all oceans
25                             result.append([r, c])
26
27             return result
```

Solution Explanation:

The WaterFlowCells class defines a method `find_water_flow_cells` that takes the grid dimensions `m`, `n`, and the elevations. It uses Depth-First Search (DFS) to traverse the grid from each cell and checks if water can flow to all four oceans (North, South, East, and West). The `directions` list represents the possible movement directions (up, down, left, right). The `dfs`

function is used to explore neighboring cells recursively, updating the visited cells with an ocean mask. Cells with an ocean mask of 0b1111 (15) indicate they can flow to all four oceans.

Question 30

You are given an $m \times n$ 2D grid representing a city map, where each cell can be a building ('B') or an

empty space ('E'). Buildings are represented by 'B', and empty spaces are represented by 'E'. The city is surrounded by an outer boundary formed by empty spaces.

A structure is a group of adjacent buildings connected horizontally or vertically. Two structures are considered distinct if they have different layouts or arrangements of buildings.

Write a function `count_unique_structures` that takes the following parameters:

An integer `m` representing the number of rows in the grid.

An integer `n` representing the number of columns in the grid.

A 2D list `grid` of size `m x n`, where each element `grid[i][j]` represents the content of the cell at coordinate (i, j) .

Your function should return the number of distinct structures in the city.

Examples:

Input:

$m = 4$

$n = 4$

grid = [

["B","B","E","E"],

["B","B","E","E"],

["E","E","B","E"],

```
["E","E","E","B"]  
]
```

Output: 2

Explanation: There are two distinct structures in the city: One with buildings in the top left corner (2x2 square) and one with a building in the bottom right corner.

Input:

$m = 3$

$n = 5$

grid = [

["E","E","B","B","E"],

["B","B","E","E","E"],

["B","B","E","E","E"]

]

Output: 3

Explanation: There are three distinct structures in the city: One with buildings on the left (2x3 rectangle), one with buildings on the right (2x2 square), and one with buildings in the middle (1x1 squares).

Constraints:

- $1 \leq m, n \leq 300$
- grid[i][j] is either 'B' or 'E'.

Solution:

```
1  class Solution:
2      def count_unique_structures(self, m, n, grid):
3          def dfs(i, j, structure_id):
4              if i < 0 or i >= m or j < 0 or j >= n or grid[i][j] != 'B':
5                  return
6              grid[i][j] = structure_id
7              dfs(i + 1, j, structure_id)
8              dfs(i - 1, j, structure_id)
9              dfs(i, j + 1, structure_id)
10             dfs(i, j - 1, structure_id)
11
12             structure_count = 0
13             for i in range(m):
14                 for j in range(n):
15                     if grid[i][j] == 'B':
16                         structure_count += 1
17                         dfs(i, j, structure_count)
18
19             return structure_count
```

Solution Explanation:

1. We define a class Solution that contains a function count_unique_structures to solve the problem.
2. In the count_unique_structures function, we define a nested function dfs that performs Depth-First Search to mark all connected buildings in the same structure with the same unique identifier (structure_id).
3. We initialize a structure_count variable to keep track of the number of distinct structures in the city.

4. We iterate through each cell in the grid. If a cell contains a building ('B'), we increment the structure_count and initiate a DFS from that cell to mark all connected buildings with the current structure_count as their identifier.
5. Finally, we return the structure_count.

Question 31

Given a list of integers ‘nums’, find the maximum length of a subsequence where the absolute differ-

ence between consecutive elements is exactly k.

Write a function `max_subsequence_length` that takes the following parameters:

A list of integers `nums`, where $0 \leq \text{nums.length} \leq 10^5$ and $-10^9 \leq \text{nums}[i] \leq 10^9$.

An integer `k` representing the required absolute difference between consecutive elements.

Your function should return an integer, the maximum length of a subsequence that satisfies the given condition.

Test Case Format:

Input:

A list of integers 'nums'.

An integer `k`.

Output:

An integer representing the maximum length of a subsequence.

Example 1:

Input: `nums = [1, 3, 7, 9]`, $k = 2$

Output: 4

Explanation: The subsequence `[1, 3, 7, 9]` satisfies the condition as the absolute difference between consecutive elements is 2. The maximum length of such a subsequence is 4.

Example 2:

Input: `nums = [5, 10, 15, 20]`, $k = 5$

Output: 2

Explanation: The subsequence `[5, 10]` satisfies the condition as the absolute difference between consecutive elements is 5.

utive elements is 5. The maximum length of such a subsequence is 2.

Example 3:

Input: `nums = [3, 8, 15, 30]`, `k = 7`

Output: 3

Explanation: The subsequence `[3, 8, 15]` satisfies the condition as the absolute difference between consecutive elements is 7. The maximum length of such a subsequence is 3.

Constraints:

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $0 \leq k \leq 10^9$

Solution:

```
1  class Solution:
2      def max_subsequence_length(self, nums, k):
3          num_set = set(nums)
4          max_length = 0
5
6          for num in num_set:
7              if num - k in num_set:
8                  length = 1
9                  current = num
10                 while current + k in num_set:
11                     current += k
12                     length += 1
13                 max_length = max(max_length, length)
14
15             return max_length
```

Solution Explanation:

1. We start by creating a set `num_set` from the input list `nums`. This will allow us to quickly check whether a number exists in the list or not.
2. We initialize `max_length` to 0, which will store the maximum length of the subsequence.
3. We iterate through each number '`num`' in the `num_set`.
4. For each number `num`, we check if `num - k` exists in the `num_set`. If it does, we know that we can potentially form a subsequence starting with `num - k` and ending with `num`. So, we initialize a length variable to 1.
5. We then start from `num` and keep adding `k` to it until `num + k` is in the `num_set`. This way, we can find the length of the subsequence that satisfies the condition.

6. We update max_length to the maximum of the current length and max_length.
7. Finally, we return the max_length as the result.

SECTION 5 - INTERVAL

Question 32

You are given a sorted list of meeting times, represented by start and end timestamps. Each meeting has a unique identifier, and you are also given a new meeting with a start and end time.

Incorporate the new meeting into the list, ensuring that it remains sorted by start time and that no meetings overlap. Return the list of meetings after the new meeting has been inserted.

Example:

Input: meetings = [(1, 3), (5, 7), (9, 11)], newMeeting = (4, 6)

Output: [(1, 3), (4, 7), (9, 11)]

Note:

The new meeting (4, 6) overlaps with the existing meeting (5, 7), so they are merged into a single meeting (4, 7).

Solution:

```
1  class MeetingScheduler:
2      def __init__(self):
3          self.meetings = []
4
5      def insert_meeting(self, new_meeting):
6          merged_meetings = []
7          i = 0
8
9          while i < len(self.meetings) and self.meetings[i][1] < new_meeting[0]:
10              merged_meetings.append(self.meetings[i])
11              i += 1
12
13          while i < len(self.meetings) and self.meetings[i][0] <= new_meeting[1]:
14              new_meeting = (min(new_meeting[0], self.meetings[i][0]),
15                             max(new_meeting[1], self.meetings[i][1]))
16              i += 1
17
18          merged_meetings.append(new_meeting)
19          merged_meetings.extend(self.meetings[i:])
20          self.meetings = merged_meetings
21
22      def get_sorted_meetings(self):
23          return self.meetings
```

Solution Explanation:

1. We define the `MeetingScheduler` class with an `__init__` method that initializes an empty list `self.meetings` to store the meetings.
2. The `insert_meeting` method is responsible for incorporating the new meeting into the list of meetings while maintaining the sorted order and merging any overlapping intervals.
3. The first while loop iterates through the existing meetings and adds meetings that occur before the start of the new meeting (based on the end timestamp of each existing meeting).
4. The second while loop merges overlapping intervals between the existing meetings and the new meeting. It adjusts the start and end times of the new meeting accordingly, and the loop continues until no more overlaps are found.

5. After handling the merging and insertion, the remaining meetings from the original list are added to merged_meetings using the extend method.
6. Finally, we update self.meetings with the merged_meetings list, effectively storing the sorted and merged list of meetings.
7. The get_sorted_meetings method simply returns the list of meetings in sorted order.
8. In the example usage, we create an instance of the MeetingScheduler class, insert the new meeting, and retrieve the sorted list of meetings using the get_sorted_meetings method.

Question 33

Given an array of time intervals, each represented as [start_time, end_time], find the longest continuous span of time that is covered by the intervals. Return

the start and end times of this longest continuous span.

Example:

Input: intervals = [[1,3],[6,9],[10,12],[14,16],[18,20]]

Output: [6,12]

Explanation: The longest continuous span covered by the intervals is [6,12], as intervals [6,9] and [10,12] overlap and can be merged into a single interval.

Input: intervals = [[1,5],[8,10],[12,14],[16,18],[20,22]]

Output: [8,18]

Explanation: The longest continuous span covered by the intervals is [8,18], as intervals [8,10], [12,14], and [16,18] overlap and can be merged into a single interval.

Constraints:

- $1 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals}[i].length = 2$
- $0 \leq \text{start_time} \leq \text{end_time} \leq 10^4$

Solution:

```
1  class LongestContinuousSpanFinder:
2      def __init__(self):
3          self.intervals = []
4
5      def add_interval(self, interval):
6          self.intervals.append(interval)
7
8      def find_longest_continuous_span(self):
9          self.intervals.sort()
10         merged_intervals = [self.intervals[0]]
11
12         for i in range(1, len(self.intervals)):
13             current_interval = self.intervals[i]
14             last_merged_interval = merged_intervals[-1]
15
16             if current_interval[0] <= last_merged_interval[1]:
17                 merged_intervals[-1] = [last_merged_interval[0],
18                                         max(last_merged_interval[1], current_interval[1])]
19             else:
20                 merged_intervals.append(current_interval)
21
22         longest_span = max(merged_intervals, key=lambda interval: interval[1] - interval[0])
23         return longest_span
```

Solution Explanation:

1. We define a class called LongestContinuousSpanFinder with three methods: `__init__`, `add_interval`, and `find_longest_continuous_span`.
2. The `__init__` method initializes an empty list `self.intervals` to store the intervals.
3. The `add_interval` method adds an interval to the list of intervals.
4. The `find_longest_continuous_span` method finds the longest continuous span of time covered by the intervals. It

first sorts the intervals based on their start times.

5. We iterate through the sorted intervals and maintain a list of merged intervals called merged_intervals. We compare the current interval with the last merged interval to check for overlap.
6. If there's an overlap, we update the last merged interval to cover the combined span. Otherwise, we add the current interval as a new merged interval.
7. After merging all overlapping intervals, we use the max function with a custom key function to find the longest span based on the difference between start and end times.
8. We return the longest span.
9. In the example usage, we create two instances of the LongestContinuousSpanFinder class and add intervals to them using the add_interval method. Then, we find the longest continuous spans

using the `find_longest_continuous_span` method and print the results.

Question 34

Given an array of tasks, each represented by a start time and end time, find the maximum number of non-overlapping tasks that can be performed. Tasks can't be performed simultaneously if their intervals overlap.

Example:

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

Output: 2

Explanation: The tasks [1,3] and [5,7] can be performed without overlapping.

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

Output: 4

Explanation: All tasks can be performed without overlapping.

Input: tasks = [[1,3],[2,5],[4,6]]

Output: 2

Explanation: The tasks [1,3] and [4,6] can be performed without overlapping.

Constraints:

- $1 \leq \text{tasks.length} \leq 10^5$
- $\text{tasks[i].length} == 2$
- $-5 * 10^4 \leq \text{start}_i < \text{end}_i \leq 5 * 10^4$

Solution

```
1  class TaskScheduler:
2      def __init__(self):
3          self.tasks = []
4
5      def add_task(self, task):
6          self.tasks.append(task)
7
8      def max_non_overlapping_tasks(self):
9          self.tasks.sort(key=lambda x: x[1]) # Sort tasks by end times
10         non_overlapping_count = 1
11         last_end_time = self.tasks[0][1]
12
13         for i in range(1, len(self.tasks)):
14             if self.tasks[i][0] >= last_end_time:
15                 non_overlapping_count += 1
16                 last_end_time = self.tasks[i][1]
17
18         return non_overlapping_count
```

Solution Explanation:

1. We define a class called TaskScheduler with three methods: `__init__`, `add_task`, and `max_non_overlapping_tasks`.
2. The `__init__` method initializes an empty list `self.tasks` to store the tasks.
3. The `add_task` method adds a task to the list of tasks.
4. The `max_non_overlapping_tasks` method finds the maximum number of non-overlapping tasks that can be performed. It first sorts the tasks based on their end

times using the sort method and a custom sorting key.

5. We iterate through the sorted tasks and count the number of tasks that can be performed without overlapping. We use the last_end_time variable to keep track of the end time of the last scheduled task.
6. If the start time of the current task is greater than or equal to the last_end_time, it means the task can be performed without overlapping, so we increment the count and update the last_end_time.
7. Finally, we return the count of non-overlapping tasks.
8. In the example usage, we create instances of the TaskScheduler class for each set of tasks and add the tasks using the add_task method. Then, we find the maximum non-overlapping tasks using the max_non_overlapping_tasks method and print the results.

SECTION 6 -

LINKED LIST

Question 35

You are given the head of a singly linked list and an integer k . Reverse every k nodes (where k is a positive integer) in the linked list and return the modified list.

Example:

Input: head = [1,2,3,4,5], $k = 2$

Output: [2,1,4,3,5]

Input: head = [1,2,3,4,5], k = 3

Output: [3,2,1,4,5]

Input: head = [1,2,3,4,5], k = 1

Output: [1,2,3,4,5]

Constraints:

- The number of nodes in the list is in the range [0, 5000].
- $-5000 \leq \text{Node.val} \leq 5000$
- $1 \leq k \leq \text{length of the linked list}$

Solution:

```
1  class ListNode:
2      def __init__(self, val=0, next=None):
3          self.val = val
4          self.next = next
5
6  class LinkedListReverser:
7      def reverse_k_group(self, head, k):
8          dummy = ListNode(0)
9          dummy.next = head
10         prev_group_end = dummy
11         curr = head
12         count = 0
13
14         while curr:
15             count += 1
16             if count % k == 0:
17                 prev_group_end, curr = self.reverse(
18                     prev_group_end, curr.next
19                     ), prev_group_end.next
20             else:
21                 curr = curr.next
22
23         return dummy.next
24
25     def reverse(self, prev_group_end, next_group_start):
26         prev, curr = prev_group_end.next, prev_group_end.next.next
27         prev.next = None
28
29         while curr != next_group_start:
30             curr.next, prev, curr = prev, curr, curr.next
31
32         prev_group_end.next.next, prev_group_end.next = next_group_start, prev
33
34     return prev_group_end
```

```
37 # Utility functions to convert between lists and linked lists
38 def list_to_linked_list(lst):
39     dummy = ListNode(0)
40     current = dummy
41
42     for val in lst:
43         current.next = ListNode(val)
44         current = current.next
45
46     return dummy.next
47
48 def linked_list_to_list(head):
49     lst = []
50     current = head
51
52     while current:
53         lst.append(current.val)
54         current = current.next
55
56     return lst
```

Solution Explanation:

1. The `reverse_k_group` method iterates through the linked list, maintaining a count of processed nodes. When the count becomes a multiple of k , it reverses the group and continues iterating.
2. The `reverse` method is responsible for reversing a group of nodes between `prev_group_end.next` and `next_group_start`. It uses a standard linked list reversal technique with three pointers (`prev`, `curr`, and `next`) to reverse the nodes.
3. The utility functions `list_to_linked_list` and `linked_list_to_list` convert between Python lists and linked lists.
4. The inputs list contains example inputs and values of k . The solution is tested on each input, and the result is printed.

This solution effectively reverses groups of k nodes in a linked list while maintaining the overall structure of the list.

Question 36

Given the head of a linked list and an integer k , determine if the linked list has a " k -cycle." A " k -cycle" in a linked list is a cycle where every k -th node can be reached again by following the next pointer k times.

Return true if there is a k -cycle in the linked list. Otherwise, return false.

Example:

Input: head = [1,2,3,4,5], k = 2

Output: true

Explanation: There is a 2-cycle in the linked list, where every 2nd node can be reached again by following the next pointer 2 times.

Input: head = [1,2,3,4,5], k = 3

Output: false

Explanation: There is no 3-cycle in the linked list.

Input: head = [1,2,3,4,5], k = 1

Output: true

Explanation: Every node is a 1-cycle itself.

Constraints:

- The number of nodes in the list is in the range $[0, 10^4]$.
- $-10^5 \leq \text{Node.val} \leq 10^5$
- $1 \leq k \leq \text{length of the linked list}$

Solution:


```
31 # Utility function to convert a list to a linked list
32 def list_to_linked_list(lst):
33     dummy = ListNode(0)
34     current = dummy
35
36     for val in lst:
37         current.next = ListNode(val)
38         current = current.next
39
40     return dummy.next
```

Solution Explanation:

1. We define a `ListNode` class to represent the nodes of the linked list, and a `LinkedListCycleChecker` class to implement the cycle checking logic.
2. The `has_k_cycle` method takes the head of the linked list and an integer k as inputs. It checks if there is a " k -cycle" in the linked list as described in the problem statement.
3. We use the slow and fast pointer approach to detect cycles. The fast pointer moves ahead by k nodes, and then both slow and fast pointers move one node at a time. If there is a cycle, the slow and fast pointers will eventually meet.
4. If there is a cycle, the method returns `True`; otherwise, it returns `False`.
5. In the example usage section, we create instances of the `LinkedListCycleChecker` class and use the `has_k_cycle` method to

check for k-cycles in the given linked lists.
We print the results.

This solution effectively checks for a "k-cycle" in a linked list using the slow and fast pointer technique, ensuring efficient detection of cycles.

Question 37

You are given the heads of two linked lists, list1 and list2, where each node contains a positive integer value.

Merge the two lists into a single linked list in a special pattern. Starting with the first node of list1, then the first node of list2, then the second node of list1, then the second node of list2, and so on. If one list

is longer than the other, the remaining nodes should be appended to the end of the merged list.

Return the head of the merged linked list.

Example:

Input: list1 = [1,3,5], list2 = [2,4,6]

Output: [1,2,3,4,5,6]

Input: list1 = [1,2,3,4], list2 = [5,6]

Output: [1,5,2,6,3,4]

Input: list1 = [1,4,5], list2 = [2,3,6,7]

Output: [1,2,4,3,5,6,7]

Constraints:

- The number of nodes in both lists is in the range [0, 50].
- $1 \leq \text{Node.val} \leq 100$

- Both list1 and list2 are sorted in non-decreasing order.

Solution:

```
1  class ListNode:
2      def __init__(self, val=0, next=None):
3          self.val = val
4          self.next = next
5
6  class LinkedListMerger:
7      def merge_special_pattern(self, list1, list2):
8          dummy = ListNode()           # Create a dummy node as the starting point
9          current = dummy            # Pointer to the current node
10
11         while list1 or list2:
12             if list1:
13                 current.next = list1
14                 list1 = list1.next
15                 current = current.next
16
17             if list2:
18                 current.next = list2
19                 list2 = list2.next
20                 current = current.next
21
22         return dummy.next
23
24     # Utility function to convert a list to a linked list
25     def list_to_linked_list(lst):
26         dummy = ListNode()
27         current = dummy
28
29
30         for val in lst:
31             current.next = ListNode(val)
32             current = current.next
33
34     return dummy.next
35
36     # Utility function to convert a linked list to a list
37     def linked_list_to_list(head):
38         lst = []
39         current = head
40
41         while current:
42             lst.append(current.val)
43             current = current.next
44
45     return lst
```

Solution Explanation:

1. We define a `ListNode` class to represent the nodes of the linked list, and a `LinkedListMerger` class to implement the merging logic.
2. The `merge_special_pattern` method takes the heads of two linked lists, `list1` and `list2`, as inputs. It merges the two lists into a special pattern as described in the problem.
3. We create a dummy node and a current pointer to keep track of the merged list.

4. We iterate through both list1 and list2, appending nodes alternatively to the merged list.
5. Once we exhaust either of the input lists, we simply append the remaining nodes from the other list to the merged list.
6. The merged list is constructed using the next pointers of the nodes.
7. In the example usage section, we create instances of the LinkedListMerger class and use the merge_special_pattern method to merge the given linked lists. We then convert the merged linked lists to lists using utility functions and print the results.

This solution effectively merges two linked lists into a special pattern while maintaining the order of nodes and efficiently splicing them together.

Question 38

You are given an array of linked-lists, where each linked-list contains positive integer values.

Rearrange the nodes of the linked-lists in a cyclic pattern, as follows: Starting with the first node of the first linked-list, then the first node of the second linked-list, then the second node of the first linked-list, then the second node of the second linked-list, and so on. If one linked-list is longer than the others, the remaining nodes should be appended to the end of the rearranged list.

Return the head of the rearranged linked list.

Example:

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

Output: [1, 2, 1, 4, 3, 7, 5, 6]

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

Output: [1, 2, 3]

Input: lists = [[2, 4, 6], [1, 3], [5]]

Output: [2, 1, 5, 4, 3, 6]

Constraints:

- The number of linked-lists, k, is in the range $[0, 10^4]$.
- $0 \leq \text{lists}[i].\text{length} \leq 500$
- $1 \leq \text{Node.val} \leq 10^4$
- $\text{lists}[i]$ is sorted in ascending order.
- The sum of the lengths of all linked-lists, $\text{sum}(\text{lists}[i].\text{length})$, will not exceed 10^4 .

Solution:

```
1  class ListNode:
2      def __init__(self, val=0, next=None):
3          self.val = val
4          self.next = next
5
6  class LinkedListCycler:
7      def rearrange_cyclic_pattern(self, lists):
8          if not lists:
9              return None
10
11         merged = []
12         max_length = 0
13
14         for lst in lists:
15             current = lst
16             while current:
17                 merged.append(current)
18                 current = current.next
19                 max_length = max(max_length, len(merged))
20
21         for i in range(max_length - 1):
22             merged[i].next = merged[i + 1]
23         merged[max_length - 1].next = None
24
25         return merged[0]
```

```
28 # Utility function to convert a list to a linked list
29 def list_to_linked_list(lst):
30     dummy = ListNode()
31     current = dummy
32
33     for val in lst:
34         current.next = ListNode(val)
35         current = current.next
36
37     return dummy.next
38
39 # Utility function to convert a linked list to a list
40 def linked_list_to_list(head):
41     lst = []
42     current = head
43
44     while current:
45         lst.append(current.val)
46         current = current.next
47
48     return lst
```

Solution Explanation:

1. We define a `ListNode` class to represent the nodes of the linked list, and a `LinkedListCycler` class to implement the cyclic rearranging logic.
2. The `rearrange_cyclic_pattern` method takes a list of linked-lists, `lists`, as input. It rearranges the nodes of the linked-lists in a cyclic pattern as described in the problem.
3. We iterate through each linked-list in `lists`, appending the nodes to the merged list. We also keep track of the maximum length of the merged list.
4. We then iterate through the merged list and adjust the next pointers to create the cyclic pattern.
5. In the example usage section, we create instances of the `LinkedListCycler` class and use the `rearrange_cyclic_pattern` method to rearrange the nodes of the

given linked-lists. We then convert the rearranged linked lists to lists using utility functions and print the results.

This solution effectively rearranges the nodes of the linked-lists in a cyclic pattern while maintaining the order of nodes and efficiently splicing them together.

Question 39

Given the head of a linked list, reverse every alternate group of n nodes in the list and return its head.

Example:

Input: head = [1,2,3,4,5,6,7,8,9], n = 3

Output: [3,2,1,4,5,6,9,8,7]

Explanation: The alternate groups of 3 nodes are [1,2,3], [4,5,6], and [7,8,9]. Each of these groups is reversed.

Input: head = [1,2,3,4,5], n = 2

Output: [2,1,4,3,5]

Explanation: The alternate groups of 2 nodes are [1,2], [3,4], and [5]. Each of these groups is reversed.

Input: head = [1], n = 1

Output: [1]

Explanation: There's only one node, so no change is needed.

Constraints:

- The number of nodes in the list is sz.
- $1 \leq sz \leq 30$
- $0 \leq Node.val \leq 100$
- $1 \leq n \leq sz$

Solution:

```
1  class ListNode:
2      def __init__(self, val=0, next=None):
3          self.val = val
4          self.next = next
5
6  class LinkedListAlternateReverser:
7      def reverse_alternate_groups(self, head, n):
8          if not head or n <= 1:
9              return head
10
11         dummy = ListNode()
12         dummy.next = head
13         prev = dummy
14
15         while prev.next:
16             first, second = prev.next, prev.next.next
17             for _ in range(n):
18                 if not second:
19                     break
20                 second = second.next
21
22             if second:
23                 prev.next, prev, second.next = second, first, None
24                 first, second = prev.next, prev.next
25
26             for _ in range(n - 1):
27                 prev.next, first.next, second.next = second, second.next, first
28                 first, second = second, first.next
29
30         return dummy.next
```

```
33 # Utility function to convert a list to a linked list
34 def list_to_linked_list(lst):
35     dummy = ListNode()
36     current = dummy
37
38     for val in lst:
39         current.next = ListNode(val)
40         current = current.next
41
42     return dummy.next
43
44 # Utility function to convert a linked list to a list
45 def linked_list_to_list(head):
46     lst = []
47     current = head
48
49     while current:
50         lst.append(current.val)
51         current = current.next
52
53     return lst
```

Solution Explanation:

1. The reverse_alternate_groups method is similar to the previous solution, but we've combined some assignments and loops to make the code more concise.
2. We use two pointers first and second to iterate through the linked list. The first pointer points to the first node of the current group, and the second pointer is advanced n nodes ahead.
3. We update the prev.next pointer to point to second to reverse the group. Then, we rearrange the next pointers to reverse the nodes within the group.
4. This approach helps in reducing redundant assignments and loop iterations, making the solution shorter.

This solution effectively reverses every alternate group of n nodes in the linked list while maintaining the order of nodes and efficiently updating the next pointers.

Question 40

You are given the head of a singly linked-list. The list can be represented as:

$L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$

Modify the list such that every odd-indexed node is moved to the end while maintaining the relative order of the nodes.

Example:

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

Output: [2,4,1,3]

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

Output: [2,4,1,5,3]

Constraints:

- The number of nodes in the list is in the range $[1, 5 * 10^4]$.
- $1 \leq \text{Node.val} \leq 1000$

Solution:

```
1  class ListNode:
2      def __init__(self, val=0, next=None):
3          self.val = val
4          self.next = next
5
6  class LinkedListModifier:
7      def modify_list(self, head):
8          if not head or not head.next:
9              return head
10
11         odd_head, even_head = ListNode(), ListNode()
12         odd, even, current, is_odd = odd_head, even_head, head, True
13
14         while current:
15             (odd if is_odd else even).next, current = current, current.next
16             is_odd = not is_odd
17
18         even.next = None
19         odd.next = even_head.next
20
21         return odd_head.next
22
23     # Utility function to convert a list to a linked list
24     def list_to_linked_list(lst):
25         dummy = ListNode()
26         current = dummy
27
28
29         for val in lst:
30             current.next = ListNode(val)
31             current = current.next
32
33
34         return dummy.next
35
36     # Utility function to convert a linked list to a list
37     def linked_list_to_list(head):
38         lst = []
39         current = head
40
41         while current:
42             lst.append(current.val)
43             current = current.next
44
45         return lst
```

Solution Explanation:

1. The `modify_list` method is shortened by combining assignments and loop iterations.
2. We use a single `odd_head` and `even_head` dummy nodes to separate odd-indexed and even-indexed nodes.
3. In the loop, we update the odd and even pointers directly within a single line of code based on the `is_odd` condition.
4. After processing all nodes, we connect the even-indexed list to the odd-indexed list as before.

This solution effectively modifies the linked list by moving every odd-indexed node to the end while maintaining the relative order of the nodes.

SECTION 7 -

MATRIX

Question 41

Given an $m \times n$ integer matrix `matrix`, implement an algorithm to replace all non-zero elements in the matrix with the sum of their respective row and column indices.

You must do it in place.

Example:

Input: `matrix = [[1,2,3],[4,5,6],[7,8,9]]`

Output: [[5,5,5],[9,9,9],[13,13,13]]

Constraints:

- $m == \text{matrix.length}$
- $n == \text{matrix[0].length}$
- $1 \leq m, n \leq 200$
- $-2^{31} \leq \text{matrix}[i][j] \leq 2^{31} - 1$

Solution:

```
1  class MatrixProcessor:
2      def __init__(self, matrix):
3          self.matrix = matrix
4          self.rows = len(matrix)
5          self.cols = len(matrix[0])
6
7      def process_matrix(self):
8          for i in range(self.rows):
9              for j in range(self.cols):
10                  if self.matrix[i][j] != 0:
11                      self.matrix[i][j] = self.calculate_sum(i, j)
12
13     def calculate_sum(self, row, col):
14         row_sum = sum(self.matrix[row])
15         col_sum = sum(self.matrix[i][col] for i in range(self.rows))
16         return row_sum + col_sum - self.matrix[row][col]
17
18     def print_matrix(matrix):
19         for row in matrix:
20             print(row)
```

Solution Explanation:

1. We define a class `MatrixProcessor` to encapsulate the matrix processing functionality. It takes the input matrix as a parameter during initialization and stores information about the number of rows and columns.
2. The `process_matrix` method iterates through each element of the matrix. If the element is non-zero, it replaces the element with the sum of its respective row and column indices using the `calculate_sum` method.
3. The `calculate_sum` method calculates the sum of all elements in the given row and column, excluding the current element, and returns the sum.
4. The `print_matrix` function is used to print a matrix row by row.
5. In the example usage, we create a `MatrixProcessor` object with the input matrix,

process the matrix using the process_matrix method, and then print the input and output matrices to see the transformation.

The solution iterates through the matrix once and calculates the sum of each element's row and column in constant time. Therefore, the time complexity of this solution is $O(m * n)$, where m is the number of rows and n is the number of columns in the matrix.

Question 42

Given a square matrix of size $n \times n$, where each element represents a city and the value at $\text{matrix}[i][j]$ indicates the distance between city i and city j , design an algorithm to find the shortest path that visits all cities exactly once and returns to the starting

city. Return the sequence of cities visited in the order they are traversed.

Example:

Input: matrix = [[0, 29, 20, 21],
[29, 0, 15, 17],
[20, 15, 0, 28],
[21, 17, 28, 0]]

Output: [0, 2, 1, 3, 0]

Constraints:

- $n == \text{matrix.length}$
- $n == \text{matrix[i].length}$
- $1 <= n <= 10$
- $0 <= \text{matrix[i][j]} <= 100$

Solution:

```
1  class TravelingSalesman:
2      def __init__(self, matrix):
3          self.matrix = matrix
4          self.n = len(matrix)
5          self.visited = [False] * self.n
6          self.path = []
7          self.min_distance = float('inf')
8
9      def solve(self):
10         self._solve_tsp(0, [0], 0)
11
12     def _solve_tsp(self, city, current_path, current_distance):
13         if len(current_path) == self.n and self.matrix[city][0] > 0:
14             current_distance += self.matrix[city][0]
15             if current_distance < self.min_distance:
16                 self.min_distance = current_distance
17                 self.path = current_path[:]
18             return
19
20         for next_city in range(self.n):
21             if not self.visited[next_city] and self.matrix[city][next_city] > 0:
22                 self.visited[next_city] = True
23                 self._solve_tsp(next_city, current_path + [next_city],
24                                current_distance + self.matrix[city][next_city])
25                 self.visited[next_city] = False
26
27     def find_shortest_path(matrix):
28         tsp_solver = TravelingSalesman(matrix)
29         tsp_solver.solve()
30         return tsp_solver.path + [0]
```

Solution Explanation:

1. We define a class `TravelingSalesman` to encapsulate the traveling salesman problem-solving functionality. It takes the input matrix as a parameter during initialization and stores information about the number of cities (n), visited cities, the current path being explored, and the minimum distance found so far.
2. The `solve` method initiates the solving process by calling the private `_solve_tsp` method starting from the first city (index 0).
3. The `_solve_tsp` method is a recursive backtracking function that explores all possible paths from the current city to unvisited cities. It keeps track of the current path and current distance traveled. If a valid complete path is found (visiting all cities and returning to the starting city),

it updates the minimum distance and the optimal path.

4. The `find_shortest_path` function initializes the `TravelingSalesman` solver, calls the `solve` method, and returns the shortest path found.
5. In the example usage, we provide the input matrix and use the `find_shortest_path` function to obtain the shortest path that visits all cities and returns to the starting city.

The solution uses backtracking to explore all possible paths and find the optimal solution. Since the number of cities is limited to 10 (as per the constraints), the time complexity of the solution is manageable.

Question 43

You are given a grid of size $n \times n$, representing a puzzle with colored cells. The puzzle is solved by merging adjacent cells of the same color to form larger clusters. Implement an algorithm to find the final state of the puzzle after merging all possible clusters.

You must modify the input grid in-place.

Example:

Input:

```
grid = [[0, 2, 2, 0],  
        [2, 4, 8, 2],  
        [4, 16, 8, 4],  
        [0, 4, 4, 0]  
    ]
```

Output:

```
grid = [[0, 0, 0, 0],  
        [0, 2, 4, 2],  
        [0, 16, 8, 4],  
        [0, 0, 0, 0]  
    ]
```

Constraints:

- $n == \text{grid.length} == \text{grid[i].length}$
- $1 \leq n \leq 20$
- $0 \leq \text{grid[i][j]} \leq 1000$

Solution:

```
1 class PuzzleSolver:
2     def __init__(self, grid):
3         self.grid = grid
4         self.n = len(grid)
5         self.directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
6
7     def solve(self):
8         for i in range(self.n):
9             for j in range(self.n):
10                 if self.grid[i][j] != 0:
11                     self.merge_cluster(i, j, self.grid[i][j])
12
13     def merge_cluster(self, row, col, color):
14         if not (0 <= row < self.n and 0 <= col < self.n) or self.grid[row][col] != color:
15             return
16
17         self.grid[row][col] = 0
18         for direction in self.directions:
19             newRow, newCol = row + direction[0], col + direction[1]
20             self.merge_cluster(newRow, newCol, color)
```

Solution Explanation:

1. The `PuzzleSolver` class remains mostly the same, but we've simplified the code within the methods.
2. In the `solve` method, we iterate through each cell in the grid. If a cell is non-zero, we call the `merge_cluster` method to merge adjacent cells of the same color.
3. In the `merge_cluster` method, we recursively traverse in all four directions and merge adjacent cells with the same color by setting their values to 0.

The solution uses DFS to merge clusters of cells with the same color. Since each cell is visited at most once, the time complexity of this solution is $O(n^2)$, where n is the size of the grid.

Question 44

Given a board of size $m \times n$, where each cell contains a positive integer, design an algorithm to find a path

that starts at the top-left corner and ends at the bottom-right corner, maximizing the product of numbers along the path.

Return the maximum product achievable along such a path.

Example:

Input: board = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Output: 2016 (1 * 4 * 7 * 8 * 9)

Constraints:

- $m == \text{board.length}$
- $n == \text{board}[i].length$
- $1 \leq m, n \leq 6$
- $1 \leq \text{board}[i][j] \leq 15$

Solution:

```
1  class MaxProductPath:
2      def __init__(self, board):
3          self.board = board
4          self.m = len(board)
5          self.n = len(board[0])
6          self.max_product = float('-inf')
7          self.visited = set()
8
9      def solve(self):
10         self.dfs(0, 0, self.board[0][0])
11         return self.max_product
12
13     def dfs(self, row, col, current_product):
14         if (row, col) in self.visited:
15             return
16
17         if row == self.m - 1 and col == self.n - 1:
18             self.max_product = max(self.max_product, current_product)
19             return
20
21         self.visited.add((row, col))
22
23         if row + 1 < self.m:
24             self.dfs(row + 1, col, current_product * self.board[row + 1][col])
25
26         if col + 1 < self.n:
27             self.dfs(row, col + 1, current_product * self.board[row][col + 1])
28
29         self.visited.remove((row, col))
```

Solution Explanation:

1. We define a class `MaxProductPath` to encapsulate the maximum product path-solving functionality. It takes the input board as a parameter during initialization and stores information about the board's dimensions, visited cells, and the maximum product found.
2. The `solve` method initiates the solving process by calling the private `dfs` method starting from the top-left corner of the board.
3. The `dfs` method performs depth-first search (DFS) to explore all possible paths from the current cell. It calculates the current product of the path and updates the `max_product` if a valid path to the bottom-right corner is found.
4. The `visited` set is used to keep track of visited cells to avoid revisiting the same cell in the same path.

5. In the example usage, we create a MaxProductPath object with the input board, solve the problem using the solve method, and then print the maximum product achievable along the path.

The solution uses DFS to explore all possible paths from the top-left corner to the bottom-right corner of the board. Since each cell is visited at most once, the time complexity of this solution is $O(m * n)$, where m is the number of rows and n is the number of columns in the board.

SECTION 8 -

STRING

Question 45

Given a string s , find the length of the shortest substring containing all unique characters at least twice.

Example 1:

Input: $s = \text{"abacbc"}$

Output: 3

Explanation: The answer is "aba", with the length of 3, which contains all unique characters at least twice (a and b).

Example 2:

Input: s = "xyxy"

Output: 2

Explanation: The answer is "xy", with the length of 2, which contains all unique characters at least twice (x and y).

Example 3:

Input: s = "abcde"

Output: 0

Explanation: There is no substring that contains all unique characters at least twice.

Constraints:

- $0 \leq s.length \leq 5 * 10^4$
- s consists of English letters, digits, symbols, and spaces.

Solution:

```
1  class ShortestSubstring:
2      def shortestSubstringLength(self, s: str) -> int:
3          n = len(s)
4          char_indices = {} # To store the last index of each character seen
5          min_length = float(
6              'inf'
7          ) # Initialize the minimum length to positive infinity
8
9          left = 0 # Initialize the left pointer
10         for right in range(n):
11             char = s[right]
12             if char in char_indices:
13                 left = max(
14                     left, char_indices[char] + 1
15                 ) # Update the left pointer to skip repeated character
16             char_indices[
17                 char
18             ] = right # Update the index of the current character
19             if right - left + 1 >= 2:
20                 min_length = min(
21                     min_length, right - left + 1
22                 ) # Update the minimum length
23
24         return min_length if min_length != float('inf') else 0
```

Solution Explanation:

1. We define a class `ShortestSubstring` with a method `shortestSubstringLength` that takes a string `s` as input and returns the length of the shortest substring containing all unique characters at least twice.
2. We initialize a dictionary `char_indices` to store the last index of each character seen in the string.
3. We initialize `min_length` to positive infinity. This variable will be used to keep track of the minimum length of the substring containing all unique characters at least twice.
4. We initialize the left pointer to 0 and iterate through the string using the right pointer.
5. For each character `char` at index `right`, if it is already present in `char_indices`, we update the left pointer to skip

the repeated character by setting it to `char_indices[char] + 1`.

6. We update the index of the current character in `char_indices`.
7. If the length of the current substring (`right - left + 1`) is greater than or equal to 2, we update the `min_length` with the minimum of its current value and the length of the current substring.
8. Finally, we return the `min_length` if it is not equal to positive infinity (indicating a valid substring) or 0 if no valid substring is found.
9. We test the class with the provided examples and print the outputs.

This solution uses a sliding window approach to find the shortest substring containing all unique characters at least twice. The `char_indices` dictionary helps in efficiently updating the left pointer to skip repeated characters and find valid substrings.

Question 46

You are given a string s and an integer k . You can choose any character of the string and change its case (uppercase to lowercase or vice versa). You can perform this operation at most k times.

Return the length of the longest substring containing the same characters (ignoring case) you can get after performing the above operations.

Example 1:

Input: $s = "aAabB"$, $k = 2$

Output: 3

Explanation: Change 'A' and 'B' to 'a' and 'b', resulting in "aabb".

Example 2:

Input: s = "ABCdefgabc", k = 3

Output: 7

Explanation: Change 'A', 'B', and 'C' to 'a', 'b', and 'c', resulting in "abcdefgabc".

The substring "abcdefg" has the longest repeating letters (ignoring case), which is 7.

There may exist other ways to achieve this answer too.

Constraints:

- $1 \leq s.length \leq 10^5$
- s consists of only uppercase and lowercase English letters.
- $0 \leq k \leq s.length$

Solution:

```
1  class LongestSubstringWithCaseChange:
2      def longestSubstring(self, s: str, k: int) -> int:
3          n = len(s)
4          char_count = [0] * 26 # Initialize an array to store character counts
5          max_length = 0 # Initialize the maximum substring length
6          left = 0 # Initialize the left pointer
7
8          for right in range(n):
9              char_count[
10                  ord(s[right].lower()) - ord('a')
11              ] += 1 # Count characters ignoring case
12
13          # Calculate the total characters to be changed
14          total_changes = (right - left + 1) - max(char_count)
15
16          if total_changes > k: # Check if total changes exceed the allowed limit
17              char_count[
18                  ord(s[left].lower()) - ord('a')
19              ] -= 1 # Decrease the count of left character
20              left += 1 # Move the left pointer
21
22          max_length = max(
23              max_length, right - left + 1
24          ) # Update the maximum length
25
26      return max_length
```

Solution Explanation:

1. We define a class `LongestSubstringWithCaseChange` with a method `longestSubstring` that takes a string s and an integer k as input and returns the length of the longest substring containing the same characters (ignoring case) after performing the case change operation at most k times.
2. We initialize an array `char_count` of size 26 to store the count of characters. The index i in this array represents the character '`a`' + i (ignoring case).
3. We initialize the `max_length` to keep track of the maximum substring length.
4. We initialize the left pointer to 0 and iterate through the string using the right pointer.
5. For each character at index right , we increment the corresponding count in

`char_count` array, considering the lower-case form of the character.

6. We calculate the total number of characters that need to be changed to make all characters in the current substring the same. This is done by subtracting the maximum count of any character from the length of the current substring.
7. If the total changes exceed the allowed limit k , we need to move the left pointer to shrink the window. We decrease the count of the character at the left index and increment the left pointer.
8. We update the `max_length` with the maximum of its current value and the length of the current substring.
9. Finally, we return the `max_length`.
10. We test the class with the provided examples and print the outputs.

This solution uses a sliding window approach to find the longest substring containing the same char-

acters (ignoring case) by optimizing the number of allowed case changes. The `char_count` array helps in efficiently counting characters, and the `left` pointer is adjusted to keep the total changes within the allowed limit.

Question 47

Given two strings s and t of lengths m and n respectively, find the maximum window substring of

s such that no character in t is included in the window. If there is no such substring, return the empty string "".

The testcases will be generated such that the answer is unique.

Example 1:

Input: $s = \text{"ADOBECODEBANC"}$, $t = \text{"ABC"}$

Output: "DOECODEBA"

Explanation: The maximum window substring "DOECODEBA" excludes 'A', 'B', and 'C' from string t .

Example 2:

Input: $s = \text{"a"}$, $t = \text{"a"}$

Output: ""

Explanation: There is no valid window substring that excludes the character 'a' from t .

Example 3:

Input: s = "a", t = "aa"

Output: "a"

Explanation: The entire string s is the maximum window excluding the 'a's from t.

Constraints:

- $m == s.length$
- $n == t.length$
- $1 \leq m, n \leq 10^5$
- s and t consist of uppercase and lowercase English letters.

Solution:

```
1 class MaximumWindowSubstring:
2     def maxWindowSubstring(self, s: str, t: str) -> str:
3         char_count = [0] * 128 # Initialize an array to store character counts
4         t_set = set(t) # Create a set of characters from t
5         left = 0 # Initialize the left pointer
6         max_length = 0 # Initialize the maximum window length
7         max_window = "" # Initialize the maximum window substring
8
9         for right in range(len(s)):
10             char_count[
11                 ord(s[right])
12             ] += 1 # Increment the count of the current character
13
14             while all(char_count[ord(c)] >= 0 for c in t_set):
15                 if right - left + 1 > max_length:
16                     max_length = right - left + 1 # Update the maximum window length
17                     max_window = s[
18                         left:right+1
19                     ] # Update the maximum window substring
20
21             char_count[
22                 ord(s[left])
23             ] -= 1 # Decrease the count of the character at the left pointer
24             left += 1 # Move the left pointer
25
26         return max_window
```

Solution Explanation:

1. We define a class MaximumWindowSubstring with a method maxWindowSubstring that takes two strings s and t as input and returns the maximum window substring of s such that no character in t is included in the window.
2. We initialize an array char_count of size 128 to store the count of characters. The index i in this array represents the character with ASCII value i.
3. We create a set t_set containing characters from string t.
4. We initialize the left pointer to 0 and iterate through the string using the right pointer.
5. For each character at index right, we increment the corresponding count in the char_count array.
6. We enter a while loop as long as all characters in t_set are present in the current

window. This ensures that we exclude characters from t in the window.

7. Inside the while loop, we check if the current window length is greater than the maximum window length. If it is, we update the maximum window length and the maximum window substring.
8. We decrease the count of the character at the left pointer and move the left pointer to shrink the window.
9. Finally, we return the max_window containing the maximum window substring.
10. We test the class with the provided examples and print the outputs.

This solution uses a sliding window approach to find the maximum window substring of s such that no character in t is included in the window. The char_count array helps in efficiently counting characters, and the left pointer is adjusted to exclude characters from t in the window.

Question 48

Given two strings s and t , determine if t can be formed by deleting exactly one character from s .

Example 1:

Input: s = "leetcode", t = "letcode"

Output: true

Explanation: The string t can be formed by deleting the character 'o' from the string s.

Example 2:

Input: s = "hello", t = "he~~l~~o"

Output: true

Explanation: The string t can be formed by deleting the character 'l' from the string s.

Example 3:

Input: s = "intention", t = "execution"

Output: false

Explanation: The string t cannot be formed by deleting exactly one character from the string s.

Constraints:

- $1 \leq s.length, t.length \leq 5 * 10^4$
- s and t consist of lowercase English letters.

Solution:

```
1  class OneCharacterDeletion:
2      def isOneCharacterDeletion(self, s: str, t: str) -> bool:
3          m, n = len(s), len(t)
4
5          if abs(m - n) != 1:
6              return False
7
8          i, j = 0, 0
9
10         while i < m and j < n:
11             if s[i] != t[j]:
12                 if i != j:
13                     return False
14                 if m > n:
15                     i += 1
16                 else:
17                     j += 1
18             else:
19                 i += 1
20                 j += 1
21
22         return True
```

Solution Explanation:

1. We define a class `OneCharacterDeletion` with a method `isOneCharacterDeletion` that takes two strings `s` and `t` as input and returns `True` if `t` can be formed by deleting exactly one character from `s`, otherwise returns `False`.
2. We first check if the absolute difference between the lengths of `s` and `t` is not equal to 1. If it's not, we return `False` immediately since it's not possible to form `t` by deleting one character from `s`.
3. We initialize two pointers `i` and `j` to traverse `s` and `t` respectively.
4. We iterate through both strings using the pointers `i` and `j`.
5. If the characters at the current positions `i` and `j` are not equal, we check if the pointers `i` and `j` are also not equal. If they are, it means we have encountered more than one mismatch, and we return `False`.

6. If the lengths of s and t are different, we increment i (if s is longer) or j (if t is longer) to simulate the deletion of a character from the longer string.
7. If the characters at the current positions i and j are equal, we increment both i and j.
8. After iterating through both strings, if we reach the end of either string, it means that the last character in that string could be deleted to match the other string, thus forming t by deleting one character from s. In this case, we return True.
9. We test the class with the provided examples and print the outputs.

This solution efficiently checks whether t can be formed by deleting exactly one character from s by iteratively comparing characters and adjusting pointers. It handles both cases where s is longer or shorter than t, and returns the appropriate result accordingly.

Question 49

Given an array of strings `strs`, group the strings together based on a custom rule. Two strings are con-

sidered to be in the same group if they have the same characters at even indices.

Example 1:

Input: `strs = ["hello", "letlo", "world", "wodlr"]`

Output: `[["hello", "letlo"], ["world", "wodlr"]]`

Explanation: The strings "hello" and "letlo" have the same characters at even indices (h, l, o) and (l, e, l), so they are in the same group. Similarly, the strings "world" and "wodlr" have the same characters at even indices (w, r, d) and (o, d, r), so they are in the same group.

Example 2:

Input: `strs = ["apple", "pleap", "banana", "naanab"]`

Output: `[["apple", "pleap"], ["banana", "naanab"]]`

Explanation: The strings "apple" and "pleap" have the same characters at even indices (a, p) and (p, a), so

they are in the same group. Similarly, the strings "banana" and "naanab" have the same characters at even indices (b, n, n) and (a, a, a), so they are in the same group.

Example 3:

Input: `strs = ["car","rac","truck","krutc"]`

Output: `[["car","rac"],["truck","krutc"]]`

Explanation: The strings "car" and "rac" have the same characters at even indices (c, r) and (r, a), so they are in the same group. Similarly, the strings "truck" and "krutc" have the same characters at even indices (t, u, c) and (k, r, c), so they are in the same group.

Constraints:

- $1 \leq \text{strs.length} \leq 10^4$
- $0 \leq \text{strs[i].length} \leq 100$
- `strs[i]` consists of lowercase English letters.

Solution:

```
1 class CustomGrouping:
2     def groupStrings(self, strs):
3         groups = {}
4         for s in strs:
5             key = tuple([(ord(s[i]) -
6                         ord(s[0])) % 26 for i in range(len(s)) if i % 2 == 0])
7             if key not in groups:
8                 groups[key] = []
9                 groups[key].append(s)
10
11     return list(groups.values())
```

Solution Explanation:

1. We define a class `CustomGrouping` with a method `groupStrings` that takes a list of strings `strs` as input and returns a list of groups where two strings are considered to be in the same group if they have the same characters at even indices (0-indexed).
2. We initialize an empty dictionary `groups` to store the groups.
3. We iterate through each string `s` in `strs`.
4. For each string `s`, we calculate a key by computing the difference of character ASCII values between each character and the first character of the string. We take the modulo 26 to handle the circular nature of letters in the English alphabet. We only consider characters at even indices (0, 2, 4, ...) for this key.
5. We check if the calculated key exists in the `groups` dictionary. If not, we create an empty list for that key.

6. We append the current string s to the list corresponding to the calculated key in the groups dictionary.
7. After processing all strings, we return the values of the groups dictionary, which represent the grouped strings.
8. We test the class with the provided examples and print the outputs.

This solution efficiently groups strings based on the custom rule where two strings are in the same group if they have the same characters at even indices. The calculated key using character differences helps in identifying the groups, and the dictionary ensures that the strings are efficiently organized into their respective groups.

Question 50

Given a string s containing various characters, including parentheses ' $($ ', ' $)$ ', and other non-parentheses characters, determine if the input string is valid.

An input string is valid if:

Open brackets must be closed by the same type of brackets.

Open brackets must be closed in the correct order.

Every close bracket has a corresponding open bracket of the same type.

Write a function `isValidString(s: str) -> bool` to determine the validity of the input string.

Example 1:

Input: s = "(a[b{c}]d)"

Output: true

Explanation: The string is valid as parentheses are correctly nested and balanced.

Example 2:

Input: s = "a(b[c)d]"

Output: false

Explanation: The string is not valid as the parentheses are not correctly nested and balanced.

Example 3:

Input: s = "{a[b)c}"

Output: false

Explanation: The string is not valid as the parentheses are not correctly nested and balanced.

Constraints:

- $1 \leq s.length \leq 10^4$
- s consists of parentheses '(', ')', and other non-parentheses characters.

Solution:

```
1 ✓ class ValidStringChecker:
2 ✓     def isValidString(self, s: str) -> bool:
3         stack = []
4         mapping = {')': '(', '}': '{', ']': '['}
5
6 ✓     for char in s:
7 ✓         if char in mapping.values():
8             stack.append(char)
9 ✓         elif char in mapping.keys():
10            if not stack or stack.pop() != mapping[char]:
11                return False
12
13     return not stack
```

Solution Explanation:

1. We define a class `ValidStringChecker` with a method `isValidString` that takes a string `s` as input and returns `True` if the input string is valid according to the specified conditions, otherwise returns `False`.
2. We initialize an empty stack to keep track of open brackets.
3. We create a mapping dictionary that maps closing brackets to their corresponding opening brackets.
4. We iterate through each character `char` in the string `s`.
5. If the current character `char` is an opening bracket (value in mapping), we push it onto the stack.
6. If the current character `char` is a closing bracket (key in mapping), we check if the stack is empty or if the top element of the stack (last open bracket) doesn't match the corresponding opening bracket

for the current closing bracket. If either of these conditions is true, the string is not valid, and we return False.

7. After iterating through all characters, we check if the stack is empty. If it is, all brackets are balanced, and we return True, indicating a valid string. If the stack is not empty, there are unmatched open brackets, and we return False.
8. We test the class with the provided examples and print the outputs.

This solution uses a stack-based approach to check the validity of the input string according to the given conditions. It efficiently ensures that open brackets are properly closed and in the correct order by utilizing a stack to keep track of the open brackets encountered.

Question 51

Given a string s , determine if it can be transformed into a palindrome by changing at most one character.

ter.

A valid transformation involves changing any character in the string to any other character.

Write a function `isTransformableToPalindrome(s: str) -> bool` to determine whether the given string can be transformed into a palindrome with at most one character change.

Example 1:

Input: `s = "abca"`

Output: `true`

Explanation: The string "`abca`" can be transformed into a palindrome by changing the '`b`' to '`c`' or the '`c`' to '`b`'.

Example 2:

Input: `s = "hello"`

Output: `false`

Explanation: The string "hello" cannot be transformed into a palindrome with at most one character change.

Example 3:

Input: s = "radar"

Output: true

Explanation: The string "radar" is already a palindrome, so no changes are needed.

Constraints:

- $1 \leq s.length \leq 2 * 10^5$
- s consists only of printable ASCII characters.

Solution:

Solution Explanation:

1. We define a class `PalindromeTransformer` with a method `isTransformableToPalindrome` that takes a string `s` as input and returns `True` if the given string can be transformed into a palindrome by changing at most one character, otherwise returns `False`.
2. We define a helper function `is_palindrome` that takes a string as input and returns `True` if the string is a palindrome, otherwise returns `False`.
3. We check if the input string `s` is already a palindrome using the `is_palindrome` helper function. If it is, we return `True` immediately since no changes are needed.
4. If the string is not a palindrome, we iterate through the first half of the string. For each character at index `i`, we compare it with the corresponding character at the

opposite end of the string (index length - i - 1).

5. If the characters at indices i and length - i - 1 are not the same, it means a change is required. We increment the 'changes' counter.
6. If the number of changes exceeds 1, we return False immediately, as more than one change is not allowed to form a palindrome.
7. After iterating through the string, if the number of changes is at most 1, we return True, indicating that the string can be transformed into a palindrome with at most one character change.
8. We test the class with the provided examples and print the outputs.

This solution efficiently checks whether the given string can be transformed into a palindrome by comparing characters from the start and end of the string. The use of the `is_palindrome` helper func-

tion helps in quickly identifying if no changes are needed.

Question 52

Given a string s , find and return the shortest palindromic substring in s .

Example 1:

Input: $s = \text{"racecarabcde"}$

Output: "cec"

Explanation: The shortest palindromic substring is "cec".

Example 2:

Input: $s = \text{"hello"}$

Output: "ll"

Explanation: The shortest palindromic substring is "ll".

Constraints:

- $1 \leq s.length \leq 1000$
- s consists of only digits and English letters.

Solution:

```
1  class ShortestPalindromicSubstringFinder:
2      def shortestPalindromicSubstring(self, s: str) -> str:
3          def expand_around_center(left, right):
4              while left >= 0 and right < len(s) and s[left] == s[right]:
5                  left -= 1
6                  right += 1
7              return s[left + 1:right]
8
9          shortest_palindrome = ""
10
11         for i in range(len(s)):
12             odd_palindrome = expand_around_center(i, i)
13             even_palindrome = expand_around_center(i, i + 1)
14
15             if len(odd_palindrome) < len(shortest_palindrome):
16                 shortest_palindrome = odd_palindrome
17
18             if len(even_palindrome) < len(shortest_palindrome):
19                 shortest_palindrome = even_palindrome
20
21         return shortest_palindrome
```

Solution Explanation:

1. We define a class `ShortestPalindromicSubstringFinder` with a method `shortestPalindromicSubstring` that takes a string `s` as input and returns the shortest palindromic substring in the given string.
2. We define a helper function `expand_around_center` that takes two indices `left` and `right` as input and expands the substring around the center to find the longest palindromic substring with the center at the given indices.
3. We initialize an empty string `shortest_palindrome` to store the shortest palindromic substring found.
4. We iterate through each character in the string `s` using the index `i`.
5. For each character at index `i`, we call the `expand_around_center` helper function twice: once with the same index `i` as both left and right (for odd-length palin-

dromes) and once with i as left and $i + 1$ as right (for even-length palindromes).

6. We compare the lengths of the obtained odd and even palindromes with the length of the `shortest_palindrome`, and if either of them is shorter, we update `shortest_palindrome` accordingly.
7. After iterating through the string, we return the `shortest_palindrome`, which represents the shortest palindromic substring in the given string.
8. We test the class with the provided examples and print the outputs.

This solution efficiently finds the shortest palindromic substring by iteratively expanding around each character in the string and comparing the lengths of odd and even palindromes. The use of the `expand_around_center` helper function allows for a clean implementation of the palindrome expansion logic.

Question 53

Given a string s, return the length of the longest palindromic substring in it.

A string is a palindrome when it reads the same backward as forward.

Write a function `longestPalindromicSubstringLength(s: str) -> int` to determine the length of the longest palindromic substring in the given string.

Example 1:

Input: `s = "babad"`

Output: 3

Explanation: The longest palindromic substring is "`bab`" or "`aba`".

Example 2:

Input: `s = "cbbd"`

Output: 2

Explanation: The longest palindromic substring is "`bb`".

Constraints:

- $1 \leq s.length \leq 1000$
- s consists of lowercase English letters.

Solution:

```
1  class LongestPalindromicSubstring:
2      def longestPalindromicSubstringLength(self, s: str) -> int:
3          def expand_around_center(left, right):
4              while left >= 0 and right < len(s) and s[left] == s[right]:
5                  left -= 1
6                  right += 1
7              return right - left - 1
8
9          max_length = 0
10
11         for i in range(len(s)):
12             odd_length = expand_around_center(i, i)
13             even_length = expand_around_center(i, i + 1)
14             max_length = max(max_length, odd_length, even_length)
15
16         return max_length
```

Solution Explanation:

1. We define a class `LongestPalindromicSubstring` with a method `longestPalindromicSubstringLength` that takes a string `s` as input and returns the length of the longest palindromic substring in the given string.
2. We define a helper function `expand_around_center` that takes two indices `left` and `right` as input and expands the substring around the center to find the length of the palindromic substring with the center at the given indices.

3. We initialize `max_length` to store the length of the longest palindromic substring found.
4. We iterate through each character in the string `s` using the index `i`.
5. For each character at index `i`, we call the `expand_around_center` helper function twice: once with the same index `i` as both left and right (for odd-length palindromes) and once with `i` as left and `i + 1` as right (for even-length palindromes).
6. We calculate the length of the palindromic substring using the difference between right and left indices and subtracting 1 to account for the extra character added in the expansion.
7. We update the `max_length` with the maximum value among `max_length`, the length of odd-length palindrome, and the length of even-length palindrome.
8. After iterating through the string, we return `max_length`, which represents the

length of the longest palindromic substring in the given string.

9. We test the class with the provided examples and print the outputs.

This solution efficiently finds the length of the longest palindromic substring by iteratively expanding around each character in the string and calculating the length of odd and even palindromes. The use of the `expand_around_center` helper function allows for a clean implementation of the palindrome expansion logic.

SECTION 9 - TREE

Question 54

Given the root of a binary tree, return the minimum depth of the tree.

The minimum depth is defined as the number of nodes along the shortest path from the root node down to the nearest leaf node.

Write a function `minDepth(root: TreeNode) -> int` to determine the minimum depth of the binary tree.

Example 1:

Input: `root = [3,9,20,null,null,15,7]`

Output: 2

Explanation: The minimum depth is 2, as the shortest path is from the root node with value 3 to the leaf node with value 15.

Example 2:

Input: root = [1,null,2]

Output: 2

Explanation: The minimum depth is 2, as the shortest path is from the root node with value 1 to the leaf node with value 2.

Constraints:

- The number of nodes in the tree is in the range $[0, 10^4]$.

- $-100 \leq \text{Node.val} \leq 100$

Solution:

```
1 ✓ class TreeNode:
2   ✓     def __init__(self, val=0, left=None, right=None):
3       self.val = val
4       self.left = left
5       self.right = right
6
7 ✓ class MinimumDepthBinaryTree:
8   ✓     def minDepth(self, root: TreeNode) -> int:
9       ✓         if not root:
10           return 0
11
12       ✓         if not root.left and not root.right:
13             return 1
14
15       min_depth = float('inf')
16
17       ✓         if root.left:
18             min_depth = min(min_depth, self.minDepth(root.left))
19
20       ✓         if root.right:
21             min_depth = min(min_depth, self.minDepth(root.right))
22
23       return min_depth + 1
```

Solution Explanation:

1. We define a class `TreeNode` to represent the nodes of the binary tree, with attributes `val`, `left`, and `right`.
2. We define a class `MinimumDepthBinaryTree` with a method `minDepth` that takes the root of a binary tree as input and returns the minimum depth of the tree.
3. In the `minDepth` method, we handle base cases: if the root is `None`, return 0 (indicating an empty tree), and if the root has no left and right children, return 1 (indicating a leaf node).
4. We initialize `min_depth` with a large value (`infinity`) to keep track of the minimum depth of the binary tree.
5. We recursively calculate the minimum depth of the left subtree by calling `self.minDepth(root.left)` and update `min_depth` with the minimum value be-

tween `min_depth` and the calculated depth.

6. We recursively calculate the minimum depth of the right subtree by calling `self.minDepth(root.right)` and update `min_depth` again.
7. Finally, we return `min_depth + 1` to account for the current node in the calculation.
8. We test the class with the provided examples and print the outputs.

This solution efficiently calculates the minimum depth of a binary tree by recursively traversing the tree and updating the minimum depth at each step. The use of the `TreeNode` class allows for easy representation of the binary tree nodes.

Question 55

Given the roots of two binary trees p and q, write a function to check if they are mirror images of each other.

Two binary trees are considered mirror images if one tree is a reflection of the other around the root.

Write a function `areMirrors(p: TreeNode, q: TreeNode) -> bool` to determine whether the two given binary trees p and q are mirror images of each other.

Example 1:

Input:

p =

1

/ \

2 2

/ \ / \

3 4 4 3

q =

1

/ \

2 2

/ \ / \

3 4 4 3

Output: true

Explanation: The trees are mirror images of each other.

Example 2:

Input:

p =

```
1
/\ 
2 2
\ \
3 3
```

q =

```
1
/\ 
2 2
\ \
3 3
```

Output: false

Explanation: The trees are not mirror images of each other.

Constraints:

- The number of nodes in both trees is in the range [0, 100].
- $-10^4 \leq \text{Node.val} \leq 10^4$

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7  class MirrorBinaryTrees:
8      def areMirrors(self, p: TreeNode, q: TreeNode) -> bool:
9          if not p and not q:
10              return True
11          if not p or not q:
12              return False
13          return (
14              p.val == q.val) and self.areMirrors(
15                  p.left, q.right
16              ) and self.areMirrors(
17                  p.right, q.left
18              )
```

Solution Explanation:

1. We define a class `TreeNode` to represent the nodes of the binary tree, with attributes `val`, `left`, and `right`.
2. We define a class `MirrorBinaryTrees` with a method `areMirrors` that takes two root nodes of binary trees `p` and `q` as input and

returns a boolean indicating whether the trees are mirror images of each other.

3. In the `areMirrors` method, we handle the base cases: if both `p` and `q` are `None`, return `True` (indicating they are mirror images), and if only one of `p` or `q` is `None`, return `False` (indicating they are not mirror images).
4. We recursively check whether the values of the current nodes `p` and `q` are equal and whether their subtrees are mirror images of each other. This is done by comparing the left subtree of `p` with the right subtree of `q`, and the right subtree of `p` with the left subtree of `q`.
5. We test the class with the provided examples and print the outputs.

This solution efficiently checks whether two binary trees are mirror images of each other by recursively comparing their structures and values. The use of

the `TreeNode` class allows for easy representation of the binary tree nodes.

Question 56

Given the root of a binary tree, perform a "twist" operation on the tree, where for each node, you swap its left and right subtrees. Return the root of the modified tree.

Example 1:

Input: `root = [4,2,7,1,3,6,9]`

Output: `[4,7,2,9,6,3,1]`

Explanation: The tree is twisted by swapping the left and right subtrees for each node.

Example 2:

Input: root = [2,1,3]

Output: [2,3,1]

Explanation: The tree is twisted by swapping the left and right subtrees for each node.

Example 3:

Input: root = []

Output: []

Explanation: The tree is empty, so no twist is performed.

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- $-100 \leq \text{Node.val} \leq 100$

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7  class BinaryTreeTwist:
8      def twist(self, root: TreeNode) -> TreeNode:
9          if not root:
10              return None
11
12          # Perform the twist operation
13          root.left, root.right = root.right, root.left
14
15          # Recursively twist the left and right subtrees
16          self.twist(root.left)
17          self.twist(root.right)
18
19          return root
```

Solution Explanation:

1. We define a class `TreeNode` to represent the nodes of the binary tree, with attributes `val`, `left`, and `right`.
2. We define a class `BinaryTreeTwist` with a method `twist` that takes the root node of a binary tree as input and returns the root of the modified tree after performing the twist operation.
3. In the `twist` method, we handle the base case where the current node is `None`, indicating an empty tree, and simply return `None`.
4. For each node in the tree, we perform the twist operation by swapping the left

and right subtrees using the line: `root.left, root.right = root.right, root.left`.

5. We then recursively call the `twist` method on the left and right subtrees to perform the twist operation on them.
6. Finally, we return the root node of the modified tree.
7. We test the class with the provided examples and print the structure of the twisted trees for verification (printing the tree structure is not necessary in the actual solution).

This solution efficiently performs the twist operation on a binary tree using recursion and modifies the tree in place. The `TreeNode` class allows us to easily represent and manipulate the tree nodes.

Question 57

Given the root of a binary tree, find the maximum sum of any path between two nodes in the tree. The path may or may not pass through the root.

Write a function `maxSumPathBetweenNodes(root: TreeNode) -> int` to determine the maximum sum of any path between two nodes in the binary tree.

Example 1:

Input:

```
1
 / \
2 3
```

Output: 6

Explanation: The maximum sum path is 2 -> 1 -> 3 with a sum of 6.

Example 2:

Input:

-10

/ \

9 20

/ \

15 7

Output: 42

Explanation: The maximum sum path is 15 -> 20 -> 7 with a sum of 42.

Constraints:

- The number of nodes in the tree is in the range $[1, 3 * 10^4]$.
- $-1000 \leq \text{Node.val} \leq 1000$

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7  class MaxPathSum:
8      def maxSumPathBetweenNodes(self, root: TreeNode) -> int:
9          self.max_sum = float('-inf') # Initialize max_sum to negative infinity
10
11         # Helper function to recursively calculate the maximum path sum
12         def max_path_sum(node):
13             if not node:
14                 return 0
15
16             # Calculate the maximum path sum for the left and right subtrees
17             left_sum = max(max_path_sum(node.left), 0)
18             right_sum = max(max_path_sum(node.right), 0)
19
20             # Update the maximum path sum by considering the current node
21             self.max_sum = max(self.max_sum, left_sum + right_sum + node.val)
22
23             # Return the maximum path sum that can be extended from the current node
24             return max(left_sum, right_sum) + node.val
25
26         # Call the helper function to calculate the maximum path sum
27         max_path_sum(root)
28
29         return self.max_sum
```

Solution Explanation:

1. We define a class `TreeNode` to represent the nodes of the binary tree, with attributes `val`, `left`, and `right`.
2. We define a class `MaxPathSum` with a method `maxSumPathBetweenNodes` that takes the root node of a binary tree as input and returns the maximum sum of any path between two nodes in the tree.
3. Inside the `maxSumPathBetweenNodes` method, we initialize the `max_sum` variable to negative infinity. This variable will be used to store the maximum path sum.
4. We define a helper function `max_path_sum` that takes a node as input and recur-

sively calculates the maximum path sum that can be extended from that node.

5. Inside the `max_path_sum` function, we handle the base case where the current node is `None` and return 0.
6. For each non-null node, we calculate the maximum path sum for the left and right subtrees using the `max_path_sum` function recursively. We use `max(..., 0)` to consider only positive path sums.
7. We then update the `max_sum` by considering the current node's value and the maximum path sums of the left and right subtrees.
8. Finally, we return the maximum path sum that can be extended from the current node.
9. We call the `max_path_sum` helper function with the root node to calculate the maximum path sum for the entire tree.
10. We return the calculated `max_sum` as the result.

11. We test the MaxPathSum class with the provided examples and print the calculated maximum path sums for verification.

This solution efficiently calculates the maximum sum of any path between two nodes in the binary tree by recursively exploring all possible paths and keeping track of the maximum sum encountered. The TreeNode class allows us to easily represent and manipulate the tree nodes.

Question 58

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. Zigzag level order

traversal is a variation of level order traversal where, in addition to moving from left to right, nodes at even levels are traversed from right to left.

Write a function `zigzagLevelOrderTraversal(root: TreeNode) -> List[List[int]]` to determine the zigzag level order traversal of the binary tree.

Example:

Input:

```
3
 / \
9 20
 / \
15 7
```

Output: [[3], [20, 9], [15, 7]]

Explanation: The zigzag level order traversal is done as follows:

- Level 0: [3]
- Level 1: [20, 9]

- Level 2: [15, 7]

Example:

Input:

1

Output: [[1]]

Explanation: The zigzag level order traversal has only one level.

Constraints:

- The number of nodes in the tree is in the range [0, 2000].
- $-1000 \leq \text{Node.val} \leq 1000$

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7  from collections import deque
8  from typing import List
9  class ZigzagLevelOrderTraversal:
10     def zigzagLevelOrder(self, root: TreeNode) -> List[List[int]]:
11         if not root:
12             return []
13
14         result = []
15         queue = deque([root])
16         reverse = False
17
18         while queue:
19             level_values = []
20             level_size = len(queue)
21
22             for _ in range(level_size):
23                 node = queue.popleft()
24                 if reverse:
25                     level_values.insert(0, node.val)
26                 else:
27                     level_values.append(node.val)
28
29                 if node.left:
30                     queue.append(node.left)
31                 if node.right:
32                     queue.append(node.right)
33
34             result.append(level_values)
35             reverse = not reverse
36
37         return result
```

Solution Explanation:

1. We define a class `TreeNode` to represent the nodes of the binary tree, with attributes `val`, `left`, and `right`.
2. We import the `deque` class from the `collections` module to use it as our queue for level order traversal.
3. We define a class `ZigzagLevelOrderTraversal` with a method `zigzagLevelOrder` that takes the root node of a binary tree as input and returns the zigzag level order traversal of the tree.
4. Inside the `zigzagLevelOrder` method, we handle the base case where the root is `None` by returning an empty list.

5. We initialize an empty list result to store the final zigzag level order traversal.
6. We initialize a deque called queue with the root node.
7. We use a boolean variable reverse to keep track of whether the current level should be reversed (odd levels).
8. We start a while loop to traverse the tree level by level.

Question 59

Description:

Serialization and deserialization of binary trees are common tasks in computer science. In this chal-

lenge, you're tasked with designing a custom algorithm for serializing and deserializing binary trees. Your algorithm should be able to serialize a binary tree into a compact string representation and then reconstruct the original tree from that string.

Task:

Design an algorithm to perform custom serialization and deserialization of a binary tree. Your solution should meet the following requirements:

The serialization algorithm should convert a binary tree into a string representation.

The deserialization algorithm should reconstruct the original binary tree from the string representation.

You are free to choose the format of the string representation and the logic for serialization/deserialization. Focus on creating an efficient and clear algorithm.

Example:

```
# Example binary tree
```

```
1
 / \
2 3
 / \
4 5
```

```
# Serialized string format: "1 2 XX 3 4 XX 5 XX"
```

```
# Deserialization: Reconstruct the binary tree from
the serialized string
```

```
root = deserialize("1 2 XX 3 4 XX 5 XX")
```

```
# root should now contain the original binary tree
```

Note:

1. You are not required to use the LeetCode serialization format, but feel free to come up with your own creative approach.

2. You can assume that the tree has at most 10^4 nodes and node values are within the range of -1000 to 1000.

Constraints:

- The number of nodes in the tree is in the range $[0, 10^4]$.
- $-1000 \leq \text{Node.val} \leq 1000$

Solution:

```
1 class TreeNode:
2     def __init__(self, val=0, left=None, right=None):
3         self.val = val
4         self.left = left
5         self.right = right
6
7     def serialize(root):
8         if not root:
9             return "X" # Represent None with 'X'
10
11         left = serialize(root.left)
12         right = serialize(root.right)
13
14         return f"{root.val} {left} {right}"
15
16     def deserialize(data):
17         def build_tree(nodes):
18             val = nodes.pop(0)
19             if val == 'X':
20                 return None
21             node = TreeNode(int(val))
22             node.left = build_tree(nodes)
23             node.right = build_tree(nodes)
24             return node
25
26         nodes = data.split()
27         return build_tree(nodes)
```

Solution Explanation:

The provided Python code defines a `TreeNode` class to represent nodes of the binary tree. The `serialize` function takes the root of a binary tree and returns a string representation of the tree in a custom format. The serialization process is done using a pre-order traversal, where each node's value is followed by the serialized left subtree and then the serialized right subtree.

The `deserialize` function takes the serialized string and reconstructs the original binary tree. It does this by splitting the serialized string into a list of nodes, then recursively building the tree based on the nodes.

Question 60

Given the roots of two binary trees `root` and `subRoot`, write a function to check if the binary tree rooted at

`subRoot` is a subtree of the binary tree rooted at `root`.

A binary tree `tree` is considered a subtree of another binary tree `subtree` if there exists a node `node` in the `subtree` such that the subtree of `node` is identical to `tree`.

Write a function `isSubtree(root: TreeNode, subRoot: TreeNode) -> bool` to determine whether the binary tree rooted at `subRoot` is a subtree of the binary tree rooted at `root`.

Examples:

Example 1:

Input: `root = [3,4,5,1,2]`, `subRoot = [4,1,2]`

Output: `true`

Explanation: The binary tree rooted at node 4 in the root tree is identical to the `subRoot` tree.

Example 2:

Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]

Output: false

Explanation: There is no subtree rooted at any node in the root tree that is identical to the subRoot tree.

Constraints:

- The number of nodes in the root tree is in the range [1, 2000].
- The number of nodes in the subRoot tree is in the range [1, 1000].
- Node values are within the range of -10^4 to 10^4 .

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7  def isSubtree(root, subRoot):
8      if not root:
9          return False
10
11     if isIdentical(root, subRoot):
12         return True
13
14     return isSubtree(root.left, subRoot) or isSubtree(root.right, subRoot)
15
16 def isIdentical(root1, root2):
17     if not root1 and not root2:
18         return True
19     if not root1 or not root2:
20         return False
21     return (
22         root1.val == root2.val
23         ) and isIdentical(
24             root1.left, root2.left
25             ) and isIdentical(root1.right, root2.right)
```

Solution Explanation:

In this solution, the `isSubtree` function takes two parameters, `root` and `subRoot`, representing the roots of the main binary tree and the subtree, respectively. It first checks if the current node in the root tree is identical to the `subRoot` tree using the `isIdentical` helper function. If it is, then we return `True`. Otherwise, we recursively check if the subtree rooted at the left or right child of the current node is identical to the `subRoot` tree.

The `isIdentical` function checks whether two trees are identical by comparing their values and recursively checking their left and right subtrees.

This solution works by traversing the root tree and checking for each node if its subtree is identical to the `subRoot` tree. If we find any such node, we return `True`, indicating that the `subRoot` tree is a subtree of the root tree. If we traverse the entire root tree without finding a matching subtree, we return `False`.

Question 61

You are given the postorder and inorder traversals of a binary tree. Your task is to construct and return the binary tree.

Write a function `buildTreeFromPostorderAndInorder(postorder: List[int], inorder: List[int]) -> TreeNode` to construct the binary tree using the given postorder and inorder traversals.

Example:

Input: postorder = [9,15,7,20,3], inorder = [9,3,15,20,7]

Output: [3,9,20,null,null,15,7]

Input: postorder = [-1], inorder = [-1]

Output: [-1]

Constraints:

- $1 \leq \text{postorder.length} \leq 3000$
- $\text{inorder.length} == \text{postorder.length}$
- $-3000 \leq \text{postorder}[i], \text{inorder}[i] \leq 3000$
- postorder and inorder consist of unique values.
- Each value of inorder also appears in postorder.
- postorder is guaranteed to be the postorder traversal of the tree.
- inorder is guaranteed to be the inorder traversal of the tree.

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7  def buildTreeFromPostorderAndInorder(postorder, inorder):
8      if not postorder or not inorder:
9          return None
10
11     root_val = postorder.pop()
12     root = TreeNode(root_val)
13     root_idx = inorder.index(root_val)
14
15     root.right = buildTreeFromPostorderAndInorder(postorder, inorder[root_idx+1:])
16     root.left = buildTreeFromPostorderAndInorder(postorder, inorder[:root_idx])
17
18     return root
19
20 # Helper function to print the tree (for verification)
21 def printTree(root):
22     if root:
23         printTree(root.left)
24         print(root.val, end=' ')
25         printTree(root.right)
```

Solution Explanation:

1. We define a `TreeNode` class to represent each node in the binary tree.
2. The `buildTreeFromPostorderAndInorder` function takes postorder and inorder traversals as input and recursively constructs the binary tree.
3. The base case of the recursion is when either postorder or inorder is empty, in which case we return `None`.
4. We pop the last element from postorder to get the root value of the current subtree.
5. We find the index of the root value in the inorder traversal to determine the left and right subtrees.
6. We recursively build the right subtree first, then the left subtree.
7. Finally, we return the root node of the current subtree.

8. The `printTree` function is a helper function to print the tree nodes in inorder traversal.

This solution efficiently constructs the binary tree using the given postorder and inorder traversals.

Question 62

You are given a list of unique integers. Write a function `isMonotonicSequence(arr: List[int]) -> bool` to determine if the given list forms a monotonic sequence.

A monotonic sequence is defined as follows:

It is either entirely non-increasing (each element is less than or equal to the previous element), or

It is entirely non-decreasing (each element is greater than or equal to the previous element).

Write a function to determine whether the given list of integers is a monotonic sequence or not.

Example 1:

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

Output: true

Explanation: The list is non-decreasing.

Example 2:

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

Output: true

Explanation: The list is non-increasing.

Example 3:

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

Output: false

Explanation: The list is neither entirely non-decreasing nor non-increasing.

Constraints:

- The length of the array is in the range $[1, 10^4]$.
- $-2^{31} \leq \text{arr}[i] \leq 2^{31} - 1$

Solution:

```
1  from typing import List
2
3  class Solution:
4      def isMonotonicSequence(self, arr: List[int]) -> bool:
5          # Check if the list is entirely non-decreasing
6          def isNonDecreasing(arr):
7              for i in range(1, len(arr)):
8                  if arr[i] < arr[i - 1]:
9                      return False
10             return True
11
12          # Check if the list is entirely non-increasing
13          def isNonIncreasing(arr):
14              for i in range(1, len(arr)):
15                  if arr[i] > arr[i - 1]:
16                      return False
17              return True
18
19          # Check if the list is either non-decreasing or non-increasing
20          return isNonDecreasing(arr) or isNonIncreasing(arr)
```

Solution Explanation:

1. We define two helper functions `isNonDecreasing` and `isNonIncreasing` to check if the given list is entirely non-decreasing or non-increasing, respectively. These functions iterate through the list and compare each element with its previous element. If the condition is violated, we return `False`, indicating that the list is not monotonic.
2. The `isMonotonicSequence` function first checks if the list is non-decreasing using the `isNonDecreasing` function. If it is, we return `True`. Otherwise, we check if the list is non-increasing using the `isNonIncreasing` function. If it is, we also return `True`. If neither condition is satisfied, we return `False`.
3. We create an instance of the `Solution` class and use it to call the `isMonotonicSequence` function with different input lists to test the solution.

This solution has a time complexity of $O(n)$, where n is the length of the input list arr , as we need to iterate through the list to check if it is non-decreasing or non-increasing.

Question 63

You are given the root of a binary search tree and an integer k . Your task is to design a function to return the k^{th} largest value among all the values of the nodes in the tree.

Example:

Input:

$\text{root} = [8, 4, 10, 2, 6, 9, 11]$, $k = 2$

Output: 10

Explanation: The k^{th} largest value for $k = 2$ is 10.

Constraints:

- The number of nodes in the tree is n .
- $1 \leq k \leq n \leq 10^4$
- $0 \leq \text{Node.val} \leq 10^4$

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7      def kthLargest(self, k):
8          def reverse_inorder(node):
9              if not node:
10                  return []
11              return reverse_inorder(node.right) + [node.val] + reverse_inorder(node.left)
12
13          values = reverse_inorder(self)
14          return values[k - 1]
```

Solution Explanation:

1. We define a `TreeNode` class to represent nodes in the binary search tree. Each node has a value (`val`), a left child (`left`), and a right child (`right`).
2. The function `kthLargest(root, k)` takes the root of the binary search tree and an integer `k` as parameters and returns the k^{th} largest value.
3. Inside the function, we define a nested helper function `reverse_inorder(node)` which performs a reverse inorder traversal of the tree. In a reverse inorder traversal, we visit the right child first, then the root, and finally the left child.

sal, we visit the right subtree first, then the current node, and finally the left subtree. This results in values in descending order.

4. The base case for the reverse inorder traversal is when node is None, in which case we return an empty list.
5. For each non-empty node, we recursively perform a reverse inorder traversal on its right subtree, then append the current node's value to the list, and finally perform a reverse inorder traversal on its left subtree. This ensures that we collect values in descending order.
6. The values list will contain the values of the binary search tree in descending order.
7. Finally, we return values[k - 1] to get the k^{th} largest value. Since the list is 0-indexed, we subtract 1 from k.

This solution performs a reverse inorder traversal of the binary search tree to collect values in descending order, and then returns the k^{th} largest value from the list of values. The time complexity of this solution is $O(n)$, where n is the number of nodes in the binary search tree, since we visit each node once.

Question 64

Given a binary tree (not necessarily a binary search tree), find the lowest common ancestor (LCA) node of two given nodes in the tree.

The lowest common ancestor is defined between two nodes p and q as the lowest node in the tree that has both p and q as descendants (where we allow a node to be a descendant of itself).

Example 1:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

Example 3:

Input: root = [2,1,3], p = 2, q = 3

Output: 2

Constraints:

- The number of nodes in the tree is in the range $[2, 10^5]$.
- $-10^9 \leq \text{Node.val} \leq 10^9$
- All Node.val are unique.
- p and q will exist in the tree.

Solution:

```
1  class TreeNode:
2      def __init__(self, val=0, left=None, right=None):
3          self.val = val
4          self.left = left
5          self.right = right
6
7  class Solution:
8      def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) -> TreeNode:
9          if root is None:
10              return None
11
12          # If either p or q is the root, then root is the LCA
13          if root == p or root == q:
14              return root
15
16          # Recur for left and right subtrees
17          left_lca = self.lowestCommonAncestor(root.left, p, q)
18          right_lca = self.lowestCommonAncestor(root.right, p, q)
19
20          # If both left and right subtrees have an LCA, then root is the LCA
21          if left_lca and right_lca:
22              return root
23
24          # Otherwise, return the non-empty subtree
25          return left_lca if left_lca else right_lca
```

Solution Explanation:

1. We define a `TreeNode` class to represent nodes of the binary tree.
2. The `lowestCommonAncestor` function takes the root of the binary tree, and two nodes `p` and `q` as inputs.
3. We handle base cases: if the root is `None`, we return `None`. If either `p` or `q` is the root, then the root is the LCA.
4. We recursively search for the LCA in the left and right subtrees.
5. If both left and right subtrees have an LCA, then the root is the LCA.

6. If only one subtree has an LCA, we return that LCA.
7. This solution ensures that we traverse the binary tree once, making it efficient.

Now, you can create a `Solution` object and call the `lowestCommonAncestor` method to find the LCA of two nodes in a given binary tree.

Question 65

You are given a list of words. Your task is to implement a class that supports the following operations:

1. 'WordFilter(words: List[str])' Initializes the word filter object with the given list of words.
2. 'f(prefix: str, suffix: str) -> int' Returns the index of a word in the list that has the prefix 'prefix' and the suffix 'suffix'. If there is no such word in the list, return -1.

Example 1:

Input:

```
wordFilter = WordFilter(["apple"])
wordFilter.f("a", "e") # Output: 0
wordFilter.f("b", "") # Output: -1
```

Explanation: The `f("a", "e")` returns 0 because the word "apple" has both the prefix "a" and the suffix "e".

The `f("b", "")` returns -1 because there is no word in the list that starts with "b" and has an empty suffix.

Constraints:

- $1 \leq \text{words.length} \leq 15000$
- $1 \leq \text{words}[i].length \leq 10$
- $1 \leq \text{prefix.length}, \text{suffix.length} \leq 10$
- `words[i]`, `prefix`, and `suffix` consist only of lowercase English letters.
- At most 10^4 calls will be made to `f`.

Solution:

```
1  class WordFilter:
2
3      def __init__(self, words):
4          self.word_map = {}
5          for weight, word in enumerate(words):
6              for i in range(len(word) + 1):
7                  for j in range(len(word) + 1):
8                      self.word_map[word[:i] + '#' + word[j:]] = weight
9
10     def f(self, prefix, suffix):
11         query = prefix + '#' + suffix
12         if query in self.word_map:
13             return self.word_map[query]
14         return -1
```

Solution Explanation:

1. In the constructor `__init__`, we create a `word_map` dictionary to store the mappings of prefixes and suffixes of each word to their corresponding indices in the given list of words. For each word, we generate all possible prefixes and suffixes by adding a '#' in between different positions of the word, and store the mapping of these substrings to the index of the word in the list.
2. In the `f` method, we concatenate the prefix and suffix with '#' in between to create the

query string. We then check if this query string exists in the word_map. If it does, we return the corresponding index of the word; otherwise, we return -1.

This approach efficiently preprocesses the words in the constructor to allow quick lookups in the f method, resulting in a time complexity of O(1) for each query.

Question 66

Design a data structure called PatternDictionary that supports adding new patterns and finding if a string matches any previously added pattern.

Implement the PatternDictionary class:

PatternDictionary() Initializes the object.

void addPattern(pattern) Adds the pattern to the data structure, it can be matched later. A pattern is a sequence of characters consisting of lowercase English letters.

bool isMatch(string) Returns true if there is any pattern in the data structure that matches the input string or false otherwise. The input string may contain wildcards '?' that can match any single character.

Example:

Input

```
["PatternDictionary", "addPattern", "addPattern",
 "addPattern", "isMatch", "isMatch", "isMatch", "is-
Match"]
[[], ["bad"], ["dad"], ["mad"], ["pad"], ["bad"], [".ad"],
 ["b.."]]
```

Output

```
[null, null, null, null, False, True, True, True]
```

Explanation

```
PatternDictionary patternDictionary = new Pattern-
Dictionary();
patternDictionary.addPattern("bad");
patternDictionary.addPattern("dad");
patternDictionary.addPattern("mad");
patternDictionary.isMatch("pad"); // return False
patternDictionary.isMatch("bad"); // return True
patternDictionary.isMatch(".ad"); // return True
patternDictionary.isMatch("b.."); // return True
```

Constraints:

- $1 \leq \text{pattern.length} \leq 25$
- pattern in addPattern consists of lowercase English letters.
- pattern in isMatch consists of '.' or lowercase English letters.
- There will be at most 2 dots in pattern for isMatch queries.
- At most 10^4 calls will be made to addPattern and isMatch.

Solution:

Solution Explanation:

1. We define a `TrieNode` class to represent nodes in the trie.
2. The `PatternDictionary` class uses a trie to store patterns. The `addPattern` method inserts patterns into the trie.
3. The `_search` method recursively searches for a match in the trie while considering wildcard characters.
4. The `isMatch` method calls the `_search` method to check if a given string matches any pattern in the trie.

This solution efficiently handles adding patterns and searching for matches, taking advantage of the trie structure to optimize the search process. The time complexity for adding a pattern is $O(k)$, where k is the length of the pattern, and the time complexity for searching is $O(n)$, where n is the length of the input string.

Question 67

You are given a 2D grid of characters and a list of strings. Your task is to design a function that returns

all the words from the list that can be found in the grid.

Implement the function `findWordsInGrid(board: List[List[str]], words: List[str]) -> List[str]` that takes a 2D grid board of characters and a list of strings words. The function should return a list of words from the input list that can be formed by following a path of adjacent cells (horizontally or vertically) in the grid.

A cell in the grid can only be used once in a word. Words can start from any cell in the grid, and they cannot change directions mid-path.

Example:

Input:

board = [

 ["o","a","a","n"],

```
    ["e","t","a","e"],  
    ["i","h","k","r"],  
    ["i","f","l","v"]  
]  
  
words = ["oath","pea","eat","rain"]
```

Output:

```
["eat","oath"]
```

Explanation: The words "eat" and "oath" can be formed by following a path of adjacent cells in the grid.

Input:

```
board = [  
    ["a","b"],  
    ["c","d"]  
]
```

```
words = ["abcb"]
```

Output:

```
[]
```

Explanation: The word "abcb" cannot be formed by following a path of adjacent cells in the grid.

Constraints:

- The dimensions of the board grid are $m \times n$ where $1 \leq m, n \leq 12$.
- Each cell $\text{board}[i][j]$ contains a lowercase English letter.
- The list words has $1 \leq \text{words.length} \leq 3 * 10^4$.
- Each word $\text{words}[i]$ has $1 \leq \text{words}[i].length \leq 10$.
- The words in the list 'words' are unique.

Solution:

```
1  class TrieNode:
2      def __init__(self):
3          self.children = {}
4          self.word = None
5
6  class Trie:
7      def __init__(self):
8          self.root = TrieNode()
9
10     def insert(self, word):
11         node = self.root
12         for char in word:
13             if char not in node.children:
14                 node.children[char] = TrieNode()
15                 node = node.children[char]
16         node.word = word
```

```
18 class Solution:
19     def findWordsInGrid(self, board, words):
20         def backtrack(node, row, col):
21             char = board[row][col]
22             curr_node = node.children.get(char)
23             if not curr_node:
24                 return
25
26             if curr_node.word:
27                 result.append(curr_node.word)
28                 curr_node.word = None # Avoid duplicate results
29
30             board[row][col] = '#' # Mark cell as visited
31             for dr, dc in directions:
32                 newRow, newCol = row + dr, col + dc
33                 if 0 <= newRow < rows and 0 <= newCol < cols:
34                     backtrack(curr_node, newRow, newCol)
35             board[row][col] = char # Reset cell
36
37         trie = Trie()
38         for word in words:
39             trie.insert(word)
40
41         rows, cols = len(board), len(board[0])
42         result = []
43         directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
44
45         for row in range(rows):
46             for col in range(cols):
47                 backtrack(trie.root, row, col)
48
49         return result
```

Solution Explanation:

1. We define a TrieNode class to represent nodes in the Trie and a Trie class to store the words from the list.
2. We implement the findWordsInGrid function using DFS. For each cell in the grid, we start a DFS traversal to find words that can be formed by following adjacent cells.
3. During DFS traversal, we update the Trie node according to the characters in the current cell.
4. If we find a word in the Trie, we append it to the result list and mark the word as None to avoid duplicates.
5. We reset the cells to their original characters after backtracking.

This solution efficiently finds words in the grid by using a Trie data structure and performing a depth-first search on the grid. It avoids unnecessary traversals and ensures that each cell is visited only once.

SECTION 10

- HEAP

Question 68

You are given a list of tasks performed by different workers, along with the time taken by each worker to complete each task. Your task is to design a function that returns the k most efficient workers based on their overall performance.

Implement the function `getMostEfficientWorkers(tasks: List[List[int]], k: int) -> List[int]`, where `tasks` is a list of lists where each sublist contains the time

taken by a worker to complete each task. The function should return a list of k worker indices that have the highest overall performance.

The overall performance of a worker is calculated as the sum of the time taken to complete all their tasks.

Example:

Input:

```
tasks = [  
    [5, 3, 2],  
    [8, 2, 6],  
    [7, 4, 9]  
]  
k = 2
```

Output:

[1, 2]

Explanation: Worker 1 has an overall performance of $16 (8 + 2 + 6)$, and worker 2 has an overall performance of $20 (7 + 4 + 9)$. Therefore, the $k = 2$ most efficient workers are workers 1 and 2.

Constraints:

- The number of workers is in the range [1, 1000].
- The number of tasks performed by each worker is in the range [1, 100].
- The time taken to complete each task is in the range [1, 100].
- k is in the range [1, the number of workers].

Solution:

```
1  from typing import List
2
3  class EfficientWorkerFinder:
4      def getMostEfficientWorkers(self, tasks: List[List[int]], k: int) -> List[int]:
5          performance = []
6          for i, task_times in enumerate(tasks):
7              total_time = sum(task_times)
8              performance.append((total_time, i))
9
10         performance.sort(reverse=True)
11         most_efficient_workers = [worker_idx for _, worker_idx in performance[:k]]
12         return most_efficient_workers
```

Solution Explanation:

1. We create a class `EfficientWorkerFinder` with a method `getMostEfficientWorkers` that takes tasks and k as input.
2. We iterate through the tasks list, calculate the total time for each worker's tasks, and store the total time and worker index as a tuple in the performance list.
3. We sort the performance list in descending order based on the total time.
4. We extract the worker indices from the sorted performance list for the first k workers and return the result.

This solution has a time complexity of $O(n * m * \log n)$, where n is the number of workers and m is the number of tasks performed by each worker. The most time-consuming operation is the sorting step.

Question 69

You are designing a data structure to monitor a stream of temperature readings. Your task is to im-

plement a class PeakFinder that supports the following operations:

1. PeakFinder(): Initializes the PeakFinder object.
2. addTemperature(temperature): Adds a new temperature reading temperature to the data structure.
3. findPeak() -> int: Returns the highest peak temperature observed so far. A peak temperature is defined as a temperature that is higher than its neighboring temperatures. If there are multiple peaks, return the highest one. If no peak is found, return -1.
4. findValley() -> int: Returns the lowest valley temperature observed so far. A valley temperature is defined as a temperature that is lower than its neighboring temperatures. If there are multiple valleys, return the lowest one. If no valley is found, return -1.

5. `findAverage()` -> float: Returns the average temperature of all the recorded readings.

Example:

```
# Example usage
peakFinder = PeakFinder()
peakFinder.addTemperature(72)
peakFinder.addTemperature(68)
peakFinder.addTemperature(76)
peakFinder.addTemperature(74)

print(peakFinder.findPeak())    # Output: 76
print(peakFinder.findValley())  # Output: 68
print(peakFinder.findAverage()) # Output: 72.5
```

Constraints:

- The temperature readings are in the range [-100, 100].
- At most 10^4 calls will be made to addTemperature, findPeak, findValley, and findAverage combined.

Solution:

```
1  class PeakFinder:
2      def __init__(self):
3          self.temperatures = []
4
5      def addTemperature(self, temperature):
6          self.temperatures.append(temperature)
7
8      def findPeak(self):
9          peak = -1
10         for i in range(1, len(self.temperatures) - 1):
11             if self.temperatures[
12                 i
13                 ] > self.temperatures[
14                     i - 1
15                     ] and self.temperatures[i] > self.temperatures[i + 1]:
16                 peak = max(peak, self.temperatures[i])
17
18         return peak
```

```
19     def findValley(self):
20         valley = float('inf')
21         for i in range(1, len(self.temperatures) - 1):
22             if self.temperatures[
23                 i
24                 ] < self.temperatures[
25                     i - 1
26                     ] and self.temperatures[i] < self.temperatures[i + 1]:
27                 valley = min(valley, self.temperatures[i])
28         return valley if valley != float('inf') else -1
29
30     def findAverage(self):
31         if not self.temperatures:
32             return -1
33         return sum(self.temperatures) / len(self.temperatures)
```

Solution Explanation:

1. In the PeakFinder class, we use a list 'self.temperatures' to store the recorded temperature readings.
2. The addTemperature method simply appends the given temperature to the list.

3. The `findPeak` method iterates through the list of temperatures, checking if the temperature at the current index is greater than its neighboring temperatures. If so, it updates the `peak` variable with the maximum of the current peak and the temperature at that index.
4. The `findValley` method is similar to `findPeak`, but it checks for temperatures that are lower than their neighbors to find valleys.
5. The `findAverage` method calculates the average temperature by summing up all the temperatures and dividing by the number of readings.
6. Example usage demonstrates how to create a `PeakFinder` object, add temperatures, and call the different methods to find peaks, valleys, and the average.

Please note that this solution iterates through the list to find peaks and valleys, which may not be

the most efficient approach for very large datasets. There are more efficient algorithms for finding peaks and valleys, such as using binary search or divide and conquer.